LEARNING MADE EASY

C# 7.0

ALL-IN-ONE

for dummies®
A Wiley Brand

6 Books in one!

John Paul Mueller
Bill Sempf
Chuck Sphar

C# 7.0

ALL-IN-ONE

C# 7.0

ALL-IN-ONE

by John Paul Mueller,
Bill Sempf, and Chuck Sphar

for dummies®
A Wiley Brand

C# 7.0 All-in-One For Dummies®

Published by: **John Wiley & Sons, Inc.,** 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

Published simultaneously in Canada

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For technical support, please visit https://hub.wiley.com/community/support/dummies.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2017958295

ISBN: 978-111-9-42811-4; ISBN 978-111-9-42810-7 (ebk); ISBN ePDF 978-111-9-42812-1 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents at a Glance

Table of Contents

Introduction

C# is an amazing language! You can use this single language to do everything from desktop development to creating web applications and even web-based application programming interfaces (APIs). While other developers have to overcome deficiencies in their languages to create even a subset of the application types that C# supports with aplomb, you can be coding your application, testing, and then sitting on the beach enjoying the fruits of your efforts. Of course, any language that does this much requires a bit of explanation, and *C# 7.0 All-in-One For Dummies* is your doorway to this new adventure in development.

So, why do you need *C# 7.0 All-in-One For Dummies* specifically? This book stresses learning the basics of the C# language before you do anything else. With this in mind, the book begins with all the C# basics in Books 1 through 3, helps you get Visual Studio 2017 installed in Book 4, and then takes you through more advanced development tasks, including basic web development, in Books 5 through 6. Using this book helps you get the most you can from C# 7.0 in the least possible time.

About This Book

Even if you have past experience with C#, the new features in C# 7.0 will have you producing feature-rich applications in an even shorter time than you may have before. *C# 7.0 All-in-One For Dummies* introduces you to all these new features. For example, you discover the new pattern-matching techniques that C# 7.0 provides. You also discover the wonders of using tuples and local functions. Even the use of literals has improved, but you'll have to look inside to find out how. This particular book is designed to make using C# 7.0 fast and easy; it removes the complexity that you may have experienced when trying to learn about these topics online.

To help you absorb the concepts, this book uses the following conventions:

- Text that you're meant to type just as it appears in the book is in **bold**. The exception is when you're working through a step list: Because each step is bold, the text to type is not bold.

- Words for you to type that are also in *italics* are meant as placeholders; you need to replace them with something that works for you. For example, if you see "Type ***Your Name*** and press Enter," you need to replace *Your Name* with your actual name.
- I also use *italics* for terms I define. This means that you don't have to rely on other sources to provide the definitions you need.
- Web addresses and programming code appear in `monofont`. If you're reading a digital version of this book on a device connected to the Internet, you can click the live link to visit a website, like this: `www.dummies.com`.
- When you need to click command sequences, you see them separated by a special arrow, like this: File➪New File, which tells you to click File and then click New File.

Foolish Assumptions

You might have a hard time believing that I've assumed anything about you — after all, I haven't even met you yet! Although most assumptions are indeed foolish, I made certain assumptions to provide a starting point for the book.

The most important assumption is that you know how to use Windows, have a copy of Windows properly installed, and are familiar with using Windows applications. If installing an application is still a mystery to you, you might find this book a bit hard to use. While reading this book, you need to install applications, discover how to use them, and create simple applications of your own.

You also need to know how to work with the Internet to some degree. Many of the materials, including the downloadable source, appear online, and you need to download them in order to get the maximum value from the book. In addition, Book 6 assumes that you have a certain knowledge of the Internet when working through web-based applications and web-based services.

Icons Used in This Book

As you read this book, you encounter icons in the margins that indicate material of special interest (or not, as the case may be!). Here's what the icons mean:


TIP

Tips are nice because they help you save time or perform some task without a lot of extra work. The tips in this book are timesaving techniques or pointers to resources that you should try so that you can get the maximum benefit when performing C#-related tasks.


WARNING

I don't want to sound like an angry parent or some kind of maniac, but you should avoid doing anything that's marked with a Warning icon. Otherwise, you might find that your configuration fails to work as expected, you get incorrect results from seemingly bulletproof processes, or (in the worst-case scenario) you lose data.


