O'REILLY®

6th Edition

Covers .NET 4.6 & the Roslyn Compiler

Free Sampler

C# 6.0 in a Nutshell

THE DEFINITIVE REFERENCE

Joseph Albahari & Ben Albahari

O'REILLY®

C# 6.0 in a Nutshell

When you have questions about C# 6.0 or the .NET CLR and its core Framework assemblies, this bestselling guide has the answers you need. C# has become a language of unusual flexibility and breadth since its premiere in 2000, but this continual growth means there's still much more to learn.

Organized around concepts and use cases, this thoroughly updated sixth edition provides intermediate and advanced programmers with a concise map of C# and .NET knowledge. Dive in and discover why this Nutshell guide is considered the definitive reference on C#.

- Get up to speed with all aspects of the C# language, from the basics of syntax and variables, to advanced topics such as pointers and operator overloading
- Dig deep into LINQ via three chapters dedicated to the topic
- Learn about dynamic, asynchronous, and parallel programming
- Work with .NET features, including XML, networking, serialization, reflection, security, application domains, and code contracts
- Explore the new C# 6.0 compiler-as-a-service, Roslyn

"*C# 6.0 in a Nutshell* is one of the few books I keep on my desk as a quick reference."

—**Scott Guthrie**
Microsoft

"Novices and experts alike will find the latest techniques in C# programming here."

—**Eric Lippert**
C# MVP

Joseph Albahari, author of *C# 5.0 in a Nutshell*, *C# 5.0 Pocket Reference*, and *LINQ Pocket Reference*, also wrote LINQPad, the popular code scratchpad and LINQ querying utility..

Ben Albahari, a former program manager at Microsoft, is cofounder of Auditionist, a casting website for actors in the UK.

C#/MICROSOFT .NET

US $59.99 CAN $68.99
ISBN: 978-1-491-92706-9

Twitter: @oreillymedia
facebook.com/oreilly

Want to read more?

You can buy this book at oreilly.com in print and ebook format.

Buy 2 books, get the 3rd FREE!

Use discount code OPC10

All orders over $29.95 qualify for **free shipping** within the US.

It's also available at your favorite book retailer, including the iBookstore, the Android Marketplace, and Amazon.com.

O'REILLY®

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 15055

C# 6.0 in a Nutshell

by Joseph Albahari and Ben Albahari

Copyright © 2016 Joseph Albahari and Ben Albahari. All rights reserved.

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

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

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *C# 6.0 in a Nutshell,* the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-92706-9

[M]

Table of Contents

Preface xi

1. Introducing C# and the .NET Framework 1

- Object Orientation 1
- Type Safety 2
- Memory Management 3
- Platform Support 3
- C#'s Relationship with the CLR 3
- The CLR and .NET Framework 3
- C# and Windows Runtime 5
- What's New in C# 6.0 6
- What Was New in C# 5.0 8
- What Was New in C# 4.0 8
- What Was New in C# 3.0 9

2. C# Language Basics 11

- A First C# Program 11
- Syntax 14
- Type Basics 17
- Numeric Types 26
- Boolean Type and Operators 33
- Strings and Characters 35
- Arrays 38
- Variables and Parameters 42
- Expressions and Operators 51
- Null Operators 55
- Statements 56
- Namespaces 65

3. Creating Types in C# 73

Classes 73
Inheritance 88
The object Type 97
Structs 101
Access Modifiers 102
Interfaces 104
Enums 109
Nested Types 113
Generics 114

4. Advanced C#. 127
Delegates 127
Events 136
Lambda Expressions 143
Anonymous Methods 147
try Statements and Exceptions 148
Enumeration and Iterators 156
Nullable Types 162
Operator Overloading 168
Extension Methods 171
Anonymous Types 174
Dynamic Binding 175
Attributes 183
Caller Info Attributes (C# 5) 185
Unsafe Code and Pointers 187
Preprocessor Directives 190
XML Documentation 193

5. Framework Overview. 199
The CLR and Core Framework 202
Applied Technologies 206

6. Framework Fundamentals. 213
String and Text Handling 213
Dates and Times 226
Dates and Time Zones 234
Formatting and Parsing 240
Standard Format Strings and Parsing Flags 246
Other Conversion Mechanisms 253
Globalization 257
Working with Numbers 258
Enums 262

Tuples 266
The Guid Struct 267
Equality Comparison 267
Order Comparison 278
Utility Classes 281

7. Collections. 285
Enumeration 285
The ICollection and IList Interfaces 293
The Array Class 297
Lists, Queues, Stacks, and Sets 305
Dictionaries 314
Customizable Collections and Proxies 321
Plugging in Equality and Order 327

8. LINQ Queries. 335
Getting Started 335
Fluent Syntax 337
Query Expressions 344
Deferred Execution 348
Subqueries 355
Composition Strategies 358
Projection Strategies 362
Interpreted Queries 364
LINQ to SQL and Entity Framework 371
Building Query Expressions 385

9. LINQ Operators. 391
Overview 393
Filtering 396
Projecting 400
Joining 412
Ordering 420
Grouping 423
Set Operators 426
Conversion Methods 427
Element Operators 430
Aggregation Methods 432
Quantifiers 437
Generation Methods 438

10. LINQ to XML. 441

Architectual Overview 441
X-DOM Overview 442
Instantiating an X-DOM 446
Navigating and Querying 448
Updating an X-DOM 453
Working with Values 456
Documents and Declarations 459
Names and Namespaces 463
Annotations 468
Projecting into an X-DOM 469

11. Other XML Technologies. 477
XmlReader 478
XmlWriter 487
Patterns for Using XmlReader/XmlWriter 489
XSD and Schema Validation 493
XSLT 496

12. Disposal and Garbage Collection. 499
IDisposable, Dispose, and Close 499
Automatic Garbage Collection 505
Finalizers 507
How the Garbage Collector Works 512
Managed Memory Leaks 516
Weak References 520

13. Diagnostics and Code Contracts. 525
Conditional Compilation 525
Debug and Trace Classes 529
Code Contracts Overview 532
Preconditions 537
Postconditions 541
Assertions and Object Invariants 543
Contracts on Interfaces and Abstract Methods 545
Dealing with Contract Failure 546
Selectively Enforcing Contracts 548
Static Contract Checking 549
Debugger Integration 551
Processes and Process Threads 552
StackTrace and StackFrame 553
Windows Event Logs 555
Performance Counters 557

The Stopwatch Class 562

14. Concurrency and Asynchrony. 563
Introduction 563
Threading 564
Tasks 581
Principles of Asynchrony 589
Asynchronous Functions in C# 594
Asynchronous Patterns 610
Obsolete Patterns 618

15. Streams and I/O. 623
Stream Architecture 623
Using Streams 625
Stream Adapters 639
Compression Streams 647
Working with ZIP Files 649
File and Directory Operations 650
File I/O in Windows Runtime 661
Memory-Mapped Files 663
Isolated Storage 666

16. Networking. 673
Network Architecture 673
Addresses and Ports 675
URIs 676
Client-Side Classes 679
Working with HTTP 692
Writing an HTTP Server 698
Using FTP 701
Using DNS 703
Sending Mail with SmtpClient 703
Using TCP 704
Receiving POP3 Mail with TCP 708
TCP in Windows Runtime 709

17. Serialization. 713
Serialization Concepts 713
The Data Contract Serializer 717
Data Contracts and Collections 727
Extending Data Contracts 730
The Binary Serializer 733

Binary Serialization Attributes 735
Binary Serialization with ISerializable 738
XML Serialization 742

18. Assemblies. 753
What's in an Assembly 753
Strong Names and Assembly Signing 758
Assembly Names 761
Authenticode Signing 764
The Global Assembly Cache 768
Resources and Satellite Assemblies 770
Resolving and Loading Assemblies 779
Deploying Assemblies Outside the Base Folder 784
Packing a Single-File Executable 785
Working with Unreferenced Assemblies 787

19. Reflection and Metadata. 789
Reflecting and Activating Types 790
Reflecting and Invoking Members 797
Reflecting Assemblies 810
Working with Attributes 812
Dynamic Code Generation 818
Emitting Assemblies and Types 825
Emitting Type Members 828
Emitting Generic Methods and Types 834
Awkward Emission Targets 836
Parsing IL 840

20. Dynamic Programming. 847
The Dynamic Language Runtime 847
Numeric Type Unification 849
Dynamic Member Overload Resolution 850
Implementing Dynamic Objects 856
Interoperating with Dynamic Languages 859

21. Security. 863
Permissions 863
Code Access Security (CAS) 868
Allowing Partially Trusted Callers 871
The Transparency Model 873
Sandboxing Another Assembly 881
Operating System Security 885

Identity and Role Security 888
Cryptography Overview 889
Windows Data Protection 890
Hashing 891
Symmetric Encryption 892
Public Key Encryption and Signing 897

22. Advanced Threading... 903
Synchronization Overview 904
Exclusive Locking 904
Locking and Thread Safety 912
Nonexclusive Locking 918
Signaling with Event Wait Handles 923
The Barrier Class 932
Lazy Initialization 933
Thread-Local Storage 936
Interrupt and Abort 938
Suspend and Resume 939
Timers 940

23. Parallel Programming... 945
Why PFX? 945
PLINQ 948
The Parallel Class 961
Task Parallelism 968
Working with AggregateException 978
Concurrent Collections 980
BlockingCollection<T> 983

24. Application Domains... 989
Application Domain Architecture 989
Creating and Destroying Application Domains 990
Using Multiple Application Domains 992
Using DoCallBack 994
Monitoring Application Domains 995
Domains and Threads 995
Sharing Data Between Domains 997

25. Interoperability... 1003
Calling into Native DLLs 1003
Type Marshaling 1004
Callbacks from Unmanaged Code 1007

Simulating a C Union 1007
Shared Memory 1008
Mapping a Struct to Unmanaged Memory 1011
COM Interoperability 1015
Calling a COM Component from C# 1017
Embedding Interop Types 1020
Primary Interop Assemblies 1021
Exposing C# Objects to COM 1022

26. Regular Expressions. 1023
Regular Expression Basics 1024
Quantifiers 1028
Zero-Width Assertions 1029
Groups 1032
Replacing and Splitting Text 1033
Cookbook Regular Expressions 1035
Regular Expressions Language Reference 1038

27. The Roslyn Compiler. 1043
Roslyn Architecture 1044
Syntax Trees 1045
Compilations and Semantic Models 1060

Index. 1073

2

C# Language Basics

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 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 forward 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 see 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 *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 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) {...}
```

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 38.

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" on page 17.

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 (*refer-*

enced) 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*.

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


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`.

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

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`.

Punctuators help demarcate the structure of the program. These are the punctuators we used in our example program:

```
{  }  ;
```

The braces group multiple statements into a *statement block*.

The semicolon terminates a statement. (Statement blocks, however, do not require a semicolon.) Statements can wrap multiple lines:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

An *operator* transforms and combines expressions. Most operators in C# are denoted with a symbol, such as the multiplication operator, `*`. We will discuss operators in more detail later in this chapter. These are the operators we used in our example program:

```
.  ()  *  =
```

A period denotes a member of something (or a decimal point with numeric literals). Parentheses are used when declaring or calling a method; empty parentheses are used when the method accepts no arguments. (Parentheses also have other purposes that we'll see later in this chapter.) An equals sign performs *assignment*. (The double equals sign, ==, performs equality comparison, as we'll see later.)

Comments

C# offers two different styles of source-code documentation: *single-line comments* and *multiline comments*. A single-line comment begins with a double forward slash and continues until the end of the line. For example:

