

Programming in C#

Rex Jaeschke

Programming in C#

© 2001–2002, 2004–2005, 2007, 2009 Rex Jaeschke. All rights reserved.

Edition: 2.0

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means whatsoever, except in the case of brief quotations embodied in critical reviews and articles.

The information in this book is subject to change without notice. It should not be construed as a commitment by the author or the publisher. Although every precaution has been taken in the preparation of this book, the author and the publisher assume no responsibility for errors or omissions.

Java and JavaScript are trademarks of Sun Microsystems.

.NET, Visual Basic, Visual C#, Visual C++, Visual Studio, and JScript are trademarks of Microsoft.

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

| | |
|---|------------|
| Preface | vii |
| C#'s Design Goals | viii |
| Reader Assumptions | ix |
| Limitations..... | ix |
| Presentation Style | x |
| Exercises and Solutions | x |
| The Status of C# | x |
| Acknowledgments..... | xi |
| 1. The Basics | 1 |
| 1.1 Basic Program Structure | 1 |
| 1.2 Identifiers..... | 6 |
| 1.3 Introduction to Formatted Output | 7 |
| 1.4 Data Types | 14 |
| 1.4.1 The <code>bool</code> Type..... | 14 |
| 1.4.2 The Integer Types | 14 |
| 1.4.3 The Floating-Point Types | 17 |
| 1.4.4 The <code>decimal</code> Type | 18 |
| 1.4.5 Enumeration Types..... | 18 |
| 1.4.6 User-Defined Types | 20 |
| 1.4.7 The <code>string</code> Type | 20 |
| 1.5 Constants | 22 |
| 1.6 Literals..... | 22 |
| 1.6.1 Boolean literals | 22 |
| 1.6.2 Character Literals..... | 22 |
| 1.6.3 String Literals | 22 |
| 1.6.4 Integer Literals..... | 23 |
| 1.6.5 Floating-Point Literals..... | 24 |
| 1.6.6 Decimal Literals | 25 |
| 1.7 Operator Precedence..... | 26 |
| 1.8 Type Conversion | 28 |
| 1.9 Arithmetic Overflow | 31 |
| 1.10 Introduction to Class Members | 32 |
| 1.11 Nullable Types..... | 38 |
| 2. Looping, Testing, and Branching | 41 |
| 2.1 The <code>while</code> Statement | 41 |
| 2.2 The <code>for</code> Statement..... | 47 |
| 2.3 The <code>do/while</code> Statement..... | 51 |
| 2.4 The <code>if/else</code> Statement..... | 51 |
| 2.5 The <code>return</code> Statement | 54 |
| 2.6 The <code>break</code> Statement | 54 |
| 2.7 The <code>continue</code> Statement..... | 55 |
| 2.8 The <code>goto</code> Statement..... | 56 |
| 2.9 The Empty Statement | 57 |
| 2.10 The <code>switch</code> Statement | 58 |
| 3. Methods | 63 |
| 3.1 Introduction | 63 |
| 3.2 Passing Arguments..... | 66 |
| 3.2.1 Value Parameters | 67 |

| | | |
|-----------|---|------------|
| 3.2.2 | Reference Parameters | 68 |
| 3.2.3 | Output Parameters | 69 |
| 3.3 | Recursion | 69 |
| 3.4 | Argument Type Matching | 70 |
| 3.5 | Overloaded Methods | 71 |
| 3.6 | Generic Methods | 76 |
| 3.7 | Issues Regarding <code>Main</code> | 77 |
| 3.8 | Variable-Length Argument Lists | 77 |
| 3.9 | Programs, Assemblies, and Accessibility | 78 |
| 4. | References, Strings, and Arrays | 85 |
| 4.1 | Introduction | 85 |
| 4.2 | Sharing of Like Strings | 90 |
| 4.3 | Passing and Returning References | 91 |
| 4.4 | Allocating Memory for Objects | 92 |
| 4.5 | Releasing Allocated Memory | 93 |
| 4.6 | Arrays | 95 |
| 4.6.1 | Command-Line Arguments | 104 |
| 4.6.2 | Copying Arrays | 105 |
| 4.7 | Variable-Length Argument Lists | 112 |
| 4.8 | Modifiable Strings | 113 |
| 5. | Classes | 117 |
| 5.1 | Introduction | 117 |
| 5.2 | Class-Specific Methods | 119 |
| 5.3 | Data Hiding within a Class | 123 |
| 5.4 | <code>this</code> and <code>That</code> | 128 |
| 5.5 | Static Fields and Methods Revisited | 130 |
| 5.6 | Containment | 134 |
| 5.7 | Copying Objects | 136 |
| 5.8 | Object Finalization | 137 |
| 5.9 | Resource Disposal | 139 |
| 5.10 | Nested Classes | 142 |
| 5.11 | Partial Classes | 144 |
| 6. | Inheritance | 145 |
| 6.1 | Introduction | 145 |
| 6.2 | The Is-A versus Has-A Test | 147 |
| 6.3 | A Simple Class Hierarchy | 150 |
| 6.4 | Abstract Classes and Methods | 154 |
| 6.5 | Constructor Calls | 157 |
| 6.6 | Protected Members | 158 |
| 6.7 | Disabling Inheritance | 161 |
| 6.8 | Hiding versus Overriding | 161 |
| 6.9 | Disabling Overriding | 165 |
| 6.10 | A Universal Base Class | 165 |
| 6.11 | Object Finalization Revisited | 167 |
| 6.12 | Run-Time Type Checking | 168 |
| 6.13 | Arrays and Inheritance | 172 |
| 6.14 | Enums and Inheritance | 172 |
| 6.15 | Value Types and Inheritance | 174 |