TECHNICAL STUFF

Whenever you see this icon, think advanced tip or technique. You might find these tidbits of useful information just too boring for words, or they could contain the solution you need to get a C# application running. Skip these bits of information whenever you like.


REMEMBER

If you don't get anything else out of a particular chapter or section, remember the material marked by this icon. This text usually contains an essential process or a bit of information that you must know to work with C#.

Beyond the Book

This book isn't the end of your C# learning experience — it's really just the beginning. John Mueller provides online content to make this book more flexible and better able to meet your needs. Also, you can send John email. He'll address your book-specific questions and tell you how updates to C# or its associated add-ons affect book content through blog posts. Here are some cool online additions to this book:

- **Cheat sheet:** You remember using crib notes in school to make a better mark on a test, don't you? You do? Well, a cheat sheet is sort of like that. It provides you with some special notes about tasks that you can do with C# that not every other person knows. To find the cheat sheet for this book, go to `www.dummies.com` and search for *C# 7.0 All-in-One For Dummies Cheat Sheet.* It contains really neat information such as how to figure out which template you want to use.
- **Updates:** Sometimes changes happen. For example, I might not have seen an upcoming change when I looked into my crystal ball during the writing of this book. In the past, this possibility simply meant that the book became outdated and less useful, but you can now find updates to the book at `www.dummies.com`.

In addition to these updates, check out the blog posts with answers to reader questions and demonstrations of useful book-related techniques at http://blog.johnmuellerbooks.com/.

» **Companion files:** Hey! Who really wants to type all the code in the book manually? Most readers prefer to spend their time actually working with C#, creating amazing new applications that change the world, and seeing the interesting things they can do, rather than typing. Fortunately for you, the examples used in the book are available for download, so all you need to do is read the book to learn C# development techniques. You can find these files at www.dummies.com. You can also download Online Chapters 1–7. To find the source code and online chapters, search this book's title at www.dummies.com and locate the Downloads tab on the page that appears.

Where to Go from Here

Anyone who is unfamiliar with C# should start with Book 1, Chapter 1 and move from there to the end of the book. This book is designed to make it easy for you to discover the benefits of using C# from the outset. Later, after you've seen enough C# code, you can install Visual Studio and then try the programming examples found in the first three minibooks.

This book assumes that you want to see C# code from the outset. However, if you want to interact with that code, you really need to have a copy of Visual Studio 2017 installed. (Some examples will not work at all with older Visual Studio versions.) With this in mind, you may want to skip right to Book 4 to discover how to get your own copy of Visual Studio 2017. To help ensure that everyone can participate, this book focuses on the features offered by Visual Studio 2017 Community Edition, which is a free download. That's right, you can discover the wonders of C# 7.0 without paying a dime!

The more you know about C#, the further you can start in the book. If all you're really interested in is an update of your existing skills, check out Book 1, Chapter 1 to discover the changes in C#. Then, scan the first three minibooks looking for points of interest. Install C# by using the instructions in Book 4, Chapter 1, and then move on toward the advanced techniques found in later chapters.

1 The Basics of C# Programming

Contents at a Glance

IN THIS CHAPTER

- Getting a quick introduction to programming
- Creating a simple console application
- Examining the console application
- Saving code for later

Chapter 1

Creating Your First C# Console Application

This chapter explains a little bit about computers, computer languages — including the computer language C# (pronounced "see sharp") — and Visual Studio 2017. You then create a simple program written in C#.

Getting a Handle on Computer Languages, C#, and .NET

A computer is an amazingly fast but incredibly stupid servant. Computers will do anything you ask them to (within reason); they do it extremely fast — and they're getting faster all the time.

Unfortunately, computers don't understand anything that resembles a human language. Oh, you may come back at me and say something like, "Hey, my telephone lets me dial my friend by just speaking his name." Yes, a tiny computer runs your telephone. So that computer speaks English. But that's a computer *program* that understands English, not the computer itself.

The language that computers truly understand is *machine language.* It's possible, but extremely difficult and error prone, for humans to write machine language.

Humans and computers have decided to meet somewhere in the middle. Programmers create programs in a language that isn't nearly as free as human speech, but it's a lot more flexible and easier to use than machine language. The languages occupying this middle ground — C#, for example — are *high-level* computer languages. (*High* is a relative term here.)

What's a program?