```
int x = 3;   // Comment about assigning 3 to x
```

A multiline comment begins with `/*` and ends with `*/`. For example:

```
int x = 3;   /* This is a comment that
                spans two lines */
```

Comments may embed XML documentation tags, explained in "XML Documentation" on page 193 in Chapter 4.

Type Basics

A *type* defines the blueprint for a value. In our example, we used two literals of type `int` with values 12 and 30. We also declared a *variable* of type `int` whose name was `x`:

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

A *variable* denotes a storage location that can contain different values over time. In contrast, a *constant* always represents the same value (more on this later):

```
const int y = 360;
```

All values in C# are *instances* of a type. The meaning of a value, and the set of possible values a variable can have, is determined by its type.

Predefined Type Examples

Predefined types are types that are specially supported by the compiler. The `int` type is a predefined type for representing the set of integers that fit into 32 bits of memory, from -2^{31} to $2^{31}-1$, and is the default type for numeric literals within this range. We can perform functions such as arithmetic with instances of the `int` type as follows:

```
int x = 12 * 30;
```

Another predefined C# type is `string`. The `string` type represents a sequence of characters, such as ".NET" or "*http://oreilly.com*." We can work with strings by calling functions on them as follows:

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage);               // HELLO WORLD

int x = 2015;
message = message + x.ToString();
Console.WriteLine (message);                    // Hello world2015
```

The predefined `bool` type has exactly two possible values: `true` and `false`. The `bool` type is commonly used to conditionally branch execution flow based with an `if` statement. For example:

```
bool simpleVar = false;
if (simpleVar)
```

```
  Console.WriteLine ("This will not print");

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");
```

In C#, predefined types (also referred to as built-in types) are recognized with a C# keyword. The `System` namespace in the .NET Framework contains many important types that are not predefined by C# (e.g., `DateTime`).

Custom Type Examples

Just as we can build complex functions from simple functions, we can build complex types from primitive types. In this example, we will define a custom type named `UnitConverter`—a class that serves as a blueprint for unit conversions:

```
using System;

public class UnitConverter
{
  int ratio;                                                // Field
  public UnitConverter (int unitRatio) {ratio = unitRatio; } // Constructor
  public int Convert   (int unit)    {return unit * ratio; } // Method
}

class Test
{
  static void Main()
  {
    UnitConverter feetToInchesConverter = new UnitConverter (12);
    UnitConverter milesToFeetConverter  = new UnitConverter (5280);

    Console.WriteLine (feetToInchesConverter.Convert(30));    // 360
    Console.WriteLine (feetToInchesConverter.Convert(100));   // 1200
    Console.WriteLine (feetToInchesConverter.Convert(
                         milesToFeetConverter.Convert(1)));   // 63360
  }
}
```

Members of a type

A type contains *data members* and *function members*. The data member of `UnitConverter` is the *field* called `ratio`. The function members of `UnitConverter` are the `Convert` method and the `UnitConverter`'s *constructor*.

Symmetry of predefined types and custom types

A beautiful aspect of C# is that predefined types and custom types have few differences. The predefined `int` type serves as a blueprint for integers. It holds data—32 bits—and provides function members that use that data, such as `ToString`. Simi-

larly, our custom `UnitConverter` type acts as a blueprint for unit conversions. It holds data—the ratio—and provides function members to use that data.

Constructors and instantiation

Data is created by *instantiating* a type. Predefined types can be instantiated simply by using a literal such as `12` or `"Hello world"`. The `new` operator creates instances of a custom type. We created and declared an instance of the `UnitConverter` type with this statement:

```
UnitConverter feetToInchesConverter = new UnitConverter (12);
```

Immediately after the `new` operator instantiates an object, the object's *constructor* is called to perform initialization. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type:

```
public class UnitConverter
{
  ...
  public UnitConverter (int unitRatio) { ratio = unitRatio; }
  ...
}
```

Instance versus static members

The data members and function members that operate on the *instance* of the type are called instance members. The `UnitConverter`'s `Convert` method and the `int`'s `ToString` method are examples of instance members. By default, members are instance members.

Data members and function members that don't operate on the instance of the type, but rather on the type itself, must be marked as `static`. The `Test.Main` and `Console.WriteLine` methods are static methods. The `Console` class is actually a *static class*, which means *all* its members are static. You never actually create instances of a `Console`—one console is shared across the whole application.

Let's contrast instance from static members. In the following code, the instance field `Name` pertains to an instance of a particular `Panda`, whereas `Population` pertains to the set of all `Panda` instances:

```
public class Panda
{
  public string Name;             // Instance field
  public static int Population;   // Static field

  public Panda (string n)         // Constructor
  {
    Name = n;                     // Assign the instance field
    Population = Population + 1;  // Increment the static Population field
  }
}
```

The following code creates two instances of the Panda, prints their names, and then prints the total population:

```
using System;

class Test
{
  static void Main()
  {
    Panda p1 = new Panda ("Pan Dee");
    Panda p2 = new Panda ("Pan Dah");

    Console.WriteLine (p1.Name);      // Pan Dee
    Console.WriteLine (p2.Name);      // Pan Dah

    Console.WriteLine (Panda.Population);   // 2
  }
}
```

Attempting to evaluate `p1.Population` or `Panda.Name` will generate a compile-time error.

The public keyword

The `public` keyword exposes members to other classes. In this example, if the `Name` field in `Panda` was not marked as public, it would be private, and the `Test` class could not access it. Marking a member `public` is how a type communicates: "Here is what I want other types to see—everything else is my own private implementation details." In object-oriented terms, we say that the public members *encapsulate* the private members of the class.

Conversions

C# can convert between instances of compatible types. A conversion always creates a new value from an existing one. Conversions can be either *implicit* or *explicit*: implicit conversions happen automatically, and explicit conversions require a *cast*. In the following example, we *implicitly* convert an `int` to a `long` type (which has twice the bitwise capacity of an `int`) and *explicitly* cast an `int` to a `short` type (which has half the capacity of an `int`):

```
int x = 12345;       // int is a 32-bit integer
long y = x;          // Implicit conversion to 64-bit integer
short z = (short)x;  // Explicit conversion to 16-bit integer
```

Implicit conversions are allowed when both of the following are true:

- The compiler can guarantee they will always succeed.
- No information is lost in conversion.[1]

Conversely, *explicit* conversions are required when one of the following is true:

- The compiler cannot guarantee they will always succeed.
- Information may be lost during conversion.

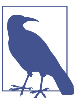
(If the compiler can determine that a conversion will *always* fail, both kinds of conversion are prohibited. Conversions that involve generics can also fail in certain conditions—see "Type Parameters and Conversions" on page 121 in Chapter 3.)

The *numeric conversions* that we just saw are built into the language. C# also supports *reference conversions* and *boxing conversions* (see Chapter 3) as well as *custom conversions* (see "Operator Overloading" on page 168 in Chapter 4). The compiler doesn't enforce the aforementioned rules with custom conversions, so it's possible for badly designed types to behave otherwise.

Value Types Versus Reference Types

All C# types fall into the following categories:

- Value types
- Reference types
- Generic type parameters
- Pointer types



In this section, we'll describe value types and reference types. We'll cover generic type parameters in "Generics" on page 114 in Chapter 3, and pointer types in "Unsafe Code and Pointers" on page 187 in Chapter 4.

Value types comprise most built-in types (specifically, all numeric types, the `char` type, and the `bool` type), as well as custom `struct` and `enum` types.

Reference types comprise all class, array, delegate, and interface types. (This includes the predefined `string` type.)

The fundamental difference between value types and reference types is how they are handled in memory.

1 A minor caveat is that very large `long` values lose some precision when converted to `double`.

Value types

The content of a *value type* variable or constant is simply a value. For example, the content of the built-in value type, `int`, is 32 bits of data.

You can define a custom value type with the `struct` keyword (see Figure 2-1):

```
public struct Point { public int X; public int Y; }
```

or more tersely:

```
public struct Point { public int X, Y; }
```

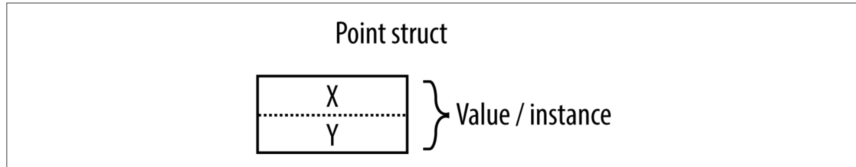


Figure 2-1. A value-type instance in memory

The assignment of a value-type instance always *copies* the instance. For example:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;              // Assignment causes copy

  Console.WriteLine (p1.X);   // 7
  Console.WriteLine (p2.X);   // 7

  p1.X = 9;                   // Change p1.X

  Console.WriteLine (p1.X);   // 9
  Console.WriteLine (p2.X);   // 7
}
```

Figure 2-2 shows that `p1` and `p2` have independent storage.

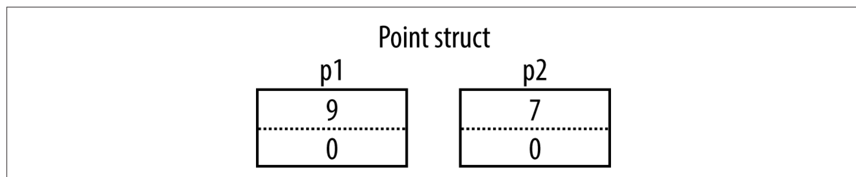


Figure 2-2. Assignment copies a value-type instance

Reference types

A reference type is more complex than a value type, having two parts: an *object* and the *reference* to that object. The content of a reference-type variable or constant is a

reference to an object that contains the value. Here is the `Point` type from our previous example rewritten as a class, rather than a `struct` (shown in Figure 2-3):

```
public class Point { public int X, Y; }
```

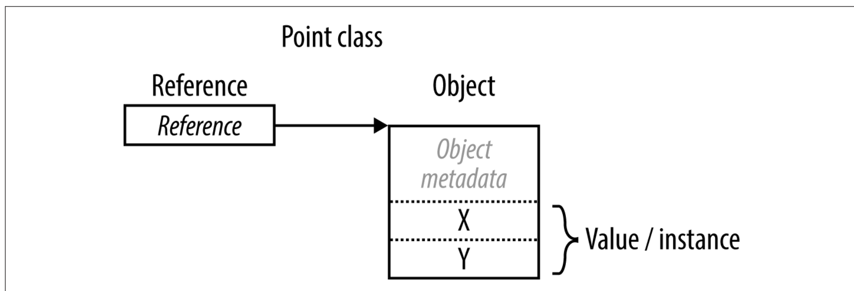


Figure 2-3. A reference-type instance in memory

Assigning a reference-type variable copies the reference, not the object instance. This allows multiple variables to refer to the same object—something not ordinarily possible with value types. If we repeat the previous example, but with `Point` now a class, an operation to `p1` affects `p2`:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;             // Copies p1 reference

  Console.WriteLine (p1.X);  // 7
  Console.WriteLine (p2.X);  // 7

  p1.X = 9;                  // Change p1.X

  Console.WriteLine (p1.X);  // 9
  Console.WriteLine (p2.X);  // 9
}
```

Figure 2-4 shows that `p1` and `p2` are two references that point to the same object.