| | | |
|------------|--|------------|
| 6.16 | Boxing and Unboxing | 174 |
| 6.17 | Inheritance after the Fact | 175 |
| 6.18 | Interfaces | 175 |
| 6.19 | Generic Types | 179 |
| 6.19.1 | Defining a Generic Type | 179 |
| 6.19.2 | Using a Generic Type | 182 |
| 6.19.3 | Generic Type Constraints | 184 |
| 7. | Exception Handling | 185 |
| 7.1 | Introduction | 185 |
| 7.2 | Catching Exceptions | 186 |
| 7.3 | Throwing Exceptions | 188 |
| 7.4 | Handling Families of Exception Types | 193 |
| 7.5 | Catching Unexpected Exceptions | 195 |
| 7.6 | A Final Note | 195 |
| 8. | Operator Overloading | 197 |
| 8.1 | Introduction | 197 |
| 8.2 | A Simple Example | 198 |
| 8.3 | A Vector Class | 201 |
| 8.4 | Overloading Unary Operators | 204 |
| 8.5 | Overloading Binary Operators | 206 |
| 8.6 | Conversion Operators | 207 |
| 8.7 | Some Rules of Thumb | 209 |
| 9. | Delegates and Events | 211 |
| 9.1 | Introduction | 211 |
| 9.2 | Passing and Returning Delegates | 213 |
| 9.3 | Delegate Type Compatibility | 215 |
| 9.4 | Arrays of Delegates | 216 |
| 9.5 | Combining Delegates | 217 |
| 9.6 | System.Delegate | 221 |
| 9.7 | Events | 223 |
| 9.8 | Anonymous Delegates | 226 |
| 10. | Structs | 227 |
| 10.1 | Introduction | 227 |
| 10.2 | A Complex Number Example | 230 |
| 10.3 | Simple Type Mapping | 233 |
| 10.4 | Miscellaneous Notes | 233 |
| 11. | Namespaces | 235 |
| 11.1 | Introduction | 235 |
| 11.2 | Declaring Namespaces | 236 |
| 11.3 | using Namespace Directives | 238 |
| 11.4 | The Standard Namespaces | 238 |
| 11.5 | Namespace Aliases | 241 |
| 11.6 | Access Modifiers | 242 |
| 12. | Input and Output | 245 |
| 12.1 | Introduction | 245 |
| 12.2 | The Basic I/O Classes | 247 |

| | |
|---|------------|
| Programming in C# | |
| 12.3 File I/O | 248 |
| 12.4 String I/O..... | 251 |
| 12.5 Typed Unformatted I/O | 252 |
| 12.6 Random Access I/O | 254 |
| 12.7 File and Directory Operations..... | 256 |
| 12.8 Miscellaneous Issues | 257 |
| 13. The Preprocessor | 259 |
| 13.1 Introduction | 259 |
| 13.2 Conditional Compilation Symbols..... | 259 |
| 13.3 Conditional Compilation | 260 |
| 13.4 Diagnostic Directives | 261 |
| 13.4.1 The #error Directive..... | 261 |
| 13.4.2 The #warning Directive | 261 |
| 13.5 Line Directives..... | 262 |
| 13.6 Region Directives | 262 |
| Annex A. Operator Precedence | 265 |
| Annex B. C# Keywords | 269 |
| Index | 271 |

Preface

Welcome to the world of C#. Throughout this book, we will look at the statements and constructs of the C# programming language. Each statement and construct will be introduced by example with corresponding explanations, and, except where errors are intentional; the examples will be complete programs or subroutines that are error-free. I encourage you to run and modify these examples, because the only realistic way to master a language is to write programs in it, run them, and debug them.

