2nd Edition
Includes Bonus Visual Studio .NET add-in

C#

IN A NUTSHELL

A Desktop Quick Reference

Peter Drayton,
Ben Albahari & Ted Neward

O'REILLY®

C#

IN A NUTSHELL

Other Microsoft .NET resources from O'Reilly

Related titles

- Programming C#
- Programming Visual Basic .NET
- Programming ASP.NET
- ASP.NET in a Nutshell
- ADO.NET in a Nutshell
- .NET Windows Forms in a Nutshell
- .NET Framework Essentials
- Mastering Visual Studio .NET

.NET Books Resource Center

dotnet.oreilly.com is a complete catalog of O'Reilly's books on .NET and related technologies, including sample chapters and code examples.

ONDotnet.com provides independent coverage of fundamental, interoperable and emerging Microsoft .NET programming and web services technologies.

Conferences

O'Reilly & Associates bring diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

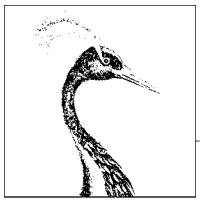
C#

IN A NUTSHELL

Second Edition

Peter Drayton,
Ben Albahari, and Ted Neward

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo



1

Introducing C# and the .NET Framework

C# is a programming language from Microsoft designed specifically to target the .NET Framework. Microsoft's .NET Framework is a runtime environment and class library that dramatically simplifies the development and deployment of modern, component-based applications.

When the .NET Framework and C# language compiler were shipped in final form in January 2002, both the platform and programming language had already garnered much industry attention and widespread use among Microsoft-centric early adopters. Why this level of success? Certainly, the C# language and the .NET Framework address many of the technical challenges facing modern developers as they strive to develop increasingly complex distributed systems with ever-shrinking schedules and team sizes.

However, in addition to its technical merits, one of the main reasons for the success that the language and platform has enjoyed thus far is the unprecedented degree of openness that Microsoft has shown. From July 2000 to January 2002, the .NET Framework underwent an extensive public beta that allowed tens of thousands of developers to "kick the tires" of the programming environment. This allowed Microsoft to both solicit and react to developer community feedback before finalizing the new platform.

Additionally, the key specifications for both the language and the platform have been published, reviewed, and ratified by an international standards organization called the European Computer Manufacturers Association (ECMA). These standardization efforts have led to multiple third-party initiatives that bring the C# language and the .NET platform to non-Microsoft environments. They have also prompted renewed interest among academics in the use of Microsoft technologies as teaching and research vehicles.

Lastly, although the language and platform are shiny and new, the foundations for the C# language and the .NET Framework have been years in the making, reaching back more than half a decade. Understanding where the language and

platform have come from gives us a better understanding of where they are headed.

The C# Language

Reports of a new language from Microsoft first started surfacing in 1998. At that time the language was called COOL, and was said to be very similar to Java. Although Microsoft consistently denied the reports of the new language, rumors persisted.

In June 2000, Microsoft ended the speculation by releasing the specifications for a new language called C# (pronounced "see-sharp"). This was rapidly followed by the release of a preview version of the .NET Framework SDK (which included a C# compiler) at the July 2000 Professional Developer's Conference (PDC) in Orlando, Florida.

The new language was designed by Anders Hejlsberg (creator of Turbo Pascal and architect of Delphi), Scott Wiltamuth, and Peter Golde. Described in the C# Language Specification as a "...simple, modern, object-oriented, and type-safe programming language derived from C and C++," C# bears many syntactic similarities to C++ and Java.

However, focusing on the syntactic similarities between C# and Java does the C# language a disservice. Semantically, C# pushes the language-design envelope substantially beyond where the Java language was circa 2001, and could rightfully be viewed as the next step in the evolution of component-oriented programming languages. While it is outside the scope of this book to perform a detailed comparison between C# and Java, we urge interested readers to read the widely cited article "A Comparative Overview of C# and Java," by co-author Ben Albahari, available at *http://genamics.com/developer/csharp_comparative.htm*.