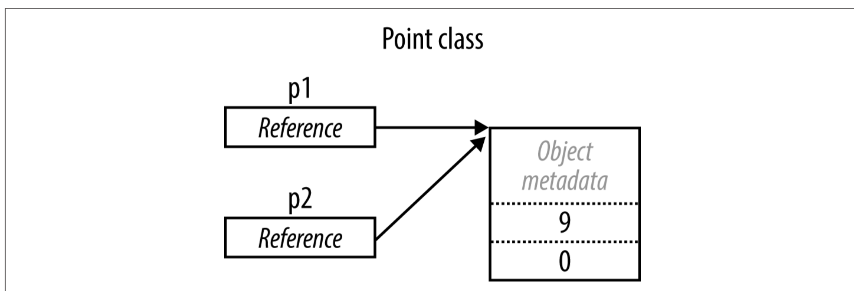


Figure 2-4. Assignment copies a reference

Null

A reference can be assigned the literal `null`, indicating that the reference points to no object:

```
class Point {...}
...

Point p = null;
Console.WriteLine (p == null);   // True

// The following line generates a runtime error
// (a NullReferenceException is thrown):
Console.WriteLine (p.X);
```

In contrast, a value type cannot ordinarily have a null value:

```
struct Point {...}
...

Point p = null;  // Compile-time error
int x = null;    // Compile-time error
```

C# also has a construct called *nullable types* for representing value-type nulls (see "Nullable Types" on page 162 in Chapter 4).

Storage overhead

Value-type instances occupy precisely the memory required to store their fields. In this example, `Point` takes eight bytes of memory:

```
struct Point
{
  int x;  // 4 bytes
  int y;  // 4 bytes
}
```

Technically, the CLR positions fields within the type at an address that's a multiple of the fields' size (up to a maximum of eight bytes). Thus, the following actually consumes 16 bytes of memory (with the seven bytes following the first field "wasted"):

```
struct A { byte b; long l; }
```

You can override this behavior with the `StructLayout` attribute (see "Mapping a Struct to Unmanaged Memory" on page 1011 in Chapter 25).

Reference types require separate allocations of memory for the reference and object. The object consumes as many bytes as its fields, plus additional administrative overhead. The precise overhead is intrinsically private to the implementation of the .NET runtime, but at minimum, the overhead is eight bytes, used to store a key to the object's type, as well as temporary information such as its lock state for multithreading and a flag to indicate whether it has been fixed from movement by the garbage collector. Each reference to an object requires an extra four or eight bytes, depending on whether the .NET runtime is running on a 32- or 64-bit platform.

Predefined Type Taxonomy

The predefined types in C# are:

Value types

- Numeric
  —Signed integer (`sbyte`, `short`, `int`, `long`)
  —Unsigned integer (`byte`, `ushort`, `uint`, `ulong`)
  —Real number (`float`, `double`, `decimal`)
- Logical (`bool`)
- Character (`char`)

Reference types

- String (`string`)
- Object (`object`)

Predefined types in C# alias Framework types in the `System` namespace. There is only a syntactic difference between these two statements:

```
int i = 5;
System.Int32 i = 5;
```

The set of predefined *value* types, excluding `decimal`, are known as *primitive types* in the CLR. Primitive types are so called because they are supported directly via instructions in compiled code, and this usually translates to direct support on the underlying processor. For example:

```
                   // Underlying hexadecimal representation
int i = 7;         // 0x7
bool b = true;     // 0x1
char c = 'A';      // 0x41
float f = 0.5f;    // uses IEEE floating-point encoding
```

The `System.IntPtr` and `System.UIntPtr` types are also primitive (see Chapter 25).

Numeric Types

C# has the predefined numeric types shown in Table 2-1.

Table 2-1. Predefined numeric types in C#

C# type	System type	Suffix	Size	Range
Integral—signed				
`sbyte`	`SByte`		8 bits	-2^7 to 2^7-1
`short`	`Int16`		16 bits	-2^{15} to $2^{15}-1$
`int`	`Int32`		32 bits	-2^{31} to $2^{31}-1$
`long`	`Int64`	`L`	64 bits	-2^{63} to $2^{63}-1$
Integral—unsigned				
`byte`	`Byte`		8 bits	0 to 2^8-1
`ushort`	`UInt16`		16 bits	0 to $2^{16}-1$
`uint`	`UInt32`	`U`	32 bits	0 to $2^{32}-1$
`ulong`	`UInt64`	`UL`	64 bits	0 to $2^{64}-1$
Real				
`float`	`Single`	`F`	32 bits	$\pm$ (~10^{-45} to 10^{38})
`double`	`Double`	`D`	64 bits	$\pm$ (~10^{-324} to 10^{308})
`decimal`	`Decimal`	`M`	128 bits	$\pm$ (~10^{-28} to 10^{28})

Of the *integral* types, `int` and `long` are first-class citizens and are favored by both C# and the runtime. The other integral types are typically used for interoperability or when space efficiency is paramount.

Of the *real* number types, `float` and `double` are called *floating-point types*[2] and are typically used for scientific and graphical calculations. The `decimal` type is typically used for financial calculations, where base-10-accurate arithmetic and high precision are required.

Numeric Literals

Integral literals can use decimal or hexadecimal notation; hexadecimal is denoted with the `0x` prefix. For example:

```
int x = 127;
long y = 0x7F;
```

Real literals can use decimal and/or exponential notation. For example:

2 Technically, `decimal` is a floating-point type too, although it's not referred to as such in the C# language specification.

```
double d = 1.5;
double million = 1E06;
```

Numeric literal type inference

By default, the compiler *infers* a numeric literal to be either `double` or an integral type:

- If the literal contains a decimal point or the exponential symbol (`E`), it is a `double`.
- Otherwise, the literal's type is the first type in this list that can fit the literal's value: `int`, `uint`, `long`, and `ulong`.

For example:

```
Console.WriteLine (        1.0.GetType());  // Double  (double)
Console.WriteLine (       1E06.GetType());  // Double  (double)
Console.WriteLine (          1.GetType());  // Int32   (int)
Console.WriteLine ( 0xF0000000.GetType());  // UInt32  (uint)
Console.WriteLine (0x100000000.GetType());  // Int64   (long)
```

Numeric suffixes

Numeric suffixes explicitly define the type of a literal. Suffixes can be either lower- or uppercase, and are as follows:

Category	C# type	Example
F	float	float f = 1.0F;
D	double	double d = 1D;
M	decimal	decimal d = 1.0M;
U	uint	uint i = 1U;
L	long	long i = 1L;
UL	ulong	ulong i = 1UL;

The suffixes `U` and `L` are rarely necessary, because the `uint`, `long`, and `ulong` types can nearly always be either *inferred* or *implicitly converted* from `int`:

```
long i = 5;     // Implicit lossless conversion from int literal to long
```

The `D` suffix is technically redundant, in that all literals with a decimal point are inferred to be `double`. And you can always add a decimal point to a numeric literal:

```
double x = 4.0;
```

The `F` and `M` suffixes are the most useful and should always be applied when specifying `float` or `decimal` literals. Without the `F` suffix, the following line would not compile, because 4.5 would be inferred to be of type `double`, which has no implicit conversion to `float`:

```
float f = 4.5F;
```

The same principle is true for a decimal literal:

```
decimal d = -1.23M;     // Will not compile without the M suffix.
```

We describe the semantics of numeric conversions in detail in the following section.

Numeric Conversions

Integral to integral conversions

Integral conversions are *implicit* when the destination type can represent every possible value of the source type. Otherwise, an *explicit* conversion is required. For example:

```
int x = 12345;       // int is a 32-bit integral
long y = x;          // Implicit conversion to 64-bit integral
short z = (short)x;  // Explicit conversion to 16-bit integral
```

Floating-point to floating-point conversions

A `float` can be implicitly converted to a `double`, since a `double` can represent every possible value of a `float`. The reverse conversion must be explicit.

Floating-point to integral conversions

All integral types may be implicitly converted to all floating-point types:

```
int i = 1;
float f = i;
```

The reverse conversion must be explicit:

```
int i2 = (int)f;
```

When you cast from a floating-point number to an integral, any fractional portion is truncated; no rounding is performed. The static class `System.Convert` provides methods that round while converting between various numeric types (see Chapter 6).

Implicitly converting a large integral type to a floating-point type preserves *magnitude* but may occasionally lose *precision*. This is because floating-point types always have more magnitude than integral types, but may have less precision. Rewriting our example with a larger number demonstrates this:

```
int i1 = 100000001;
float f = i1;          // Magnitude preserved, precision lost
int i2 = (int)f;       // 100000000
```

Decimal conversions

All integral types can be implicitly converted to the decimal type, since a decimal can represent every possible C# integral value. All other numeric conversions to and from a decimal type must be explicit.

Arithmetic Operators

The arithmetic operators (+, -, *, /, %) are defined for all numeric types except the 8- and 16-bit integral types:

```
+    Addition
-    Subtraction
*    Multiplication
/    Division
%    Remainder after division
```

Increment and Decrement Operators

The increment and decrement operators (++, --) increment and decrement numeric types by 1. The operator can either follow or precede the variable, depending on whether you want its value *before* or *after* the increment/decrement. For example:

```
int x = 0, y = 0;
Console.WriteLine (x++);   // Outputs 0; x is now 1
Console.WriteLine (++y);   // Outputs 1; y is now 1
```

Specialized Integral Operations

Integral division

Division operations on integral types always truncate remainders (round toward zero). Dividing by a variable whose value is zero generates a runtime error (a `DivideByZeroException`):

```
int a = 2 / 3;      // 0

int b = 0;
int c = 5 / b;      // throws DivideByZeroException
```

Dividing by the *literal* or *constant* 0 generates a compile-time error.

Integral overflow

At runtime, arithmetic operations on integral types can overflow. By default, this happens silently—no exception is thrown, and the result exhibits "wraparound" behavior, as though the computation was done on a larger integer type and the extra significant bits discarded. For example, decrementing the minimum possible `int` value results in the maximum possible `int` value:

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue); // True
```

Integral arithmetic overflow check operators

The checked operator tells the runtime to generate an OverflowException rather than overflowing silently when an integral expression or statement exceeds the arithmetic limits of that type. The checked operator affects expressions with the ++, −−, +, − (binary and unary), *, /, and explicit conversion operators between integral types.

The checked operator has no effect on the double and float types (which overflow to special "infinite" values, as we'll see soon) and no effect on the decimal type (which is always checked).

checked can be used around either an expression or a statement block. For example:

```
int a = 1000000;
int b = 1000000;

int c = checked (a * b);      // Checks just the expression.

checked                       // Checks all expressions
{                             // in statement block.
   ...
   c = a * b;
   ...
}
```

You can make arithmetic overflow checking the default for all expressions in a program by compiling with the /checked+ command-line switch (in Visual Studio, go to Advanced Build Settings). If you then need to disable overflow checking just for specific expressions or statements, you can do so with the unchecked operator. For example, the following code will not throw exceptions—even if compiled with /checked+:

```
int x = int.MaxValue;
int y = unchecked (x + 1);
unchecked { int z = x + 1; }
```

Overflow checking for constant expressions

Regardless of the /checked compiler switch, expressions evaluated at compile time are always overflow-checked—unless you apply the unchecked operator:

```
int x = int.MaxValue + 1;               // Compile-time error
int y = unchecked (int.MaxValue + 1);   // No errors
```

Bitwise operators

C# supports the following bitwise operators:

Operator	Meaning	Sample expression	Result
~	Complement	~0xfU	0xfffffff0U
&	And	0xf0 & 0x33	0x30
|	Or	0xf0 | 0x33	0xf3
^	Exclusive Or	0xff00 ^ 0x0ff0	0xf0f0
<<	Shift left	0x20 << 2	0x80
>>	Shift right	0x20 >> 1	0x10

8- and 16-Bit Integrals

The 8- and 16-bit integral types are `byte`, `sbyte`, `short`, and `ushort`. These types lack their own arithmetic operators, so C# implicitly converts them to larger types as required. This can cause a compile-time error when trying to assign the result back to a small integral type:

```
short x = 1, y = 1;
short z = x + y;          // Compile-time error
```

In this case, `x` and `y` are implicitly converted to `int` so that the addition can be performed. This means the result is also an `int`, which cannot be implicitly cast back to a `short` (because it could cause loss of data). To make this compile, we must add an explicit cast:

```
short z = (short) (x + y);   // OK
```

Special Float and Double Values

Unlike integral types, floating-point types have values that certain operations treat specially. These special values are NaN (not a number), +∞, −∞, and −0. The `float` and `double` classes have constants for `NaN`, +∞, and −∞, as well as other values (`MaxValue`, `MinValue`, and `Epsilon`). For example:

```
Console.WriteLine (double.NegativeInfinity);   // -Infinity
```

The constants that represent special values for `double` and `float` are as follows:

Special value	Double constant	Float constant
NaN	double.NaN	float.NaN
+∞	double.PositiveInfinity	float.PositiveInfinity
−∞	double.NegativeInfinity	float.NegativeInfinity
−0	-0.0	-0.0f

Dividing a nonzero number by zero results in an infinite value. For example:

```
Console.WriteLine ( 1.0 /  0.0);                  //  Infinity
Console.WriteLine (-1.0 /  0.0);                  // -Infinity
```

```
Console.WriteLine ( 1.0 / -0.0);                    // -Infinity
Console.WriteLine (-1.0 / -0.0);                    //  Infinity
```

Dividing zero by zero, or subtracting infinity from infinity, results in a NaN. For example:

```
Console.WriteLine ( 0.0 /  0.0);                    //  NaN
Console.WriteLine ((1.0 /  0.0) - (1.0 / 0.0));    //  NaN
```

When using ==, a NaN value is never equal to another value, even another NaN value:

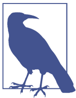
```
Console.WriteLine (0.0 / 0.0 == double.NaN);    // False
```

To test whether a value is NaN, you must use the `float.IsNaN` or `double.IsNaN` method:

```
Console.WriteLine (double.IsNaN (0.0 / 0.0));   // True
```

When using `object.Equals`, however, two NaN values are equal:

```
Console.WriteLine (object.Equals (0.0 / 0.0, double.NaN));   // True
```



NaNs are sometimes useful in representing special values. In WPF, `double.NaN` represents a measurement whose value is "Automatic". Another way to represent such a value is with a nullable type (Chapter 4); another is with a custom struct that wraps a numeric type and adds an additional field (Chapter 3).

`float` and `double` follow the specification of the IEEE 754 format types, supported natively by almost all processors. You can find detailed information on the behavior of these types at *http://www.ieee.org*.

double Versus decimal

`double` is useful for scientific computations (such as computing spatial coordinates). `decimal` is useful for financial computations and values that are "man-made" rather than the result of real-world measurements. Here's a summary of the differences:

Category	`double`	`decimal`
Internal representation	Base 2	Base 10
Decimal precision	15–16 significant figures	28–29 significant figures
Range	$\pm(\sim10^{-324}$ to $\sim10^{308})$	$\pm(\sim10^{-28}$ to $\sim10^{28})$
Special values	+0, −0, +∞, −∞, and NaN	None
Speed	Native to processor	Non-native to processor (about 10 times slower than `double`)

Real-Number Rounding Errors

`float` and `double` internally represent numbers in base 2. For this reason, only numbers expressible in base 2 are represented precisely. Practically, this means most

literals with a fractional component (which are in base 10) will not be represented precisely. For example:

```
float tenth = 0.1f;                       // Not quite 0.1
float one   = 1f;
Console.WriteLine (one - tenth * 10f);    // -1.490116E-08
```

This is why `float` and `double` are bad for financial calculations. In contrast, `decimal` works in base 10 and so can precisely represent numbers expressible in base 10 (as well as its factors, base 2 and base 5). Since real literals are in base 10, `decimal` can precisely represent numbers such as 0.1. However, neither `double` nor `decimal` can precisely represent a fractional number whose base 10 representation is recurring:

```
decimal m = 1M / 6M;               // 0.1666666666666666666666666667M
double  d = 1.0 / 6.0;             // 0.16666666666666666
```

This leads to accumulated rounding errors:

```
decimal notQuiteWholeM = m+m+m+m+m+m;  // 1.0000000000000000000000000002M
double  notQuiteWholeD = d+d+d+d+d+d;  // 0.99999999999999989
```

which breaks equality and comparison operations:

```
Console.WriteLine (notQuiteWholeM == 1M);   // False
Console.WriteLine (notQuiteWholeD < 1.0);   // True
```


Boolean Type and Operators

C#'s `bool` type (aliasing the `System.Boolean` type) is a logical value that can be assigned the literal `true` or `false`.

Although a Boolean value requires only one bit of storage, the runtime will use one byte of memory, since this is the minimum chunk that the runtime and processor can efficiently work with. To avoid space inefficiency in the case of arrays, the Framework provides a `BitArray` class in the `System.Collections` namespace that is designed to use just one bit per Boolean value.

Bool Conversions

No casting conversions can be made from the `bool` type to numeric types or vice versa.

Equality and Comparison Operators

`==` and `!=` test for equality and inequality of any type, but always return a `bool` value.[3] Value types typically have a very simple notion of equality:

3 It's possible to *overload* these operators (Chapter 4) such that they return a non-`bool` type, but this is almost never done in practice.

```
int x = 1;
int y = 2;
int z = 1;
Console.WriteLine (x == y);         // False
Console.WriteLine (x == z);         // True
```

For reference types, equality, by default, is based on *reference*, as opposed to the actual *value* of the underlying object (more on this in Chapter 6):

```
public class Dude
{
  public string Name;
  public Dude (string n) { Name = n; }
}
...
Dude d1 = new Dude ("John");
Dude d2 = new Dude ("John");
Console.WriteLine (d1 == d2);       // False
Dude d3 = d1;
Console.WriteLine (d1 == d3);       // True
```

The equality and comparison operators, `==`, `!=`, `<`, `>`, `>=`, and `<=`, work for all numeric types, but should be used with caution with real numbers (as we saw in "Real-Number Rounding Errors" on page 32). The comparison operators also work on `enum` type members, by comparing their underlying integral values. We describe this in "Enums" on page 109 in Chapter 3.

We explain the equality and comparison operators in greater detail in "Operator Overloading" on page 168 in Chapter 4, and in "Equality Comparison" on page 267 and "Order Comparison" on page 278 in Chapter 6.

Conditional Operators

The `&&` and `||` operators test for *and* and *or* conditions. They are frequently used in conjunction with the `!` operator, which expresses *not*. In this example, the `UseUmbrella` method returns `true` if it's rainy or sunny (to protect us from the rain or the sun), as long as it's not also windy (since umbrellas are useless in the wind):

```
static bool UseUmbrella (bool rainy, bool sunny, bool windy)
{
  return !windy && (rainy || sunny);
}
```

The `&&` and `||` operators *short-circuit* evaluation when possible. In the preceding example, if it is windy, the expression `(rainy || sunny)` is not even evaluated. Short-circuiting is essential in allowing expressions such as the following to run without throwing a `NullReferenceException`:

```
if (sb != null && sb.Length > 0) ...
```

The `&` and `|` operators also test for *and* and *or* conditions:

```
return !windy & (rainy | sunny);
```

The difference is that they *do not short-circuit*. For this reason, they are rarely used in place of conditional operators.

Unlike in C and C++, the `&` and `|` operators perform (non-short-circuiting) *Boolean* comparisons when applied to `bool` expressions. The `&` and `|` operators perform *bitwise* operations only when applied to numbers.

Conditional operator (ternary operator)

The *conditional operator* (more commonly called the *ternary operator*, as it's the only operator that takes three operands) has the form `q ? a : b`, where if condition `q` is true, `a` is evaluated, else `b` is evaluated. For example:

```
static int Max (int a, int b)
{
  return (a > b) ? a : b;
}
```

The conditional operator is particularly useful in LINQ queries (Chapter 8).

Strings and Characters

C#'s `char` type (aliasing the `System.Char` type) represents a Unicode character and occupies 2 bytes. A `char` literal is specified inside single quotes:

```
char c = 'A';       // Simple character
```

Escape sequences express characters that cannot be expressed or interpreted literally. An escape sequence is a backslash followed by a character with a special meaning. For example:

```
char newLine = '\n';
char backSlash = '\\';
```

The escape sequence characters are shown in Table 2-2.

Table 2-2. Escape sequence characters

Char	Meaning	Value
`\'`	Single quote	`0x0027`
`\"`	Double quote	`0x0022`
`\\`	Backslash	`0x005C`
`\0`	Null	`0x0000`
`\a`	Alert	`0x0007`
`\b`	Backspace	`0x0008`
`\f`	Form feed	`0x000C`

Char	Meaning	Value
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

The \u (or \x) escape sequence lets you specify any Unicode character via its four-digit hexadecimal code:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

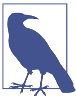
Char Conversions

An implicit conversion from a char to a numeric type works for the numeric types that can accommodate an unsigned short. For other numeric types, an explicit conversion is required.

String Type

C#'s string type (aliasing the System.String type, covered in depth in Chapter 6) represents an immutable sequence of Unicode characters. A string literal is specified inside double quotes:

```
string a = "Heat";
```



string is a reference type, rather than a value type. Its equality operators, however, follow value-type semantics:

```
string a = "test";
string b = "test";
Console.Write (a == b);  // True
```

The escape sequences that are valid for char literals also work inside strings:

```
string a = "Here's a tab:\t";
```

The cost of this is that whenever you need a literal backslash, you must write it twice:

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

To avoid this problem, C# allows *verbatim* string literals. A verbatim string literal is prefixed with @ and does not support escape sequences. The following verbatim string is identical to the preceding one:

```
string a2 = @ "\\server\fileshare\helloworld.cs";
```

A verbatim string literal can also span multiple lines:

```
string escaped  = "First Line\r\nSecond Line";
string verbatim = @"First Line
```

```
Second Line";

// True if your IDE uses CR-LF line separators:
Console.WriteLine (escaped == verbatim);
```

You can include the double-quote character in a verbatim literal by writing it twice:

```
string xml = @"<customer id=""123""></customer>";
```


String concatenation

The + operator concatenates two strings:

```
string s = "a" + "b";
```

One of the operands may be a nonstring value, in which case `ToString` is called on that value. For example:

```
string s = "a" + 5;  // a5
```

Using the + operator repeatedly to build up a string is inefficient: a better solution is to use the `System.Text.StringBuilder` type (described in Chapter 6).

String interpolation (C# 6)

A string preceded with the `$` character is called an *interpolated string*. Interpolated strings can include expressions inside braces:

```
int x = 4;
Console.Write ($"A square has {x} sides");  // Prints: A square has 4 sides
```

Any valid C# expression of any type can appear within the braces, and C# will convert the expression to a string by calling its `ToString` method or equivalent. You can change the formatting by appending the expression with a colon and a *format string* (format strings are described in "Formatting and parsing" on page 233 in Chapter 6):

```
string s = $"255 in hex is {byte.MaxValue:X2}";  // X2 = 2-digit Hexadecimal
// Evaluates to "255 in hex is FF"
```

Interpolated strings must complete on a single line, unless you also specify the verbatim string operator. Note that the `$` operator must come before `@`:

```
int x = 2;
string s = $@"this spans {
x} lines";
```

To include a brace literal in an interpolated string, repeat the desired brace character.

String comparisons

`string` does not support `<` and `>` operators for comparisons. You must use the string's `CompareTo` method, described in Chapter 6.

Arrays

An array represents a fixed number of variables (called *elements*) of a particular type. The elements in an array are always stored in a contiguous block of memory, providing highly efficient access.

An array is denoted with square brackets after the element type. For example:

```
char[] vowels = new char[5];    // Declare an array of 5 characters
```

Square brackets also *index* the array, accessing a particular element by position:

```
vowels[0] = 'a';
vowels[1] = 'e';
vowels[2] = 'i';
vowels[3] = 'o';
vowels[4] = 'u';
Console.WriteLine (vowels[1]);      // e
```

This prints "e" because array indexes start at 0. We can use a `for` loop statement to iterate through each element in the array. The `for` loop in this example cycles the integer `i` from `0` to `4`:

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write (vowels[i]);            // aeiou
```

The `Length` property of an array returns the number of elements in the array. Once an array has been created, its length cannot be changed. The `System.Collection` namespace and subnamespaces provide higher-level data structures, such as dynamically sized arrays and dictionaries.