This book was written with teaching in mind, and it evolved from an earlier series on C, C++, and Java. It is intended for use both in a classroom environment as well as for self-paced learning.

C# is a robust, general-purpose, high-level language that supports object-oriented programming (OOP). From a grammatical viewpoint, C# is a relatively simple language having about 75 keywords. Because of the power of the basic statements and constructs, and the fact that the programmer can effectively extend the language by using callable methods while still maintaining portability, there really is no need for language extensions especially considering the fact that C# supports calls to procedures written in other languages.

Depending on their language backgrounds, programmers new to C# may initially find programs hard to read because, traditionally, most code is written in lowercase. Lower- and uppercase letters are treated by the compiler as distinct. In fact, C# language keywords *must* be written in lowercase. Keywords are also reserved words.

C# uses almost all of those special punctuation marks on most keyboards for one reason or another and sometimes combines them for yet other purposes. C# supports the 16-bit character set ISO 10646 UCS-2—commonly known as Unicode—of which ASCII is a proper subset. (Character codes 0–127 represent the same characters in both sets.) As such, it does not suffer from many of the difficulties faced by other languages that were designed primarily to support a “USA-English” mode of programming.

C# encourages a structured, modular approach to programming using *classes* containing callable subroutines, known as *methods*. Classes can be compiled separately, with external references being resolved at runtime.

C# is an architecturally neutral language. When compiled by the current implementations, the output produced is in the form of Common Intermediate Language (CIL).¹ During program execution on a .NET/CLI environment, this intermediate language is compiled to native code, which is then executed. CIL is not language-specific, so a program can be made up from modules written in any language that can produce CIL.² Note, however, that the C# Standard makes no mention of CIL or .NET/CLI, and permits implementations to generate code for other platforms.

¹ Microsoft's implementation calls this language MSIL. CIL is the name given to it by the committee that standardized it.

² Currently, compilers for more than 15 different languages produce CIL. These include Microsoft's own Visual Basic, Visual C++, Visual C#, and JScript.

Programming in C#

C# lends itself to writing terse code. However, there is a fine, and subjective, line between writing code that is terse, and code that is cryptic. It is easy to write code that is unreadable and, therefore, unmaintainable. In fact, that's what you will get by default. However, with care and a small dose of discipline, you can produce nicely formatted code that is easy to read and maintain. However, remember, good code doesn't happen automatically—you have to work at it. Throughout the book, I will make numerous comments and suggestions regarding style. Perhaps the best advice I can give is "Remember that the poor fool who has to read your code in the future just might be you!" Above all, be overt and be consistent.

Numerous "Tips" and "Style Tips" have been added throughout to highlight important suggestions. Considerable attention has been paid to using and teaching a consistent, popular, and overt programming style. Both kinds of tips are highlighted so they stand out. For example:

Tip: The simplest way to convert the value of an expression to its corresponding string representation is to concatenate that expression with an empty string; for example, `"" + 123` results in the string `"123"`.

Style Tip: Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

C# is not all things to all people; nor does it claim to be. For many applications, assembly language, Pascal, COBOL, FORTRAN, and BASIC, for example, will do just fine. In any event, compared to other high-level languages, C# is a relatively inexpensive language to learn and master. It is certainly much less expensive to master—and far less error-prone—than C or C++. Learning C# is comparable to learning Java.

C#'s Design Goals

The C# language, library, and run-time environment were designed to deliver the following:

- Code reuse and reduced development effort via object-oriented programming support.
- Extremely broad portability by the elimination of almost all implementation-defined and unspecified behavior, by the use of an architecturally neutral language.
- Strong type checking.
- Support for threading.
- Easy to learn, read, and write.
- Support for internationalization.
- Support for developing software components.
- Easy integration with other languages and operating system environments.

Reader Assumptions

I assume that you know how to use your particular text editor, C# compiler, and debugger. Comments on the use of these utility programs will be limited to points of particular interest to the C# programmer.

To fully understand and exploit the material, you should be conversant with the following topics:

- The basic purpose of a compiler and linker.
- Number system theory.
- Bit operations such as AND, inclusive-OR, exclusive-OR, complement, and left- and right-shift.
- Data representation.
- Communication between procedures by passing arguments and/or by returning a value, as well as via global variables.
- Use of single- and multi-dimensional arrays.
- Creation and use of sequential files and formatted and unformatted I/O.
- Basic data structures such as linked lists.

Many of C#'s more powerful capabilities, particularly exception handling, inheritance, and threads, require advanced programming experience before they can be understood and exploited fully.