Enabling Component-Based Development

Over the last 10 years, programming techniques such as object-oriented design, interface-based programming, and component-based software have become ubiquitous. However, programming language support for these constructs has always lagged behind the current state-of-the-art best practices. As a result, developers tend to either depend on programming conventions and custom code rather than direct compiler and runtime support, or to not take advantage of the techniques at all.

As an example, consider that C++ supported object orientation, but had no formal concept of interfaces. C++ developers resorted to abstract base classes and mix-in interfaces to simulate interface-based programming, and relied on external component programming models such as COM or CORBA to provide the benefits of component-based software.

While Java extended C++ to add language-level support for interfaces and packages (among other things), it too had very little language-level support for building long-lived component-based systems (in which one needs to develop, interconnect, deploy, and version components from various sources over an

extended period of time). This is not to say that the Java community hasn't built many such systems, but rather that these needs were addressed by programming conventions and custom code: relying on naming conventions to identify common design patterns such as properties and events, requiring external metadata for deployment information, and developing custom class loaders to provide stronger component versioning semantics.

By comparison, the C# language was designed from the ground up around the assumption that modern systems are built using components. Consequently, C# provides direct language support for common component constructs such as properties, methods, and events (used by RAD tools to build applications out of components, setting properties, responding to events, and wiring components together via method calls). C# also allows developers to directly annotate and extend a component's type information to provide deployment, design, or runtime support, integrate component versioning directly into the programming model, and integrate XML-based documentation directly into C# source files. C# also discards the C++ and COM approach of spreading source artifacts across header files, implementation files, and type libraries in favor of a much simpler source organization and component reuse model.

While this is by no means an exhaustive list, the enhancements in C# over Java and C++ qualify it as the next major step in the evolution of component-based development languages.

A Modern Object-Oriented Language

In addition to deeply integrated support for building component-based systems, C# is also a fully capable object-oriented language, supporting all the common concepts and abstractions that exist in languages such as C++ and Java.

As is expected of any modern object-oriented language, C# supports concepts such as inheritance, encapsulation, polymorphism, and interface-based programming. C# supports common C, C++, and Java language constructs such as classes, structs, interfaces, and enums, as well as more novel constructs such as delegates, which provide a type-safe equivalent to C/C++ function pointers, and custom attributes, which allow annotation of code elements with additional information.

In addition, C# incorporates features from C++ such as operator overloading, user-defined conversions, true rectangular arrays, and pass-by-reference semantics that are currently missing from Java.

Unlike most programming languages, C# has no runtime library of its own. Instead, C# relies on the vast class library in the .NET Framework for all its needs, including console I/O, network and file handling, collection data structures, and many other facilities. Implemented primarily in C# and spanning more than a million lines of code, this class library served as an excellent torture-test during the development cycle for both the C# language and the C# compiler.

The C# language strives to balance the need for consistency and efficiency. Some object-oriented languages (such as Smalltalk) take the viewpoint that "everything is an object." This approach has the advantage that instances of primitive types

(such as integers) are first-class objects. However, it has the disadvantage of being very inefficient. To avoid this overhead, other languages (such as Java) choose to bifurcate the type system into primitives and everything else, leading to less overhead, but also to a schism between primitive and user-defined types.

C# balances these two conflicting viewpoints by presenting a unified type system in which all types (including primitive types) are derived from a common base type, while simultaneously allowing for performance optimizations that allow primitive types and simple user-defined types to be treated as raw memory, with minimal overhead and increased efficiency.

Building Robust and Durable Software

In a world of always-on connectivity and distributed systems, software robustness takes on new significance. Servers need to stay up and running 24×7 to service clients, and clients need to be able to download code off the network and run it locally with some guarantee that it will not misbehave. The C# language (in concert with the .NET Framework) promotes software robustness in a number of different ways.

First and foremost, C# is a type-safe language, meaning that programs are prevented from accessing objects in inappropriate ways. All code and data is associated with a type, all objects have an associated type, and only operations defined by the associated type can be performed on an object. Type-safety eliminates an entire category of errors in C and C++ programs stemming from invalid casts, bad pointer arithmetic, and even malicious code.