What is a program? In a practical sense, a Windows program is an executable file that you can run by double-clicking its icon. For example, Microsoft Word, the editor used to write this book, is a program. You call that an *executable program,* or *executable* for short. The names of executable program files generally end with the extension `.exe`. Word, for example, is `Winword.exe`.

But a program is something else as well. An executable program consists of one or more *source files*. A C# *source file,* for instance, is a text file that contains a sequence of C# commands, which fit together according to the laws of C# grammar. This file is known as a *source file,* probably because it's a source of frustration and anxiety.

Uh, grammar? There's going to be grammar? Just the C# kind, which is much easier than the kind most people struggled with in junior high school.

What's C#?

The C# programming language is one of those intermediate languages that programmers use to create executable programs. C# combines the range of the powerful but complicated C++ (pronounced "see plus plus") with the ease of use of the friendly but more verbose Visual Basic. (Visual Basic's newer .NET incarnation is almost on par with C# in most respects. As the flagship language of .NET, C# tends to introduce most new features first.) A C# program file carries the extension `.cs`.

Some people have pointed out that C sharp and D flat are the same note, but you shouldn't refer to this new language as "D flat" within earshot of Redmond, Washington.

C# is

- **Flexible:** C# programs can execute on the current machine, or they can be transmitted over the web and executed on some distant computer.
- **Powerful:** C# has essentially the same command set as C++ but with the rough edges filed smooth.
- **Easier to use:** C# error-proofs the commands responsible for most C++ errors, so you spend far less time chasing down those errors.
- **Visually oriented:** The .NET code library that C# uses for many of its capabilities provides the help needed to readily create complicated display frames with drop-down lists, tabbed windows, grouped buttons, scroll bars, and background images, to name just a few.


TIP

.NET is pronounced "dot net."

- **Internet-friendly:** C# plays a pivotal role in the .NET Framework, Microsoft's current approach to programming for Windows, the Internet, and beyond.
- **Secure:** Any language intended for use on the Internet must include serious security to protect against malevolent hackers.

Finally, C# is an integral part of .NET.


REMEMBER

This book is primarily about the C# language. If your primary goal is to use Visual Studio, program Windows 8 or 10 apps, or ASP.NET, the *For Dummies* books on those topics go well with this book. You can find a good amount of information later in this book on how to use C# to write Windows, web, and service applications.

What's .NET?

.NET began several years ago as Microsoft's strategy to open the web to mere mortals like you and me. Today, it's bigger than that, encompassing everything Microsoft does. In particular, it's the new way to program for Windows. It also gives a C-based language, C#, the simple, visual tools that made Visual Basic so popular.

A little background helps you see the roots of C# and .NET. Internet programming was traditionally very difficult in older languages such as C and C++. Sun Microsystems responded to that problem by creating the Java programming language. To create Java, Sun took the grammar of C++, made it a lot more user friendly, and centered it around distributed development.


REMEMBER

When programmers say "distributed," they're describing geographically dispersed computers running programs that talk to each other — via the Internet in many cases.

When Microsoft licensed Java some years ago, it ran into legal difficulties with Sun over changes it wanted to make to the language. As a result, Microsoft more or less gave up on Java and started looking for ways to compete with it.

Being forced out of Java was just as well because Java has a serious problem: Although Java is a capable language, you pretty much have to write your entire program *in* Java to get the full benefit. Microsoft had too many developers and too many millions of lines of existing source code, so Microsoft had to come up with some way to support multiple languages. Enter .NET.

.NET is a framework, in many ways similar to Java's libraries — and the C# language is highly similar to the Java language. Just as Java is both the language itself and its extensive code library, C# is really much more than just the keywords and syntax of the C# language. It's those things empowered by a well-organized library containing thousands of code elements that simplify doing about any kind of programming you can imagine, from web-based databases to cryptography to the humble Windows dialog box.

Microsoft would claim that .NET is much superior to Sun's suite of web tools based on Java, but that's not the point. Unlike Java, .NET doesn't require you to rewrite existing programs. A Visual Basic programmer can add just a few lines to make an existing program *web-knowledgeable* (meaning that it knows how to get data off the Internet). .NET supports all the common Microsoft languages — and hundreds of other languages written by third-party vendors. However, C# is the flagship language of the .NET fleet. C# is always the first language to access every new feature of .NET.

What is Visual Studio 2017? What about Visual C#?

