Instant Help for C# 5.0 Programmers

C# 5.0

Pocket Reference

O'REILLY®

Joseph Albahari
& Ben Albahari

Programming Languages/C#

O'REILLY®

C# 5.0 Pocket Reference

When you need answers for programming with C# 5.0, this practical and tightly focused book tells you exactly what you need to know—without long introductions or bloated samples. Easy to browse, it's ideal as a quick reference or as a guide to get you rapidly up to speed if you already know Java, C++, or an earlier version of C#.

Written by the authors of *C# 5.0 in a Nutshell*, this book covers C# 5.0 language essentials, including:

- All of C#'s fundamentals
- Advanced topics such as operator overloading, type constraints, covariance and contravariance, iterators, nullable types, operator lifting, lambda expressions, and closures
- LINQ, starting with sequences, lazy execution, and standard query operators, and finishing with a complete reference to query expressions
- Dynamic binding and C# 5.0's new asynchronous functions
- Unsafe code and pointers, custom attributes, preprocessor directives, and XML documentation

Joe Albahari is the author of LINQPad and has been a C# MVP since 2008. He has been programming for 20 years and has published 7 books with O'Reilly.

Ben Albahari, a former program manager at Microsoft, is the founder of TakeOnIt, a website for comparing the opinions of experts, leaders, and organizations on controversial topics.

oreilly.com
Twitter: @oreillymedia
facebook.com/oreilly

US $14.99 **CAN $15.99**

ISBN: 978-1-449-32017-1

C# 5.0

Pocket Reference

C# 5.0
Pocket Reference

Joseph Albahari and Ben Albahari

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

C# 5.0 Pocket Reference
by Joseph Albahari and Ben Albahari

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

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

Editor: Rachel Roumeliotis
Copyeditor: Audrey Doyle
Production Editor: Iris Febres
Proofreader: Jasmine Perez
Indexer: Angela Howard
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

June 2012: First Edition.

Revision History for the First Edition:
2012-05-25 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449320171* for release details.

ISBN: 978-1-449-32017-1

[M]

1337976996

Contents

C# 5.0 Pocket Reference **1**

Conventions Used in This Book 2
Using Code Examples 2
Safari® Books Online 3
How to Contact Us 4
A First C# Program 4
Syntax 8
Type Basics 11
Numeric Types 20
Boolean Type and Operators 27
Strings and Characters 29
Arrays 32
Variables and Parameters 36
Expressions and Operators 43
Statements 49
Namespaces 56
Classes 60
Inheritance 71
The object Type 79
Structs 83
Access Modifiers 84
Interfaces 86

Enums 89
Nested Types 92
Generics 93
Delegates 101
Events 108
Lambda Expressions 113
Anonymous Methods 117
try Statements and Exceptions 118
Enumeration and Iterators 126
Nullable Types 132
Operator Overloading 136
Extension Methods 139
Anonymous Types 141
LINQ 142
Dynamic Binding 166
Attributes 175
Caller Info Attributes (C# 5.0) 178
Asynchronous Functions (C# 5.0) 180
Unsafe Code and Pointers 189
Preprocessor Directives 193
XML Documentation 196

Index **203**

C# 5.0 Pocket Reference

C# is a general-purpose, type-safe, object-oriented programming language. The goal of the language is programmer productivity. To this end, the language balances simplicity, expressiveness, and performance. The C# language is platform-neutral, but it was written to work well with the Microsoft *.NET Framework*. C# 5.0 targets .NET Framework 4.5.

NOTE

The programs and code snippets in this book mirror those in Chapters 2 through 4 of *C# 5.0 in a Nutshell* and are all 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, click the Samples tab in LINQPad and click "Download more samples". LINQPad is free—go to *http://www.linqpad.net*.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

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

CAUTION

This icon indicates a warning or caution.

Using Code Examples

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

quire permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*C# 5.0 Pocket Reference* by Joseph Albahari and Ben Albahari (O'Reilly). Copyright 2012 Joseph Albahari and Ben Albahari, 978-1-449-3201-71."

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

Safari® Books Online

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

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

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

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreil.ly/CSharp5_PR

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

bookquestions@oreilly.com

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

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

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

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

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*. Statements in C# execute sequentially and are terminated by a semicolon. 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`.

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, both of type `int` (integer).

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

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

NOTE

An array (such as `string[]`) represents a fixed number of elements of a particular type (for more information, see "Arrays" on page 32).

Methods are one of several kinds of functions in C#. Another kind of function we used was the * *operator*, used to perform multiplication. There are also *constructors*, *properties*, *events*, *indexers*, and *finalizers*.

In our example, the two methods are grouped into a class. A *class* groups function members and data members to form an object-oriented building block. The `Console` class groups members that handle command-line input/output functionality, such as the `WriteLine` method. Our `Test` class groups two methods—the `Main` method and the `FeetToInches` method. A class is a kind of *type*, which we will examine in the section "Type Basics" on page 11.

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

The name of the C# compiler is *csc.exe*. You can either use an integrated development environment (IDE), such as Visual Studio, to compile, or call `csc` manually from the command line. To compile manually, first save a program to a file such as *MyFirstProgram.cs*, and then go to the command line and invoke `csc` (located under *%SystemRoot%\Microsoft.NET\Framework\<framework-version>* where *%SystemRoot%* is your Windows directory) as follows:

```
csc MyFirstProgram.cs
```

This produces an application named *MyFirstProgram.exe*.

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

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

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

```
using   class   static   void   int
```

Here is the full list of C# keywords:

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

Avoiding conflicts

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

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

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

Contextual keywords

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

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

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 in our example program are `12` and `30`. *Punctuators* help demarcate the structure of the program. The punctuators in our program are `{`, `}`, and `;`.

The braces group multiple statements into a *statement block*. The semicolon terminates a (nonblock) statement. 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, `*`. The operators in our program are:

```
. ()  *  =
```

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. The equals sign performs

assignment (the double equals, ==, performs equality comparison).

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 (see "XML Documentation" on page 196).

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`.

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).

All values in C# are an *instance* of a specific 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 (also called built-in 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 $2^{31}-1$. We can perform func-

tions 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 = 2012;
message = message + x.ToString();
Console.WriteLine (message);          // Hello world2012
```

The predefined `bool` type has exactly two possible values: `true` and `false`. The `bool` type is commonly used to conditionally branch execution flow 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");
```

NOTE

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)   // Constructor
  {
    ratio = unitRatio;
  }

  public int Convert (int unit)          // Method
  {
    return unit * ratio;
  }
}

class Test
{
  static void Main()
  {
    UnitConverter feetToInches = new UnitConverter(12);
    UnitConverter milesToFeet = new UnitConverter(5280);

    Console.Write (feetToInches.Convert(30));   // 360
    Console.Write (feetToInches.Convert(100));  // 1200
    Console.Write (feetToInches.Convert
                    (milesToFeet.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`. Similarly, 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 started our `Main` method by creating two instances of the `UnitConverter` type. 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 UnitConverter (int unitRatio)   // Constructor
{
  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.

To contrast instance with static members, 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 instance field
    Population = Population+1;  // Increment static field
  }
}
```

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

```
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
```

The public keyword

The `public` keyword exposes members to other classes. In this example, if the `Name` field in `Panda` was not public, 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 whereas 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 bitwise capacity of an `int`):