An *array initialization expression* lets you declare and populate an array in a single step:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

or simply:

```
char[] vowels = {'a','e','i','o','u'};
```

All arrays inherit from the `System.Array` class, providing common services for all arrays. These members include methods to get and set elements regardless of the array type, and are described in "The Array Class" on page 297 in Chapter 7.

Default Element Initialization

Creating an array always preinitializes the elements with default values. The default value for a type is the result of a bitwise zeroing of memory. For example, consider creating an array of integers. Since `int` is a value type, this allocates 1,000 integers in one contiguous block of memory. The default value for each element will be 0:

```
int[] a = new int[1000];
Console.Write (a[123]);            // 0
```

Value types versus reference types

Whether an array element type is a value type or a reference type has important performance implications. When the element type is a value type, each element value is allocated as part of the array. For example:

```
public struct Point { public int X, Y; }
...

Point[] a = new Point[1000];
int x = a[500].X;                   // 0
```

Had `Point` been a class, creating the array would have merely allocated 1,000 null references:

```
public class Point { public int X, Y; }

...
Point[] a = new Point[1000];
int x = a[500].X;                   // Runtime error, NullReferenceException
```

To avoid this error, we must explicitly instantiate 1,000 `Points` after instantiating the array:

```
Point[] a = new Point[1000];
for (int i = 0; i < a.Length; i++) // Iterate i from 0 to 999
   a[i] = new Point();             // Set array element i with new point
```

An array *itself* is always a reference type object, regardless of the element type. For instance, the following is legal:

```
int[] a = null;
```

Multidimensional Arrays

Multidimensional arrays come in two varieties: *rectangular* and *jagged*. Rectangular arrays represent an *n*-dimensional block of memory, and jagged arrays are arrays of arrays.

Rectangular arrays

Rectangular arrays are declared using commas to separate each dimension. The following declares a rectangular two-dimensional array, where the dimensions are 3 x 3:

```
int[,] matrix = new int[3,3];
```

The `GetLength` method of an array returns the length for a given dimension (starting at 0):

```
for (int i = 0; i < matrix.GetLength(0); i++)
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix[i,j] = i * 3 + j;
```

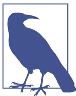
A rectangular array can be initialized as follows (to create an array identical to the previous example):

```
int[,] matrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

Jagged arrays

Jagged arrays are declared using successive square brackets to represent each dimension. Here is an example of declaring a jagged two-dimensional array, where the outermost dimension is 3:

```
int[][] matrix = new int[3][];
```



Interestingly, this is `new int[3][]` and not `new int[][3]`. Eric Lippert has written an excellent article on why this is so: see *http://albahari.com/jagged*.

The inner dimensions aren't specified in the declaration because, unlike a rectangular array, each inner array can be an arbitrary length. Each inner array is implicitly initialized to null rather than an empty array. Each inner array must be created manually:

```
for (int i = 0; i < matrix.Length; i++)
{
  matrix[i] = new int[3];                    // Create inner array
  for (int j = 0; j < matrix[i].Length; j++)
    matrix[i][j] = i * 3 + j;
}
```

A jagged array can be initialized as follows (to create an array identical to the previous example with an additional element at the end):

```
int[][] matrix = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8,9}
};
```

Simplified Array Initialization Expressions

There are two ways to shorten array initialization expressions. The first is to omit the `new` operator and type qualifications:

```
char[] vowels = {'a','e','i','o','u'};

int[,] rectangularMatrix =
{
  {0,1,2},
  {3,4,5},
```

```
  {6,7,8}
};

int[][] jaggedMatrix =
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```


The second approach is to use the `var` keyword, which tells the compiler to implicitly type a local variable:

```
var i = 3;           // i is implicitly of type int
var s = "sausage";   // s is implicitly of type string

// Therefore:

var rectMatrix = new int[,]    // rectMatrix is implicitly of type int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};

var jaggedMat = new int[][]    // jaggedMat is implicitly of type int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

Implicit typing can be taken one stage further with arrays: you can omit the type qualifier after the `new` keyword and have the compiler *infer* the array type:

```
var vowels = new[] {'a','e','i','o','u'};   // Compiler infers char[]
```

For this to work, the elements must all be implicitly convertible to a single type (and at least one of the elements must be of that type, and there must be exactly one best type). For example:

```
var x = new[] {1,10000000000};   // all convertible to long
```

Bounds Checking

All array indexing is bounds-checked by the runtime. An `IndexOutOfRangeException` is thrown if you use an invalid index:

```
int[] arr = new int[3];
arr[3] = 1;               // IndexOutOfRangeException thrown
```

As with Java, array bounds checking is necessary for type safety and simplifies debugging.

Generally, the performance hit from bounds checking is minor, and the JIT (just-in-time) compiler can perform optimizations, such as determining in advance whether all indexes will be safe before entering a loop, thus avoiding a check on each iteration. In addition, C# provides "unsafe" code that can explicitly bypass bounds checking (see "Unsafe Code and Pointers" on page 187 in Chapter 4).

Variables and Parameters

A variable represents a storage location that has a modifiable value. A variable can be a *local variable*, *parameter* (*value*, *ref*, or *out*), *field* (*instance* or *static*), or *array element*.

The Stack and the Heap

The stack and the heap are the places where variables and constants reside. Each has very different lifetime semantics.

Stack

The stack is a block of memory for storing local variables and parameters. The stack logically grows and shrinks as a function is entered and exited. Consider the following method (to avoid distraction, input argument checking is ignored):

```
static int Factorial (int x)
{
  if (x == 0) return 1;
  return x * Factorial (x-1);
}
```

This method is recursive, meaning that it calls itself. Each time the method is entered, a new `int` is allocated on the stack, and each time the method exits, the `int` is deallocated.

Heap

The heap is a block of memory in which *objects* (i.e., reference-type instances) reside. Whenever a new object is created, it is allocated on the heap, and a reference to that object is returned. During a program's execution, the heap starts filling up as new objects are created. The runtime has a garbage collector that periodically deallocates objects from the heap, so your program does not run out of memory. An object is eligible for deallocation as soon as it's not referenced by anything that's itself "alive."

In the following example, we start by creating a `StringBuilder` object referenced by the variable `ref1`, and then write out its content. That `StringBuilder` object is then immediately eligible for garbage collection, because nothing subsequently uses it.

Then, we create another StringBuilder referenced by variable ref2, and copy that reference to ref3. Even though ref2 is not used after that point, ref3 keeps the same StringBuilder object alive—ensuring that it doesn't become eligible for collection until we've finished using ref3:

```
using System;
using System.Text;

class Test
{
  static void Main()
  {
    StringBuilder ref1 = new StringBuilder ("object1");
    Console.WriteLine (ref1);
    // The StringBuilder referenced by ref1 is now eligible for GC.

    StringBuilder ref2 = new StringBuilder ("object2");
    StringBuilder ref3 = ref2;
    // The StringBuilder referenced by ref2 is NOT yet eligible for GC.

    Console.WriteLine (ref3);                   // object2
  }
}
```


Value-type instances (and object references) live wherever the variable was declared. If the instance was declared as a field within a class type, or as an array element, that instance lives on the heap.

You can't explicitly delete objects in C#, as you can in C++. An unreferenced object is eventually collected by the garbage collector.

The heap also stores static fields. Unlike objects allocated on the heap (which can get garbage-collected), these live until the application domain is torn down.

Definite Assignment

C# enforces a definite assignment policy. In practice, this means that outside of an unsafe context, it's impossible to access uninitialized memory. Definite assignment has three implications:

- Local variables must be assigned a value before they can be read.
- Function arguments must be supplied when a method is called (unless marked as optional—see "Optional parameters" on page 48).
- All other variables (such as fields and array elements) are automatically initialized by the runtime.

For example, the following code results in a compile-time error:

```
static void Main()
{
  int x;
  Console.WriteLine (x);        // Compile-time error
}
```

Fields and array elements are automatically initialized with the default values for their type. The following code outputs 0, because array elements are implicitly assigned to their default values:

```
static void Main()
{
  int[] ints = new int[2];
  Console.WriteLine (ints[0]);    // 0
}
```

The following code outputs 0, because fields are implicitly assigned a default value:

```
class Test
{
  static int x;
  static void Main() { Console.WriteLine (x); }   // 0
}
```

Default Values

All type instances have a default value. The default value for the predefined types is the result of a bitwise zeroing of memory:

Type	Default value
All reference types	null
All numeric and enum types	0
char type	'\0'
bool type	false

You can obtain the default value for any type with the `default` keyword (in practice, this is useful with generics which we'll cover in Chapter 3):

```
decimal d = default (decimal);
```

The default value in a custom value type (i.e., `struct`) is the same as the default value for each field defined by the custom type.

Parameters

A method has a sequence of parameters. Parameters define the set of arguments that must be provided for that method. In this example, the method `Foo` has a single parameter named `p`, of type `int`:

```
static void Foo (int p)
{
  p = p + 1;                 // Increment p by 1
```

```
    Console.WriteLine (p);     // Write p to screen
  }

  static void Main()
  {
    Foo (8);                  // Call Foo with an argument of 8
  }
```

You can control how parameters are passed with the `ref` and `out` modifiers:

Parameter modifier	Passed by	Variable must be definitely assigned
(None)	Value	Going *in*
`ref`	Reference	Going *in*
`out`	Reference	Going *out*

Passing arguments by value

By default, arguments in C# are *passed by value*, which is by far the most common case. This means a copy of the value is created when passed to the method:

```
class Test
{
  static void Foo (int p)
  {
    p = p + 1;                // Increment p by 1
    Console.WriteLine (p);    // Write p to screen
  }

  static void Main()
  {
    int x = 8;
    Foo (x);                  // Make a copy of x
    Console.WriteLine (x);    // x will still be 8
  }
}
```

Assigning `p` a new value does not change the contents of `x`, since `p` and `x` reside in different memory locations.

Passing a reference-type argument by value copies the *reference*, but not the object. In the following example, `Foo` sees the same `StringBuilder` object that `Main` instantiated, but has an independent *reference* to it. In other words, `sb` and `fooSB` are separate variables that reference the same `StringBuilder` object:

```
class Test
{
  static void Foo (StringBuilder fooSB)
  {
    fooSB.Append ("test");
    fooSB = null;
  }
```

```
  static void Main()
  {
    StringBuilder sb = new StringBuilder();
    Foo (sb);
    Console.WriteLine (sb.ToString());    // test
  }
}
```

Because fooSB is a *copy* of a reference, setting it to null doesn't make sb null. (If, however, fooSB was declared and called with the ref modifier, sb *would* become null.)

The ref modifier

To *pass by reference*, C# provides the ref parameter modifier. In the following example, p and x refer to the same memory locations:

```
class Test
{
  static void Foo (ref int p)
  {
    p = p + 1;               // Increment p by 1
    Console.WriteLine (p);   // Write p to screen
  }

  static void Main()
  {
    int x = 8;
    Foo (ref  x);            // Ask Foo to deal directly with x
    Console.WriteLine (x);   // x is now 9
  }
}
```

Now assigning p a new value changes the contents of x. Notice how the ref modifier is required both when writing and when calling the method.[4] This makes it very clear what's going on.

The ref modifier is essential in implementing a swap method (later, in "Generics" on page 114 in Chapter 3, we will show how to write a swap method that works with any type):

```
class Test
{
  static void Swap (ref string a, ref string b)
  {
    string temp = a;
    a = b;
    b = temp;
  }
```

4 An exception to this rule is when calling COM methods. We discuss this in Chapter 25.