C# also provides automatic memory management in the form of a high-performance tracing generational garbage collector. This frees programmers from performing manual memory management or reference counting, and eliminates an entire category of errors, such as dangling pointers, memory leaks, and circular references.

Even good programs can have bad things happen to them, and it is important to have a consistent mechanism for detecting errors. Over the years, Windows developers have had to contend with numerous error reporting mechanisms, such as simple failure return codes, Win32 structured exceptions, C++ exceptions, COM error `HResults`, and OLE automation `IErrorInfo` objects. This proliferation of approaches breeds complexity and makes it difficult for designers to create standardized error-handling strategies. The .NET Framework eliminates this complexity by standardizing on a single exception-handling mechanism that is used throughout the framework, and exposed in all .NET languages including C#.

The C# language design also includes numerous other features that promote robustness, such as language-level support for independently versioning base classes (without changing derived class semantics or mandating recompilation of derived classes), detection of attempts to use uninitialized variables, array bounds checking, and support for checked arithmetic.

A Pragmatic World View

Many of the design decisions in the C# language represent a pragmatic world view on the part of the designers. For example, the syntax was selected to be familiar to C, C++, and Java developers, making it easier to learn C# and aiding source code porting.

While C# provides many useful, high-level object-oriented features, it recognizes that in certain limited cases these features can work against raw performance. Rather than dismiss these concerns as unimportant, C# includes explicit support for features such as direct pointer manipulation, unsafe type casts, declarative pinning of garbage-collected objects, and direct memory allocation on the stack. Naturally, these features come at a cost, both in terms of the complexity they add and the elevated security privileges required to use them. However, the existence of these features gives C# programmers much more headroom than other, more restrictive languages do.

Lastly, the interop facilities in the .NET Framework make it easy to leverage existing DLLs and COM components from C# code, and to use C# components in classic COM applications. Although not strictly a function of the C# language, this capability reflects a similarly pragmatic world view, in which new functionality coexists peacefully with legacy code for as long as needed.

The .NET Framework

The Microsoft .NET Framework consists of two elements: a runtime environment called the Common Language Runtime (CLR), and a class library called the Framework Class Library (FCL). The FCL is built on top of the CLR and provides services needed by modern applications.

While applications targeting the .NET Framework interact directly with the FCL, the CLR serves as the underlying engine. In order to understand the .NET Framework, one first must understand the role of the CLR.

The Common Language Runtime

The CLR is a modern runtime environment that manages the execution of user code, providing services such as JIT compilation, memory management, exception management, debugging and profiling support, and integrated security and permission management.

Essentially, the CLR represents the foundation of Microsoft's computing platform for the next decade. However, it has been a long time in the making. Its origins can be traced back to early 1997, when products such as Microsoft Transaction Server (MTS) were starting to deliver on the promise of a more declarative, service-oriented programming model. This new model allowed developers to declaratively annotate their components at development time, and then rely on the services of a runtime (such as MTS) to hijack component activation and intercept method calls, transparently layering in additional services such as transactions, security, just-in-time (JIT) activation, and more. This need to augment COM type information pushed the limits of what was possible and

useful with IDL and type libraries. The COM+ team set out to find a generalized solution to this problem.

The first public discussion of a candidate solution occurred at the 1997 PDC in San Diego, when Mary Kirtland and other members of the COM+ team discussed a future version of COM centered on something called the COM+ Runtime, and providing many of the services such as extensible type information, cross-language integration, implementation inheritance, and automatic memory management that ultimately resurfaced in the CLR.*

Soon after the 1997 PDC, Microsoft stopped talking publicly about the technology, and the product known as COM+ that was released with Windows 2000 bore little resemblance to the COM+ Runtime originally described. Behind the scenes, however, work was continuing and the scope of the project was expanding significantly as it took on a much larger role within Microsoft.