Despite the many similarities between C# and C, C++, and Java, you do *not* need to know any of these other languages to use this book. If you do know one or more of them, simply read quickly over those sections that seem familiar to you. I say, “read quickly” rather than “skip” because you may well find that C# defines things more fully or a little differently than do C/C++ or Java. For example, you might already “know all about the types `int` and `long`”, but if you skip the coverage on that topic, you'll miss learning that their actual size and representation is defined absolutely by C#, unlike in C and C++, where these things are implementation-defined.

So, if you already know C++ or Java, the C# learning curve should be short and not too steep. If you know C, you have all the OO stuff to learn as well. If you know some OO language other than C++ or Java you'll already be familiar with the OO concepts, but not the syntax that is largely common to C#, C, C++, and Java. In addition, if your programming background is in some procedural language such as Pascal, FORTRAN, or BASIC, you'll need to read the whole book closely and do all the exercises. As for those of you having only a COBOL background, you'll have some catching-up to do with respect to thinking in binary versus decimal when it comes to data representation, as well as the use of methods and their associated argument passing and value returning.

Limitations

This book covers almost all of the C# language.¹ It also introduces the core class library. However, only a very small percentage of library facilities is mentioned or covered in any detail. The common library used by C# (and any other language that targets CIL) contains so many classes and methods that books have been written about that subject alone.

¹ With V3.0 of its Visual C# V3.0 development environment, Microsoft introduced some new language features. These features are not covered by this book, nor are they part of the current C# Standard.

Although C# was designed for use in building software components, this book is directed at teaching the C# language proper, by writing simple stand-alone applications, since without a thorough knowledge of the language, you will not be able to understand and implement other, more advanced, solutions. In particular, GUI, threading, inter-process communications, and a number of other advanced features are covered in separate texts.

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming-language training courses for more than 20 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and well focused. Specifically, I introduce the basic elements and constructs of the language using procedural programming examples. Once those fundamentals have been mastered, I move on to object-oriented concepts and syntax. Then follow the more advanced language and library topics. Many books on object-oriented languages use objects, inheritance, exception handling, GUI, and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than is any other; I simply know that my approach works well, and has formed the basis of my successful seminar business for the last two decades.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named **Source**, where each chapter has its own subdirectory. By convention, the names of C# source files end in “.cs”.

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided in a directory tree named **Labs**, in which each chapter has its own subdirectory.¹ Exercises that are not so marked have no general solution and require experimentation or research in an implementation's documentation. Exercises having solutions contain a statement of the form “(See lab directory xx.)”, which indicates the corresponding solution or test file in the **Labs** subdirectory.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

The Status of C#

Microsoft announced the C# language and .NET platform in July 2000. At that time, Microsoft stated that both the language and some subset of the library and runtime environment would be submitted for standardization, to Ecma International, a standards organization. In September 2000, Ecma committee TC39, which was responsible for standardization of the scripting language ECMAScript (known commercially as JavaScript and JScript),

¹ The solutions are only available to licensees of these materials when they are used in formal training scenarios.

agreed to take on this new work. As a result, TC39 was split into three Task Groups: TG1 (ECMAScript), TG2 (C#), and TG3 (Common Library Infrastructure [CLI]). This author served as project editor of the standards produced by TG2 and TG3.

The submission to Ecma was sponsored by Hewlett-Packard, Intel, and Microsoft. A number of other companies also agreed to participate in the standards work, which began in November 2000. That work was completed in September 2001, and both standards were adopted by Ecma in December 2001. The standard for C# is ECMA-334, and that for CLI is ECMA-335. (These standards can be downloaded free of charge from www.ecma-international.org.)

In January 2002, a 6-month review and ballot period began within ISO/IEC JTC 1 to determine if these specifications should be adopted as ISO/IEC standards as well. After a ballot-resolution meeting in September 2002, these specifications were approved unanimously, and they were forwarded to ISO for publication. The designation of the C# standard is ISO/IEC 23270, while that for the CLI standard is ISO/IEC 23271. (See www.iso.org.)

A revision of the Ecma versions of these standards started in January 2003, and was completed in March 2005. Ecma adopted them in June 2005, and ISO/IEC JTC 1 adopted them in April 2006. This revision included a number of important additions: generic types and methods, static classes, anonymous methods, partial declarations, iterators, nullable types, and pragma directives.

Throughout this text, the original definition of C# is referred to as V1, while the revised version is referred to as V2. (Microsoft's current version, V3, is not covered by this book or the C# Standard.)

In 2008, Ecma transferred the Task Groups TG2 (C#) and TG3 (CLI) from Technical Committee TC39 to TC49.