```
  static void Main()
  {
    string x = "Penn";
    string y = "Teller";
    Swap (ref x, ref y);
    Console.WriteLine (x);   // Teller
    Console.WriteLine (y);   // Penn
  }
}
```

A parameter can be passed by reference or by value, regardless of whether the parameter type is a reference type or a value type.

The out modifier

An `out` argument is like a `ref` argument, except it:

- Need not be assigned before going into the function
- Must be assigned before it comes *out* of the function

The `out` modifier is most commonly used to get multiple return values back from a method. For example:

```
class Test
{
  static void Split (string name, out string firstNames,
                     out string lastName)
  {
     int i = name.LastIndexOf (' ');
     firstNames = name.Substring (0, i);
     lastName   = name.Substring (i + 1);
  }

  static void Main()
  {
    string a, b;
    Split ("Stevie Ray Vaughan", out a, out b);
    Console.WriteLine (a);                      // Stevie Ray
    Console.WriteLine (b);                      // Vaughan
  }
}
```

Like a `ref` parameter, an `out` parameter is passed by reference.

Implications of passing by reference

When you pass an argument by reference, you alias the storage location of an existing variable rather than create a new storage location. In the following example, the variables `x` and `y` represent the same instance:

```
class Test
{
```

```
  static int x;

  static void Main() { Foo (out x); }

  static void Foo (out int y)
  {
    Console.WriteLine (x);                // x is 0
    y = 1;                                // Mutate y
    Console.WriteLine (x);                // x is 1
  }
}
```

The params modifier

The `params` parameter modifier may be specified on the last parameter of a method so that the method accepts any number of arguments of a particular type. The parameter type must be declared as an array. For example:

```
class Test
{
  static int Sum (params int[] ints)
  {
    int sum = 0;
    for (int i = 0; i < ints.Length; i++)
      sum += ints[i];                       // Increase sum by ints[i]
    return sum;
  }

  static void Main()
  {
    int total = Sum (1, 2, 3, 4);
    Console.WriteLine (total);              // 10
  }
}
```

You can also supply a `params` argument as an ordinary array. The first line in `Main` is semantically equivalent to this:

```
int total = Sum (new int[] { 1, 2, 3, 4 } );
```

Optional parameters

From C# 4.0, methods, constructors, and indexers (Chapter 3) can declare *optional parameters*. A parameter is optional if it specifies a *default value* in its declaration:

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

Optional parameters may be omitted when calling the method:

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

The *default argument* of 23 is actually *passed* to the optional parameter `x`—the compiler bakes the value 23 into the compiled code at the *calling* side. The preceding call to `Foo` is semantically identical to:

```
Foo (23);
```

because the compiler simply substitutes the default value of an optional parameter wherever it is used.

Adding an optional parameter to a public method that's called from another assembly requires recompilation of both assemblies—just as though the parameter were mandatory.

The default value of an optional parameter must be specified by a constant expression, or a parameterless constructor of a value type. Optional parameters cannot be marked with `ref` or `out`.

Mandatory parameters must occur *before* optional parameters in both the method declaration and the method call (the exception is with `params` arguments, which still always come last). In the following example, the explicit value of `1` is passed to `x`, and the default value of `0` is passed to `y`:

```
void Foo (int x = 0, int y = 0) { Console.WriteLine (x + ", " + y); }

void Test()
{
  Foo(1);    // 1, 0
}
```

To do the converse (pass a default value to `x` and an explicit value to `y`), you must combine optional parameters with *named arguments*.

Named arguments

Rather than identifying an argument by position, you can identify an argument by name. For example:

```
void Foo (int x, int y) { Console.WriteLine (x + ", " + y); }

void Test()
{
  Foo (x:1, y:2);  // 1, 2
}
```

Named arguments can occur in any order. The following calls to `Foo` are semantically identical:

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```

A subtle difference is that argument expressions are evaluated in the order in which they appear at the *calling* site. In general, this makes a difference only with interdependent side-effecting expressions such as the following, which writes `0, 1`:

```
int a = 0;
Foo (y: ++a, x: --a);  // ++a is evaluated first
```

Of course, you would almost certainly avoid writing such code in practice!

You can mix named and positional arguments:

```
Foo (1, y:2);
```

However, there is a restriction: positional arguments must come before named arguments. So we couldn't call `Foo` like this:

```
Foo (x:1, 2);         // Compile-time error
```

Named arguments are particularly useful in conjunction with optional parameters. For instance, consider the following method:

```
void Bar (int a = 0, int b = 0, int c = 0, int d = 0) { ... }
```

We can call this supplying only a value for `d` as follows:

```
Bar (d:3);
```

This is particularly useful when calling COM APIs, and is discussed in detail in Chapter 25.

var—Implicitly Typed Local Variables

It is often the case that you declare and initialize a variable in one step. If the compiler is able to infer the type from the initialization expression, you can use the keyword `var` (introduced in C# 3.0) in place of the type declaration. For example:

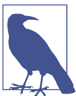
```
var x = "hello";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

This is precisely equivalent to:

```
string x = "hello";
System.Text.StringBuilder y = new System.Text.StringBuilder();
float z = (float)Math.PI;
```

Because of this direct equivalence, implicitly typed variables are statically typed. For example, the following generates a compile-time error:

```
var x = 5;
x = "hello";    // Compile-time error; x is of type int
```



`var` can decrease code readability in the case when *you can't deduce the type purely by looking at the variable declaration*. For example:

```
Random r = new Random();
var x = r.Next();
```

What type is `x`?

In "Anonymous Types" on page 174 in Chapter 4, we will describe a scenario where the use of `var` is mandatory.

Expressions and Operators

An *expression* essentially denotes a value. The simplest kinds of expressions are constants and variables. Expressions can be transformed and combined using operators. An *operator* takes one or more input *operands* to output a new expression.

Here is an example of a *constant expression*:

```
12
```

We can use the `*` operator to combine two operands (the literal expressions `12` and `30`), as follows:

```
12 * 30
```

Complex expressions can be built because an operand may itself be an expression, such as the operand `(12 * 30)` in the following example:

```
1 + (12 * 30)
```

Operators in C# can be classed as *unary*, *binary*, or *ternary*—depending on the number of operands they work on (one, two, or three). The binary operators always use *infix* notation, where the operator is placed *between* the two operands.

Primary Expressions

Primary expressions include expressions composed of operators that are intrinsic to the basic plumbing of the language. Here is an example:

```
Math.Log (1)
```

This expression is composed of two primary expressions. The first expression performs a member-lookup (with the `.` operator), and the second expression performs a method call (with the `()` operator).

Void Expressions

A void expression is an expression that has no value. For example:

```
Console.WriteLine (1)
```

A void expression, since it has no value, cannot be used as an operand to build more complex expressions:

```
1 + Console.WriteLine (1)      // Compile-time error
```

Assignment Expressions

An assignment expression uses the = operator to assign the result of another expression to a variable. For example:

```
x = x * 5
```

An assignment expression is not a void expression—it has a value of whatever was assigned, and so can be incorporated into another expression. In the following example, the expression assigns 2 to `x` and 10 to `y`:

```
y = 5 * (x = 2)
```

This style of expression can be used to initialize multiple values:

```
a = b = c = d = 0
```

The *compound assignment operators* are syntactic shortcuts that combine assignment with another operator. For example:

```
x *= 2    // equivalent to x = x * 2
x <<= 1   // equivalent to x = x << 1
```

(A subtle exception to this rule is with *events*, which we describe in Chapter 4: the `+=` and `-=` operators here are treated specially and map to the event's `add` and `remove` accessors.)

Operator Precedence and Associativity

When an expression contains multiple operators, *precedence* and *associativity* determine the order of their evaluation. Operators with higher precedence execute before operators of lower precedence. If the operators have the same precedence, the operator's associativity determines the order of evaluation.

Precedence

The following expression:

```
1 + 2 * 3
```

is evaluated as follows because `*` has a higher precedence than `+`:

```
1 + (2 * 3)
```

Left-associative operators

Binary operators (except for assignment, lambda, and null-coalescing operators) are *left-associative*; in other words, they are evaluated from left to right. For example, the following expression:

```
8 / 4 / 2
```

is evaluated as follows due to left associativity:

```
( 8 / 4 ) / 2    // 1
```

You can insert parentheses to change the actual order of evaluation:

```
8 / ( 4 / 2 )    // 4
```

Right-associative operators

The *assignment operators*, lambda, null-coalescing, and conditional operator are *right-associative*; in other words, they are evaluated from right to left. Right associativity allows multiple assignments such as the following to compile:

```
x = y = 3;
```

This first assigns 3 to y, and then assigns the result of that expression (3) to x.

Operator Table

Table 2-3 lists C#'s operators in order of precedence. Operators in the same category have the same precedence. We explain user-overloadable operators in "Operator Overloading" on page 168 in Chapter 4.