Initially codenamed Lightning, the project underwent many internal (and some external) renamings, and was known at various times as COM3, COM+ 2.0, the COM+ Runtime, the NGWS Runtime, the Universal Runtime (URT), and the Common Language Runtime. This effort ultimately surfaced as the .NET Framework, announced at the July 2000 PDC in Orlando, Florida. During the following 18 months, the .NET Framework underwent an extensive public beta, culminating in the release of Version 1.0 of the Microsoft .NET Framework on January 15, 2002.

Compilation and Execution Model

To better understand the CLR, consider how compilers that target the .NET Framework differ from traditional compilers.

Traditional compilers target a specific processor, consuming source files in a specific language, and producing binary files containing streams of instructions in the native language of the target processor. These binary files may then be executed directly on the target processor.

.NET compilers function a little differently, as they do not target a specific native processor. Instead, they consume source files and produce binary files containing an intermediate representation of the source constructs, expressed as a combination of metadata and Common Intermediate Language (CIL). In order for these binaries to be executed, the CLR must be present on the target machine.

When these binaries are executed, they cause the CLR to load. The CLR then takes over and manages execution, providing a range of services such as JIT compilation (converting the CIL as needed into the correct stream of instructions for the underlying processor), memory management (in the form of a garbage collector), exception management, debugger and profiler integration, and security services (stack walking and permission checks).

* Two sites, *http://www.microsoft.com/msj/defaulttop.asp?page=/msj/1197/inthisissuefeatures1197.htm* and *http://www.microsoft.com/msj/defaulttop.asp?page=/msj/1297/inthisissuefeatures1297.htm*, contain articles by Mary Kirtland on the COM+ Runtime.

This compilation and execution model explains why C# is referred to as a *managed language*, why code running in the CLR is referred to as *managed code*, and why the CLR is said to provide *managed execution*.

Although this dependency on a runtime environment might initially appear to be a drawback, substantial benefits arise from this architecture. Since the metadata and CIL representations are processor architecture–neutral, binaries may be used on any machine in which the Common Language Runtime is present, regardless of underlying processor architecture. Additionally, since processor-specific code generation is deferred until runtime, the CLR has the opportunity to perform processor-specific optimizations based on the target architecture the code is running on. As processor technology advances, all applications need to take advantage of these advances is an updated version of the CLR.

Unlike traditional binary representations, which are primarily streams of native processor instructions, the combination of metadata and CIL retains almost all of the original source language constructs. In addition, this representation is source language-neutral, which allows developers to build applications using multiple source languages. They can select the best language for a particular task, rather than being forced to standardize on a particular source language for each application or needing to rely on component technologies, such as COM or CORBA, to mask the differences between the source languages used to build the separate components of an application.

The Common Type System

Ultimately, the CLR exists to safely execute managed code, regardless of source language. In order to provide for cross-language integration, to ensure type safety, and to provide managed execution services such as JIT compilation, garbage collection, exception management, etc., the CLR needs intimate knowledge of the managed code that it is executing.

To meet this requirement, the CLR defines a shared type system called the Common Type System (CTS). The CTS defines the rules by which all types are declared, defined and managed, regardless of source language. The CTS is designed to be rich and flexible enough to support a wide variety of source languages, and is the basis for cross-language integration, type safety, and managed execution services.

Compilers for managed languages that wish to be first-class citizens in the world of the CLR are responsible for mapping source language constructs onto the CTS analogs. In cases in which there is no direct analog, the language designers may decide to either adapt the source language to better match the CTS (ensuring more seamless cross-language integration), or to provide additional plumbing that preserves the original semantics of the source language (possibly at the expense of cross-language integration capabilities).

Since all types are ultimately represented as CTS types, it now becomes possible to combine types authored in different languages in new and interesting ways. For example, since managed languages ultimately declare CTS types, and the CTS supports inheritance, it follows that the CLR supports cross-language inheritance.

The Common Language Specification

Not all languages support the exact same set of constructs, and this can be a barrier to cross-language integration. Consider this example: Language A allows unsigned types (which are supported by the CTS), while Language B does not. How should code written in Language B call a method written in Language A, which takes an unsigned integer as a parameter?