Acknowledgments

Many thanks go to those people who reviewed all or part of this book. In particular, Pat Bria, and students in my introductory C# seminars provided useful feedback and located numerous typographical errors. Thanks also to the folks at Microsoft for their assistance, especially to Peter Golde, Peter Hallam, and Scott Wiltamuth for help on language issues, and Brad Abrams for help with questions regarding the class libraries.

Rex Jaeschke, September 2009

1. The Basics

1.1 Basic Program Structure

A correct source program is made up of an acceptable combination of tokens, where a *token* is the smallest possible unit of source. C# has five different kinds of tokens: identifiers, keywords, literals, operators, and punctuators.

For the most part, C# is a free-format language, and space between tokens is optional. However, in some cases, something is needed between tokens so they can be recognized the way they were intended. White space performs this function. *White space* consists of one or more consecutive characters from the following set: space, horizontal tab, vertical tab, form feed, and newline.¹ The newline character is entered by pressing the RETURN or ENTER key. In any source file, an arbitrary amount of white space is permitted before the first token, between any two adjacent tokens, or after the last token.

Style Tip: Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

Let's look at the basic structure of a C# program (see directory Ba01):

```
/* Ba01.cs - A sample C# program

In case 1 below, the text "Welcome to C#" is written to the
standard output device.

In case 2, the Main method returns to its caller. */

public class Ba01
{
    public static void Main()        // program starts here
    {
/*1*/        System.Console.WriteLine("Welcome to C#");

/*2*/        return;                // redundant
    }
}
```

¹ Other, much less often used, characters are also permitted; however, they are not discussed further here.

The output produced is

```
Welcome to C#
```

There are several ways to write a *comment*. The first form, a *delimited comment*, involves both a comment start and end delimiter, represented by `/*` and `*/`, respectively. These delimiters and all the characters contained between them are ignored by the compiler. This form of comment can span an arbitrary number of source lines, and it is treated as a single space, allowing it to occur anywhere white space can be used; that is, before the first token, after the last token, or between any two adjacent tokens.

Note the unusual comments, `/*1*/` and `/*2*/`, in the example above. Throughout this book, such comments are used to give source lines pseudo-labels, so they can be referenced directly in the narrative. In production code, this approach can also be used, as follows: A comment prior to a method provides a general introduction and describes the steps used to implement the solution. By giving each step a label, we can place a comment containing that label prior to the first statement that implements that step. For example, in the program above, the introductory comment mentions case 1 (which corresponds to step 1), and the comment `/*1*/` shows where that step (or case) is implemented.

The second form of comment is a *single-line comment*, which is begun by `//` and terminated by the end of that source line.

Comments of the same form do not nest; however, a comment of the form `/* . . . */` can contain `//` as text, and vice versa.

A third form of comment (not shown here) is known as a *documentation comment*.¹

A C# program consists of one or more *methods* that can be defined in one or more source files. (A method corresponds to what some other languages refer to as a *function*, a *procedure*, or a *subroutine*.) A program must contain at least one method, called `Main`. This specially named method indicates where the program is to begin execution.

Each method can have one or more *modifiers*; `Main` has two: `public` and `static`. Names having the `public` modifier are visible outside their parent *class*.² We'll learn about the `static` modifier in §5; however, for now, we simply need to use it here. The keyword `void` preceding the method name `Main` indicates that this method does not return a value. (As we shall see in §3.7, `Main` is permitted to return an integer value to the execution environment.)

The parentheses following the method name `Main` surround that method's formal *parameter list*.³ The parentheses are required, even if no arguments are expected, as is the case in this example. (`Main` is permitted to have an argument, and this is discussed in §4.6.1.)

¹ Documentation comments have the form `/// . . .`, and contain special tags as well as text. These tags and the program components they precede may be recognized by some tool (possibly the compiler), which uses them to produce program documentation as a series of XML files. V2 allows an alternate form of documentation comments, delimited by `/**` and `*/`.

² Although an implementation might permit the `public` modifier to be omitted from `Main`, its presence is guaranteed to work with all implementations, so all examples of `Main` in this book include it.

³ A method uses parameters to declare what it is expecting to be passed, via *arguments*, at runtime.

The body of a method is enclosed within a pair of matching braces. All executable code resides within the body of some method or other. Statements are executed in sequential order unless branching or looping statements dictate otherwise. A program terminates when it returns from `Main`, either by dropping into the closing brace of that method—which acts as an implicit return statement—or via an explicit return statement, as shown in case 2 above. An alternate way of terminating a program involves calling `Environment.Exit`.