Table 2-3. C# operators (categories in order of precedence)

Category	Operator symbol	Operator name	Example	User-overloadable
Primary	.	Member access	x.y	No
	-> (unsafe)	Pointer to struct	x->y	No
	()	Function call	x()	No
	[]	Array/index	a[x]	Via indexer
	++	Post-increment	x++	Yes
	--	Post-decrement	x--	Yes
	new	Create instance	new Foo()	No
	stackalloc	Unsafe stack allocation	stackalloc(10)	No
	typeof	Get type from identifier	typeof(int)	No
	nameof	Get name of identifier	nameof(x)	No
	checked	Integral overflow check on	checked(x)	No
	unchecked	Integral overflow check off	unchecked(x)	No
	default	Default value	default(char)	No
Unary	await	Await	await myTask	No
	sizeof	Get size of struct	sizeof(int)	No
	?.	Null-conditional	x?.y	No
	+	Positive value of	+x	Yes
	-	Negative value of	-x	Yes
	!	Not	!x	Yes

Category	Operator symbol	Operator name	Example	User-overloadable
	~	Bitwise complement	~x	Yes
	++	Pre-increment	++x	Yes
	--	Pre-decrement	--x	Yes
	()	Cast	(int)x	No
	* (unsafe)	Value at address	*x	No
	& (unsafe)	Address of value	&x	No
Multiplicative	*	Multiply	x * y	Yes
	/	Divide	x / y	Yes
	%	Remainder	x % y	Yes
Additive	+	Add	x + y	Yes
	-	Subtract	x - y	Yes
Shift	<<	Shift left	x << 1	Yes
	>>	Shift right	x >> 1	Yes
Relational	<	Less than	x < y	Yes
	>	Greater than	x > y	Yes
	<=	Less than or equal to	x <= y	Yes
	>=	Greater than or equal to	x >= y	Yes
	is	Type is or is subclass of	x is y	No
	as	Type conversion	x as y	No
Equality	==	Equals	x == y	Yes
	!=	Not equals	x != y	Yes
Logical And	&	And	x & y	Yes
Logical Xor	^	Exclusive Or	x ^ y	Yes
Logical Or	|	Or	x | y	Yes
Conditional And	&&	Conditional And	x && y	Via &
Conditional Or	||	Conditional Or	x || y	Via |
Null-coalescing	??	Null-coalescing	x ?? y	No
Conditional	?:	Conditional	isTrue ? thenThisValue : elseThisValue	No
Assignment & Lambda	=	Assign	x = y	No
	*=	Multiply self by	x *= 2	Via *

Category	Operator symbol	Operator name	Example	User-overloadable
	/=	Divide self by	x /= 2	Via /
	+=	Add to self	x += 2	Via +
	-=	Subtract from self	x -= 2	Via -
	<<=	Shift self left by	x <<= 2	Via <<
	>>=	Shift self right by	x >>= 2	Via >>
	&=	And self by	x &= 2	Via &
	^=	Exclusive-Or self by	x ^= 2	Via ^
	|=	Or self by	x |= 2	Via |
	=>	Lambda	x => x + 1	No

Null Operators

C# provides two operators to make it easier to work with nulls: the *null-coalescing operator* and the *null-conditional operator*.

Null-Coalescing Operator

The ?? operator is the *null-coalescing operator*. It says "If the operand is non-null, give it to me; otherwise, give me a default value." For example:

```
string s1 = null;
string s2 = s1 ?? "nothing";   // s2 evaluates to "nothing"
```

If the left-hand expression is non-null, the right-hand expression is never evaluated. The null-coalescing operator also works with nullable value types (see "Nullable Types" on page 162 in Chapter 4).

Null-conditional operator (C# 6)

The ?. operator is the *null-conditional* or "Elvis" operator, and is new to C# 6. It allows you to call methods and access members just like the standard dot operator, except that if the operand on the left is null, the expression evaluates to null instead of throwing a NullReferenceException:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString();  // No error; s instead evaluates to null
```

The last line is equivalent to:

```
string s = (sb == null ? null : sb.ToString());
```

Upon encountering a null, the Elvis operator short-circuits the remainder of the expression. In the following example, s evaluates to null, even with a standard dot operator between ToString() and ToUpper():

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString().ToUpper();   // s evaluates to null without error
```

Repeated use of Elvis is necessary only if the operand immediately to its left may be null. The following expression is robust to both `x` being null and `x.y` being null:

```
x?.y?.z
```

and is equivalent to the following (except that `x.y` is evaluated only once):

```
x == null ? null
          : (x.y == null ? null : x.y.z)
```

The final expression must be capable of accepting a null. The following is illegal:

```
System.Text.StringBuilder sb = null;
int length = sb?.ToString().Length;   // Illegal : int cannot be null
```

We can fix this with the use of nullable value types (see "Nullable Types" on page 162 in Chapter 4): If you're already familiar with nullable types, here's a preview:

```
int? length = sb?.ToString().Length;   // OK : int? can be null
```

You can also use the null-conditional operator to call a void method:

```
someObject?.SomeVoidMethod();
```

If `someObject` is null, this becomes a "no-operation" rather than throwing a `NullReferenceException`.

The null-conditional operator can be used with the commonly used type members that we describe in Chapter 3, including *methods*, *fields*, *properties* and *indexers*. It also combines well with the *null-coalescing operator*:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString() ?? "nothing";   // s evaluates to "nothing"
```

The last line is equivalent to:

```
string s = (sb == null ? "nothing" : sb.ToString());
```

Statements

Functions comprise statements that execute sequentially in the textual order in which they appear. A *statement block* is a series of statements appearing between braces (the `{}` tokens).

Declaration Statements

A declaration statement declares a new variable, optionally initializing the variable with an expression. A declaration statement ends in a semicolon. You may declare multiple variables of the same type in a comma-separated list. For example:

```
string someWord = "rosebud";
int someNumber = 42;
bool rich = true, famous = false;
```

A constant declaration is like a variable declaration, except that it cannot be changed after it has been declared, and the initialization must occur with the declaration (see "Constants" on page 83 in Chapter 3):

```
const double c = 2.99792458E08;
c += 10;                        // Compile-time Error
```


Local variables

The scope of a local variable or local constant extends throughout the current block. You cannot declare another local variable with the same name in the current block or in any nested blocks. For example:

```
static void Main()
{
  int x;
  {
    int y;
    int x;            // Error - x already defined
  }
  {
    int y;            // OK - y not in scope
  }
  Console.Write (y);  // Error - y is out of scope
}
```

A variable's scope extends in *both directions* throughout its code block. This means that if we moved the initial declaration of x in this example to the bottom of the method, we'd get the same error. This is in contrast to C++ and is somewhat peculiar, given that it's not legal to refer to a variable or constant before it's declared.

Expression Statements

Expression statements are expressions that are also valid statements. An expression statement must either change state or call something that might change state. Changing state essentially means changing a variable. The possible expression statements are:

- Assignment expressions (including increment and decrement expressions)
- Method call expressions (both void and nonvoid)
- Object instantiation expressions

Here are some examples:

```
// Declare variables with declaration statements:
string s;
int x, y;
System.Text.StringBuilder sb;

// Expression statements
```

```
x = 1 + 2;                 // Assignment expression
x++;                       // Increment expression
y = Math.Max (x, 5);       // Assignment expression
Console.WriteLine (y);     // Method call expression
sb = new StringBuilder();  // Assignment expression
new StringBuilder();       // Object instantiation expression
```

When you call a constructor or a method that returns a value, you're not obliged to use the result. However, unless the constructor or method changes state, the statement is completely useless:

```
new StringBuilder();     // Legal, but useless
new string ('c', 3);     // Legal, but useless
x.Equals (y);            // Legal, but useless
```

Selection Statements

C# has the following mechanisms to conditionally control the flow of program execution:

- Selection statements (`if`, `switch`)
- Conditional operator (`?:`)
- Loop statements (`while`, `do..while`, `for`, `foreach`)

This section covers the simplest two constructs: the `if-else` statement and the `switch` statement.

The if statement

An `if` statement executes a statement if a `bool` expression is true. For example:

```
if (5 < 2 * 3)
  Console.WriteLine ("true");       // true
```

The statement can be a code block:

```
if (5 < 2 * 3)
{
  Console.WriteLine ("true");
  Console.WriteLine ("Let's move on!");
}
```

The else clause

An `if` statement can optionally feature an `else` clause:

```
if (2 + 2 == 5)
  Console.WriteLine ("Does not compute");
else
  Console.WriteLine ("False");        // False
```

Within an `else` clause, you can nest another `if` statement:

```
if (2 + 2 == 5)
  Console.WriteLine ("Does not compute");
else
  if (2 + 2 == 4)
    Console.WriteLine ("Computes");    // Computes
```

Changing the flow of execution with braces

An `else` clause always applies to the immediately preceding `if` statement in the statement block. For example:

```
if (true)
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("executes");
```

This is semantically identical to:

```
if (true)
{
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("executes");
}
```

We can change the execution flow by moving the braces:

```
if (true)
{
  if (false)
    Console.WriteLine();
}
else
  Console.WriteLine ("does not execute");
```

With braces, you explicitly state your intention. This can improve the readability of nested `if` statements—even when not required by the compiler. A notable exception is with the following pattern:

```
static void TellMeWhatICanDo (int age)
{
  if (age >= 35)
    Console.WriteLine ("You can be president!");
  else if (age >= 21)
    Console.WriteLine ("You can drink!");
  else if (age >= 18)
    Console.WriteLine ("You can vote!");
  else
    Console.WriteLine ("You can wait!");
}
```

Here, we've arranged the `if` and `else` statements to mimic the "elseif" construct of other languages (and C#'s `#elif` preprocessor directive). Visual Studio's auto-formatting recognizes this pattern and preserves the indentation. Semantically,

though, each if statement following an else statement is functionally nested within the else clause.

The switch statement

switch statements let you branch program execution based on a selection of possible values that a variable may have. switch statements may result in cleaner code than multiple if statements, since switch statements require an expression to be evaluated only once. For instance:

```
static void ShowCard(int cardNumber)
{
  switch (cardNumber)
  {
    case 13:
      Console.WriteLine ("King");
      break;
    case 12:
      Console.WriteLine ("Queen");
      break;
    case 11:
      Console.WriteLine ("Jack");
      break;
    case -1:                          // Joker is -1
      goto case 12;                   // In this game joker counts as queen
    default:                          // Executes for any other cardNumber
      Console.WriteLine (cardNumber);
      break;
  }
}
```

You can only switch on an expression of a type that can be statically evaluated, which restricts it to the built-in integral types, bool, and enum types (and nullable versions of these—see Chapter 4), and string type.

At the end of each case clause, you must say explicitly where execution is to go next, with some kind of jump statement. Here are the options:

- break (jumps to the end of the switch statement)
- goto case *x* (jumps to another case clause)
- goto default (jumps to the default clause)
- Any other jump statement—namely, return, throw, continue, or goto *label*

When more than one value should execute the same code, you can list the common cases sequentially:

```
switch (cardNumber)
{
  case 13:
  case 12:
  case 11:
```

```
        Console.WriteLine ("Face card");
        break;
      default:
        Console.WriteLine ("Plain card");
        break;
    }
```

This feature of a `switch` statement can be pivotal in terms of producing cleaner code than multiple `if-else` statements.

Iteration Statements

C# enables a sequence of statements to execute repeatedly with the `while`, `do-while`, `for`, and `foreach` statements.

while and do-while loops

`while` loops repeatedly execute a body of code while a `bool` expression is true. The expression is tested *before* the body of the loop is executed. For example:

```
int i = 0;
while (i < 3)
{
  Console.WriteLine (i);
  i++;
}

OUTPUT:
0
1
2
```

`do-while` loops differ in functionality from `while` loops only in that they test the expression *after* the statement block has executed (ensuring that the block is always executed at least once). Here's the preceding example rewritten with a `do-while` loop:

```
int i = 0;
do
{
  Console.WriteLine (i);
  i++;
}
while (i < 3);
```

for loops

`for` loops are like `while` loops with special clauses for *initialization* and *iteration* of a loop variable. A `for` loop contains three clauses as follows:

```
for (initialization-clause; condition-clause; iteration-clause)
  statement-or-statement-block
```

Initialization clause
: Executed before the loop begins; used to initialize one or more *iteration* variables

Condition clause
: The `bool` expression that, while true, will execute the body

Iteration clause
: Executed *after* each iteration of the statement block; used typically to update the iteration variable

For example, the following prints the numbers 0 through 2:

```
for (int i = 0; i < 3; i++)
  Console.WriteLine (i);
```

The following prints the first 10 Fibonacci numbers (where each number is the sum of the previous two):

```
for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
{
  Console.WriteLine (prevFib);
  int newFib = prevFib + curFib;
  prevFib = curFib; curFib = newFib;
}
```

Any of the three parts of the `for` statement may be omitted. One can implement an infinite loop such as the following (though `while(true)` may be used instead):