The solution is the Common Language Specification (CLS). The CLS defines the reasonable subset of the CTS that should be sufficient to support cross-language integration, and specifically excludes problem areas such as unsigned integers, operator overloading, and more.

Each managed language decides how much of the CTS to support. Languages that can consume any CLS-compliant type are known as CLS Consumers. Languages which can extend any existing CLS-compliant type are known as CLS Extenders. Naturally, managed languages are free to support CTS features over and above the CLS, and most do. As an example, the C# language is both a CLS Consumer and a CLS Extender, and supports all of the important CTS features.

The combination of the rich and flexible CTS and the widely supported CLS has led to many languages being adapted to target the .NET platform. At the time of this writing, Microsoft was offering compilers for six managed languages (C#, VB.NET, JScript, Managed Extensions for C++, Microsoft IL, and J#), and a host of other commercial vendors and academics were offering managed versions of languages, such as COBOL, Eiffel, Haskell, Mercury, Mondrian, Oberon, Forth, Scheme, Smalltalk, APL, several flavors of Pascal, and more.

Given the level of interest from industry and academia, one might say that .NET has spawned something of a renaissance in programming-language innovation.

The Framework Class Library

Developer needs (and Windows capabilities) have evolved much since Windows 1.0 was introduced in November 1985. As Windows has grown to meet new customer needs, the accompanying APIs have grown by orders of magnitude over time, becoming ever more complex, increasingly inconsistent, and almost impossible to comprehend in their totality.

Additionally, while modern paradigms such as object orientation, component software, and Internet standards had emerged and, in many cases, joined the mainstream, these advances had not yet been incorporated into the Windows programming model in a comprehensive and consistent manner.

Given the issues, and the degree of change already inherent in the move to a managed execution environment, the time was ripe for a clean start. As a result, the .NET Framework replaces most (though not all) of the traditional Windows API sets with a well-factored, object-oriented class library called the Framework Class Library (FCL).

The FCL provides a diverse array of higher-level software services, addressing the needs of modern applications. Conceptually, these can be grouped into several categories such as:

- Support for core functionality, such as interacting with basic data types and collections; console, network and file I/O; and interacting with other runtime-related facilities
- Support for interacting with databases, consuming and producing XML, and manipulating tabular and tree-structured data
- Support for building web-based (thin client) applications with a rich server-side event model
- Support for building desktop-based (thick client) applications with broad support for the Windows GUI
- Support for building SOAP-based XML web services

The FCL is vast, including more than 3,500 classes. For a more detailed overview of the facilities in the FCL, see Chapter 5.

ECMA Standardization

One of the most encouraging aspects about the .NET Framework is the degree of openness that Microsoft has shown during its development. From the earliest public previews, core specifications detailing the C# language, the classes in the FCL, and the inner workings of the CLR have been freely available.

However, this openness was taken to a new level in November 2000 when Microsoft, along with co-sponsors Intel and HP, officially submitted the specifications for the C# language, a subset of the FCL, and the runtime environment to ECMA for standardization.

This action began an intense standardization process. Organizations participating in the effort included Microsoft, HP, Intel, IBM, Fujitsu Software, ISE, Plum Hall, Monash University, and others. The work was performed under the auspices of ECMA technical committee TC39, the same committee that had previously standardized the JavaScript language as ECMAScript.

TC39 chartered two new task groups to perform the actual standardization work: one to focus on the C# language, the other to focus on what became known as the Common Language Infrastructure (CLI).

The CLI consisted of the runtime engine and the subset of the FCL being standardized. Conceptually, Microsoft's CLR is intended to be a conforming commercial implementation of the runtime engine specified in the CLI, and Microsoft's FCL is intended to be a conforming commercial implementation of the class library specified in the CLI (although obviously, it is a massive superset of the 294 classes ultimately specified in the CLI).