As we can see, `Main` is defined inside a brace-delimited block preceded by `class class-name`, where the name of the *class* is `Ba01`, a somewhat arbitrary choice.¹ Note that the class is declared as `public`. While this is permitted, it is not necessary. (We'll discuss this further in §3.9.)

Every method must be defined inside some class or another. While a class has numerous characteristics, the one of importance here is namespace control. The name `Main` is really qualified by its parent class, in this case `Ba01`.

The call to method `WriteLine` in case 1 above, writes a line of text to the standard output device, automatically adding a newline. As we can see, a *string literal* is delimited by double-quote characters. Note that we cannot call `WriteLine` using that simple name; instead, we must specify that this method belongs to a class called `Console`, which, in turn, is contained in a namespace called `System`. We separate each of these names with the dot (`.`) member selection operator. (We'll discuss namespaces in §11.)

Many library classes (and therefore their methods) are contained in the namespace `System`. To simplify library method calls, we can set a default namespace via the `using` keyword, as follows (see directory `Ba02`):

```
using System;

public class Ba02
{
    public static void Main()
    {
        Console.WriteLine("Welcome to C#");
    }
}
```

Now namespace `System` will be searched to resolve any otherwise unrecognized method calls.

Note the use of indenting and blank lines; these are intended to make the code more readable.

Style Tip: Be overt; pick a style that isn't too far outside the mainstream, and stick with it. Above all, be consistent. Remember, the style you develop when learning a language is the one you will likely stay with, so give some thought to programming style from the outset.

The style used in this book is very common. If you choose to adopt a different style, note that no matter how great you think your particular style is, every minute you spend arguing about whose style is superior and why—

¹ Actually, this name was picked because this example is the first in the chapter entitled "Basics".

4. References, Strings, and Arrays

In this chapter, we will learn about reference variables, the allocation of memory at runtime using `new`, garbage collection, array allocation and manipulation, and the class `String`.

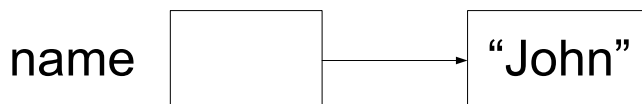
4.1 Introduction

There are two categories of types in C#: simple and reference. We have already seen examples of the simple types, `bool`, `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `decimal`, `char`, `float`, `double`, and `enum` types.

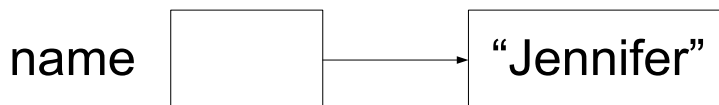
In the definition `int i;`, `i` is a *variable* and the name `i` designates some specific storage location in memory. We say that the value of `i` is some `int`. Throughout its life, that variable is always associated with the same location. (This may seem obvious based on your experience with other languages, but it's most important to understand when we look at reference types.) To help us understand reference types, let's look at the following example (see directory `Rf01`):

```
using System;

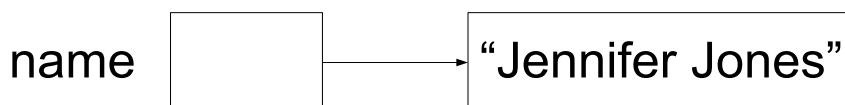
public class Rf01
{
    public static void Main()
    {
        /*1*/      string name = "John";
        /*2*/      Console.WriteLine("name is >{0}<", name);
```



```
/*3*/      name = "Jennifer";
        Console.WriteLine("name is >{0}<", name);
```



```
/*4*/      name = "Jennifer" + " Jones";
        Console.WriteLine("name is >{0}<", name);
```



```

/*5*/     name = null;
/*6*/     Console.WriteLine("name is >{0}<", name);
        }
    }

```

name

null

The output produced is:

```

name is >John<
name is >Jennifer<
name is >Jennifer Jones<
name is ><

```

The keyword `string` is nothing more than an alias for the predefined library class `String`. Writing the keyword `string` is the same as writing `System.String`, and vice versa. And a string literal refers to an object of this type. An object type is often called a *class type*. Unlike some languages, a string is not an array of byte nor is it an array of char; it is a sequence of Unicode characters. **The contents of an object of type `String` cannot be modified.**¹

In case 1, we define a variable called `name`. At first glance, we might think that `name` is a variable of type `string` and that it contains the value "John" directly; however, that is not the case. Instead, `name` is a reference to that string; that is, `name` does not actually contain that data, but simply points to that data, which resides elsewhere. (In the case of a string literal, the data representing that string really has no name.) We say that `name` is a *reference variable* and that its type is "reference to string".