```
for (;;)
  Console.WriteLine ("interrupt me");
```

foreach loops

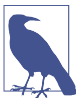
The `foreach` statement iterates over each element in an enumerable object. Most of the types in C# and the .NET Framework that represent a set or list of elements are enumerable. For example, both an array and a string are enumerable. Here is an example of enumerating over the characters in a string, from the first character through to the last:

```
foreach (char c in "beer")   // c is the iteration variable
  Console.WriteLine (c);

OUTPUT:
b
e
e
r
```

We define enumerable objects in "Enumeration and Iterators" on page 156 in Chapter 4.

Jump Statements

The C# jump statements are `break`, `continue`, `goto`, `return`, and `throw`.



Jump statements obey the reliability rules of `try` statements (see "try Statements and Exceptions" on page 148 in Chapter 4). This means that:

- A jump out of a `try` block always executes the `try`'s `finally` block before reaching the target of the jump.
- A jump cannot be made from the inside to the outside of a `finally` block (except via `throw`).

The break statement

The `break` statement ends the execution of the body of an iteration or `switch` statement:

```
int x = 0;
while (true)
{
  if (x++ > 5)
    break ;      // break from the loop
}
// execution continues here after break
...
```

The continue statement

The `continue` statement forgoes the remaining statements in a loop and makes an early start on the next iteration. The following loop skips even numbers:

```
for (int i = 0; i < 10; i++)
{
  if ((i % 2) == 0)       // If i is even,
    continue;             // continue with next iteration

  Console.Write (i + " ");
}
OUTPUT: 1 3 5 7 9
```

The goto statement

The `goto` statement transfers execution to another label within a statement block. The form is as follows:

```
goto statement-label;
```

Or, when used within a `switch` statement:

```
goto case case-constant;
```

A label is a placeholder in a code block that precedes a statement, denoted with a colon suffix. The following iterates the numbers 1 through 5, mimicking a for loop:

```
int i = 1;
startLoop:
if (i <= 5)
{
  Console.Write (i + " ");
  i++;
  goto startLoop;
}

OUTPUT: 1 2 3 4 5
```

The goto case *case-constant* transfers execution to another case in a switch block (see "The switch statement" on page 60).

The return statement

The return statement exits the method and must return an expression of the method's return type if the method is nonvoid:

```
static decimal AsPercentage (decimal d)
{
  decimal p = d * 100m;
  return p;             // Return to the calling method with value
}
```

A return statement can appear anywhere in a method (except in a finally block).

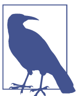
The throw statement

The throw statement throws an exception to indicate an error has occurred (see "try Statements and Exceptions" on page 148 in Chapter 4):

```
if (w == null)
  throw new ArgumentNullException (...);
```

Miscellaneous Statements

The using statement provides an elegant syntax for calling Dispose on objects that implement IDisposable, within a finally block (see "try Statements and Exceptions" on page 148 in Chapter 4 and "IDisposable, Dispose, and Close" on page 499 in Chapter 12).



C# overloads the using keyword to have independent meanings in different contexts. Specifically, the using *directive* is different from the using *statement*.

The lock statement is a shortcut for calling the Enter and Exit methods of the Monitor class (see Chapter 14 and Chapter 23).

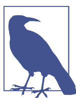
Namespaces

A namespace is a domain for type names. Types are typically organized into hierarchical namespaces, making them easier to find and avoiding conflicts. For example, the `RSA` type that handles public key encryption is defined within the following namespace:

```
System.Security.Cryptography
```

A namespace forms an integral part of a type's name. The following code calls `RSA`'s `Create` method:

```
System.Security.Cryptography.RSA rsa =
  System.Security.Cryptography.RSA.Create();
```



Namespaces are independent of assemblies, which are units of deployment such as an *.exe* or *.dll* (described in Chapter 18).

Namespaces also have no impact on member visibility—`public`, `internal`, `private`, and so on.

The `namespace` keyword defines a namespace for types within that block. For example:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
  class Class2 {}
}
```

The dots in the namespace indicate a hierarchy of nested namespaces. The code that follows is semantically identical to the preceding example:

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

You can refer to a type with its *fully qualified name*, which includes all namespaces from the outermost to the innermost. For example, we could refer to `Class1` in the preceding example as `Outer.Middle.Inner.Class1`.

Types not defined in any namespace are said to reside in the *global namespace*. The global namespace also includes top-level namespaces, such as `Outer` in our example.

The using Directive

The `using` directive *imports* a namespace, allowing you to refer to types without their fully qualified names. The following imports the previous example's `Outer.Middle.Inner` namespace:

```
using Outer.Middle.Inner;

class Test
{
  static void Main()
  {
    Class1 c;    // Don't need fully qualified name
  }
}
```

It's legal (and often desirable) to define the same type name in different namespaces. However, you'd typically do so only if it was unlikely for a consumer to want to import both namespaces at once. A good example, from the .NET Framework, is the `TextBox` class which is defined both in `System.Windows.Controls` (WPF) and `System.Web.UI.WebControls` (ASP.NET).

using static (C# 6)

From C# 6, you can import not just a namespace, but a specific type, with the `using static` directive. All static members of that type can then be used without being qualified with the type name. In the following example, we call the `Console` class's static `WriteLine` method:

```
using static System.Console;

class Test
{
  static void Main() { WriteLine ("Hello"); }
}
```

The `using static` directive imports all accessible static members of the type, including fields, properties and nested types (Chapter 3). You can also apply this directive to enum types (Chapter 3), in which case their members are imported. So, if we import the following enum type:

```
using static System.Windows.Visibility;
```

we can specify `Hidden` instead of `Visibility.Hidden`:

```
var textBox = new TextBox { Visibility = Hidden };   // XAML-style
```

Should an ambiguity arise between multiple static imports, the C# compiler is not smart enough to infer the correct type from the context, and will generate an error.

Rules Within a Namespace

Name scoping

Names declared in outer namespaces can be used unqualified within inner namespaces. In this example, `Class1` does not need qualification within `Inner`:

```
namespace Outer
{
  class Class1 {}

  namespace Inner
  {
    class Class2 : Class1  {}
  }
}
```

If you want to refer to a type in a different branch of your namespace hierarchy, you can use a partially qualified name. In the following example, we base `SalesReport` on `Common.ReportBase`:

```
namespace MyTradingCompany
{
  namespace Common
  {
    class ReportBase {}
  }
  namespace ManagementReporting
  {
    class SalesReport : Common.ReportBase  {}
  }
}
```

Name hiding

If the same type name appears in both an inner and an outer namespace, the inner name wins. To refer to the type in the outer namespace, you must qualify its name. For example:

```
namespace Outer
{
  class Foo { }

  namespace Inner
  {
    class Foo { }

    class Test
    {
      Foo f1;         // = Outer.Inner.Foo
      Outer.Foo f2;   // = Outer.Foo
    }
```

```
  }
}
```

All type names are converted to fully qualified names at compile time. Intermediate language (IL) code contains no unqualified or partially qualified names.

Repeated namespaces

You can repeat a namespace declaration, as long as the type names within the namespaces don't conflict:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}

namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

We can even break the example into two source files such that we could compile each class into a different assembly.

Source file 1:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}
```

Source file 2:

```
namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

Nested using directive

You can nest a `using` directive within a namespace. This allows you to scope the `using` directive within a namespace declaration. In the following example, `Class1` is visible in one scope, but not in another:

```
namespace N1
{
  class Class1 {}
}

namespace N2
{
  using N1;

  class Class2 : Class1 {}
```

```
}

namespace N2
{
  class Class3 : Class1 {}   // Compile-time error
}
```


Aliasing Types and Namespaces

Importing a namespace can result in type-name collision. Rather than importing the whole namespace, you can import just the specific types you need, giving each type an alias. For example:

```
using PropertyInfo2 = System.Reflection.PropertyInfo;
class Program { PropertyInfo2 p; }
```

An entire namespace can be aliased, as follows:

```
using R = System.Reflection;
class Program { R.PropertyInfo p; }
```

Advanced Namespace Features

Extern

Extern aliases allow your program to reference two types with the same fully qualified name (i.e., the namespace and type name are identical). This is an unusual scenario and can occur only when the two types come from different assemblies. Consider the following example.

Library 1:

```
// csc target:library /out:Widgets1.dll widgetsv1.cs

namespace Widgets
{
  public class Widget {}
}
```

Library 2:

```
// csc target:library /out:Widgets2.dll widgetsv2.cs

namespace Widgets
{
  public class Widget {}
}
```

Application:

```
// csc /r:Widgets1.dll /r:Widgets2.dll application.cs

using Widgets;

class Test
{
```

```
    static void Main()
    {
      Widget w = new Widget();
    }
  }
```

The application cannot compile, because `Widget` is ambiguous. Extern aliases can resolve the ambiguity in our application:

```
// csc /r:W1=Widgets1.dll /r:W2=Widgets2.dll application.cs

extern alias W1;
extern alias W2;

class Test
{
  static void Main()
  {
    W1.Widgets.Widget w1 = new W1.Widgets.Widget();
    W2.Widgets.Widget w2 = new W2.Widgets.Widget();
  }
}
```

Namespace alias qualifiers

As we mentioned earlier, names in inner namespaces hide names in outer namespaces. However, sometimes even the use of a fully qualified type name does not resolve the conflict. Consider the following example:

```
namespace N
{
  class A
  {
    public class B {}                    // Nested type
    static void Main() { new A.B(); }    // Instantiate class B
  }
}

namespace A
{
  class B {}
}
```

The `Main` method could be instantiating either the nested class `B`, or the class `B` within the namespace `A`. The compiler always gives higher precedence to identifiers in the current namespace; in this case, the nested `B` class.

To resolve such conflicts, a namespace name can be qualified, relative to one of the following:

- The global namespace—the root of all namespaces (identified with the contextual keyword `global`)
- The set of extern aliases

The `::` token is used for namespace alias qualification. In this example, we qualify using the global namespace (this is most commonly seen in auto-generated code to avoid name conflicts):

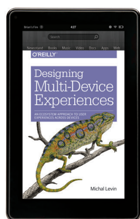
```
namespace N
{
  class A
  {
    static void Main()
    {
      System.Console.WriteLine (new A.B());
      System.Console.WriteLine (new global::A.B());
    }

    public class B {}
  }
}

namespace A
{
  class B {}
}
```

Here is an example of qualifying with an alias (adapted from the example in "Extern" on page 69):

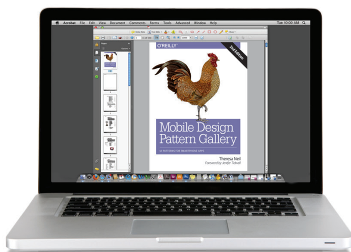
```
extern alias W1;
extern alias W2;
class Test
{
  static void Main()
  {
    W1::Widgets.Widget w1 = new W1::Widgets.Widget();
    W2::Widgets.Widget w2 = new W2::Widgets.Widget();
  }
}
```

O'Reilly ebooks.

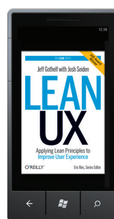
Your bookshelf on your devices.



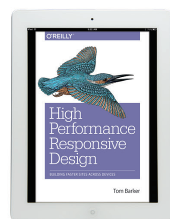
PDF



Mobi



ePub



DAISY

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 DAISY—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 ebooks.oreilly.com

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

O'REILLY®

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 15055