After more than a year of intense effort, the task groups completed their standardization work and presented the specifications to the ECMA General Assembly. In December 2001 the General Assembly ratified the C# and CLI specifications,

assigning them the ECMA standards numbers of ECMA-334 (C#) and ECMA-335 (the CLI). In late December, 2001, ECMA submitted the standards to the International Organization for Standardization (ISO) via the Fast-Track process, and in April 2003, ISO ratified the standards as ISO/IEC 23270 (C#) and ISO/IEC 23271 (CLI), giving C# and the CLI bona-fide international standard status.

Critics have claimed that the standardization process was merely a ploy by Microsoft to deflect Java's cross-platform advantages. However, the qualifications and seniority of the people working on the standardization effort, and their level of involvement during the lengthy standardization cycle, tell a different story. Microsoft, along with its co-sponsors and the other members of the standardization task groups, committed some of its best and brightest minds to this effort, spending a huge amount of time and attention on the standardization process. Given that this effort occurred concurrently with the development and release of the .NET Framework itself, this level of investment by Microsoft and others flies in the face of the conspiracy theories.

Of course, for standards to have an impact there must be implementations. In addition to the commercial .NET Framework, Microsoft itself has two other CLI implementations: the Shared Source CLI (SSCLI) and the .NET Compact Framework.

In November 2002 Microsoft released the Shared Source CLI (SSCLI) (*http://msdn.microsoft.com/net/sscli*). Formerly known by its code-name, Rotor, the SSCLI is source code for a working implementation of the CLI and C# standards that builds and runs on Windows XP, FreeBSD, and Mac OS X. Licensed for non-commercial use, the SSCLI provides academics, researchers, and technology enthusiasts a vehicle for teaching and research on the inner workings of a CLI implementation, and it also serves as a valuable resource for CLI implementers. Community response has been very positive: a substantial number of research projects and teaching efforts have been based on the SSCLI, and in April 2003 Cornell University launched a community site (*http://www.sscli.net*) that includes a CVS repository, discussion lists, and a bug tracker to coordinate these efforts.

The .NET Compact Framework, now available with Visual Studio .NET 2003, is an implementation of the CLI designed to target resource-constrained devices, and is initially available for Pocket PC 2000+ and Windows CE 4.1+ operating systems.

However, Microsoft's implementations are not the only game in town. Other CLI implementations include the Mono project and dotGNU.

The Mono project (*http://www.go-mono.com*), started by Ximian Corporation, is aiming to provide an implementation of not only the CLI platform and the C# compiler, but also a larger set of classes selected from Microsoft's .NET Framework FCL. In addition to the internal resources that Ximian has committed to the project, the Mono project has also attracted attention from the broader open source community, and appears to be gathering steam.

Another community effort to create an implementation of the C# and CLI standards is the dotGNU project (*http://www.dotgnu.org*). While not as high-profile as Mono, dotGNU has also been making headway, and includes some interesting and unique concepts. The core of dotGNU is Portable.NET, which was originally

developed by a lone developer (Rhys Weatherley) before merging his project with dotGNU in August 2000. There are unique aspects to the dotGNU project, including the fact that it was originally designed around a CIL interpreter rather than a JIT compiler, and the developers' plan to support directly executing Java binaries.

It is very likely that more implementations of the CLI will arise over time. While it is too early to say whether the .NET Framework (in the form of the CLI) will ever be available on as many platforms as Java is, the degree of openness and the level of community interest is very encouraging.

Changes in Visual C# 2003

There have been a few minor additions and changes to the C# language in Visual Studio .NET 2003. Most of these changes were made to tighten the language, to conform more precisely to the ECMA specification of C#, and to address bugs in control flow optimizations made by the compiler. These changes have been included in the language reference. (For example, documentation comments may now use standard `/* ... */` notation. See Chapter 4.)

The 1.1 version of the Framework Class Libraries has some minor changes from 1.0, which are reflected in the updated API reference section of this book. With the exception of signature changes on two methods on the `XslTransform` class, there are no breaking changes that would prevent a 1.0 application from working on the 1.1 Framework.