In case 3, `name` is made to refer to a different string. The important thing to understand here is that `name` "has been made to point" to a different string; the value of the string "Jennifer" has *not* been copied anywhere.

In case 4, the two strings are concatenated resulting in a third string to which `name` is made to point. Because a string's value is read-only, the second string cannot simply be appended to the end of the first one.

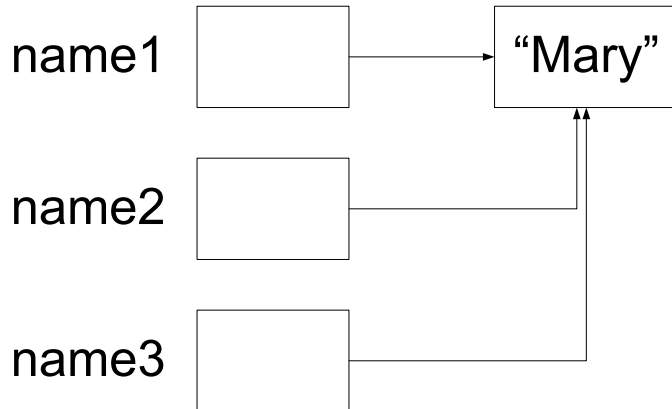
A simple variable can only ever contain a value of its type. However, a reference variable can hold either a reference to an object that is assignment compatible with the type of that variable, or the *null reference*. Case 5 introduces the name `null`. While this looks like a keyword, it really is the null literal; in any event, the word `null` is reserved. By initializing a reference with the value `null`, we make it point "nowhere". The output produced by case 6 shows us that when the value of a reference containing `null` is output, that null reference is turned into an empty string.

There can be any number of references to the same object. For example (see directory Rf02):

¹ The library class `StringBuilder` is similar to `String` except that objects of type `StringBuilder` can have their contents changed. See §4.8.

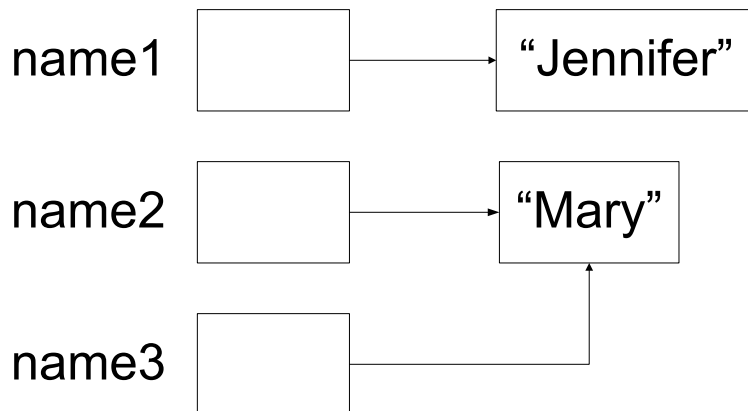
```
using System;

public class Rf02
{
    public static void Main()
    {
        string name1 = "Mary";
        /*1*/   string name2 = name1;
        /*2*/   string name3 = name2;
    }
}
```



```
    Console.WriteLine("name1 is {0}", name1);
    Console.WriteLine("name2 is {0}", name2);
    Console.WriteLine("name3 is {0}", name3);

    /*3*/   name1 = "Jennifer";
}
```



```
    Console.WriteLine("\name1 is {0}", name1);
    Console.WriteLine("name2 is {0}", name2);
    Console.WriteLine("name3 is {0}", name3);
}
}
```

The output produced is:

5. Classes

Classes are the key foundation stone of object-oriented programming. By using classes, we can take advantage of encapsulation, data hiding, inheritance, and polymorphism, the first two of which are discussed in this chapter.

5.1 Introduction

Until now, we have used the keyword `class` primarily as a means for namespace control; however, now we'll see how to exploit its full potential by showing how we can define our own object types.

So far, we've been creating variables of simple types, arrays of those types, and strings. And while we can do useful programming with these types, eventually we'll need more complex and sophisticated data types. For example, in a personnel system, we might want an `Employee` type, containing employee name, address, date of birth, veteran status, and other descriptive information. In a library catalog system, we'll probably want to keep track of each book's call number, author, title, year of publication, and the like, in some kind of a `Publication` type. Neither of these types can be accommodated directly by the types we have seen.

Throughout this and future sections, we'll use `Point` as our user-defined type. A `Point` represents a location in a two-dimensional plane. This simple type is something with which we can easily relate and it serves nicely to introduce the class support machinery. Even if you don't write graphics programs, it should be very easy to extrapolate from the principles learned here.

Let's define our new `Point` type (see directory `Point` in directory `CI01`):

```
public class Point
{
// instance variables

/*1*/  public int x;
/*2*/  public int y;
}
```

The first thing to notice is that in cases 1 and 2, the keyword `static` has been omitted. Back in §1.10, we learned that variables defined inside a class (rather than inside a method) and having the modifier `static`, are class variables; that is, they are variables belonging to the class as a whole. What we now have is a pair of *instance variables*, called `x` and `y`. (Together, class and instance variables are known as *fields*.) We can create an arbitrary number of `Point` objects each of which contains an `x`- and `y`-coordinate pair; that is, each *instance* of class `Point` contains a unique pair of instance variables, since each `Point`'s representation is separate from those of all other `Points`, even for `Points` that happen to have the same coordinate values.

The following program (see directory `Cs_test` in directory `CI01`) creates and manipulates several `Points`:

```

using System;

public class Cl01
{
    public static void Main()
    {
        /*3*/      Point p1 = new Point();
        /*4*/      Console.WriteLine("p1 = ({0},{1})", p1.x, p1.y);

        /*5*/      p1.x = 5;
        /*6*/      p1.y = 7;
        Console.WriteLine("p1 = ({0},{1})", p1.x, p1.y);

        /*7*/      Point p2 = new Point();

        /*8*/      p2.x += -4;
        /*9*/      p2.y += 12;
        Console.WriteLine("p2 = ({0},{1})", p2.x, p2.y);
    }
}

```

In case 3, we allocate memory for a `Point` using `new`, just like we did for strings in §4.1. Creating an instance of a class is known as *instantiation*. By default, all instance variables take on a zero, false, or null value, depending on their type. Like strings and `Points`, all objects of user-defined class types must be allocated on the heap using `new`. To access the instance variables, we simply prefix their names with the name of their parent, as in case 4. Note that this is different to the way in which we have been accessing class variables, which use the parent class name as their prefix. A class can have both class and instance variables, as we'll see later; in fact, many of the library classes do.

It is useful to be able to change the coordinates of a `Point` after it has been created. This operation is often referred to as *moving* the `Point`, and is done in cases 5 and 6. In case 7, we define a second `Point` and we *translate* that `Point` in cases 8 and 9. (Translation involves moving by an offset rather than to an absolute new location.)

The output produced by this program is:

```

p1 = (0,0)
p1 = (5,7)
p2 = (-4,12)

```

When we have a general-purpose class such as `Point`, we very quickly realize that it makes no sense to define our application program as a method within that class. After all, we likely will write many programs that use this class and we can't define all our programs inside that class. In any event, we can only have one method called `Main` having some given signature. As a result, not only do we need to define our application and `Point` in different classes, these classes need to be in separate source files and assemblies.

When defining user-defined types, it is customary to precede the keyword `class` with the keyword `public`. While this is permitted, it is not always necessary. We'll discuss this further in §3.9.

5.2 Class-Specific Methods

It is tedious to read and write the steps to move, translate, and display a `Point` each time. Also, every application relies on the fact that a `Point` contains an `x`- and a `y`-coordinate of type `int`, called `x` and `y`, respectively.

Therefore, any change in the way that type is represented will negatively affect those applications. By making the instance variables private and adding some public methods to class `Point`, we can deal with it in a more abstract manner. Now the `Point` class looks like the following (See directory `Point` in directory `Cl02`):

```
public class Point
{
// instance variables

    private int x;
    private int y;
```

By making the instance variables `private`, they can only be accessed by methods within their parent class. This is known as *data hiding*, since we hide the representation details of `Point` from all programs that use it. The main reason for data hiding is to cater for unanticipated changes. Since we can never say the representation of an object won't change and we can never say exactly how it might or will change, we should cater for as much flexibility as possible. That is, assume everything within reason will change.

```
// constructors

    public Point(int xor, int yor)
    {
        x = xor;
        y = yor;
    }

    public Point()
    {
        x = 0;
        y = 0;
    }
```

In the program above, we also see an example of *encapsulation*, the process by which we associate variables and methods by defining them in the same class. This allows us to control the access to those variables and methods.

The methods called `Point` are special. Strictly speaking, they are not really methods, but rather, constructors; however, for all practical purposes, they look like methods—they just have a special name. From this, we can deduce that, syntactically, a *constructor* is nothing more than a method that has the same name as its parent class. The first constructor expects two `int` arguments, which represent the `x`- and `y`-coordinates, respectively, and it initializes the private fields using those values. A constructor cannot have a return type declared, not even `void`. It can contain one or more `return` statements, however, provided they do not contain an expression.