(You sure ask lots of questions.) The first "Visual" language from Microsoft was Visual Basic. The first popular C-based language from Microsoft was Visual C++. Like Visual Basic, it had Visual in its name because it had a built-in graphical user interface (GUI — pronounced "GOO-ee"). This GUI included everything you needed to develop nifty-gifty C++ programs.

Eventually, Microsoft rolled all its languages into a single environment — Visual Studio. As Visual Studio 6.0 started getting a little long in the tooth, developers anxiously awaited version 7. Shortly before its release, however, Microsoft decided to rename it Visual Studio .NET to highlight this new environment's relationship to .NET.

That sounded like a marketing ploy to a lot of people — until they started delving into it. Visual Studio .NET differed quite a bit from its predecessors — enough to

warrant a new name. Visual Studio 2017 is the ninth-generation successor to the original Visual Studio .NET. (Book 4 is full of Visual Studio goodness, including instructions for customizing it. You may want to use the instructions in Book 4, Chapter 1 to install a copy of Visual Studio before you get to the example later in this chapter. If you're completely unfamiliar with Visual Studio, then reviewing all of Book 4 is helpful.)


REMEMBER

Microsoft calls its implementation of the language Visual C#. In reality, Visual C# is nothing more than the C# component of Visual Studio. C# is C#, with or without Visual Studio. Theoretically, you could write C# programs by using any text editor and a few special tools, but using Visual Studio is so much easier that you wouldn't want to try.

Okay, that's it. No more questions. (For now, anyway.)

Creating Your First Console Application

Visual Studio 2017 includes an Application Wizard that builds template programs and saves you a lot of the dirty work you'd have to do if you did everything from scratch. (The from-scratch approach is error prone, to say the least.)

Typically, starter programs don't really do anything — at least, not anything useful. However, they do get you beyond that initial hurdle of getting started. Some starter programs are reasonably sophisticated. In fact, you'll be amazed at how much capability the App Wizard can build on its own, especially for graphical programs.

This starter program isn't even a graphical program, though. A *console* application is one that runs in the "console" within Windows, usually referred to as the DOS prompt or command window. If you press Ctrl+R and then type `cmd`, you see a command window. It's the console where the application will run.


REMEMBER

The following instructions are for Visual Studio. If you use anything other than Visual Studio, you have to refer to the documentation that came with your environment. Alternatively, you can just type the source code directly into your C# environment.

Creating the source program

To start Visual Studio, press the Windows button on your keyboard and type **Visual Studio**. Visual Studio 2017 appears as one of the available options. You can access

the example code for this chapter in the \CSAIO4D\BK01\CH01 folder in the downloadable source, as explained in the Introduction.

Complete these steps to create your C# console app:

1. **Open Visual Studio 2017 and click the Create New Project link, shown in Figure 1-1.**

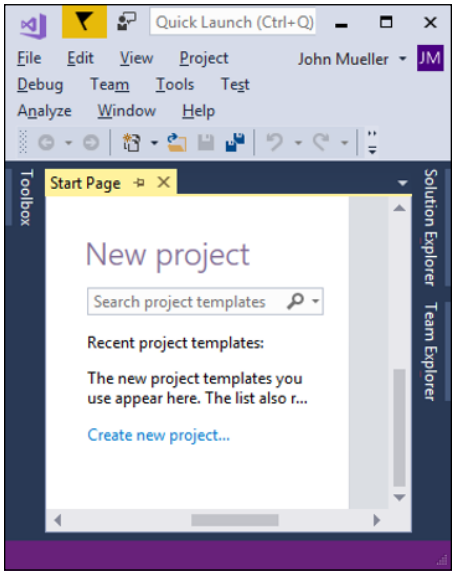

FIGURE 1-1: Creating a new project starts you down the road to a better Windows application.

Visual Studio presents you with lots of icons representing the different types of applications you can create, as shown in Figure 1-2.

2. **In this New Project window, click the Console App (.NET Framework) icon.**

WARNING

Make sure that you select Visual C# — and under it, Windows — in the Project Types pane; otherwise Visual Studio may create something awful like a Visual Basic or Visual C++ application. Then click the Console App (.NET Framework) icon in the Templates pane.

REMEMBER

Visual Studio requires you to create a project before you can start entering your C# program. A *project* is a folder into which you throw all the files that go into making your program. It has a set of configuration files that help the compiler do its work. When you tell your compiler to build (*compile*) the program, it sorts through the project to find the files it needs in order to re-create the executable program.

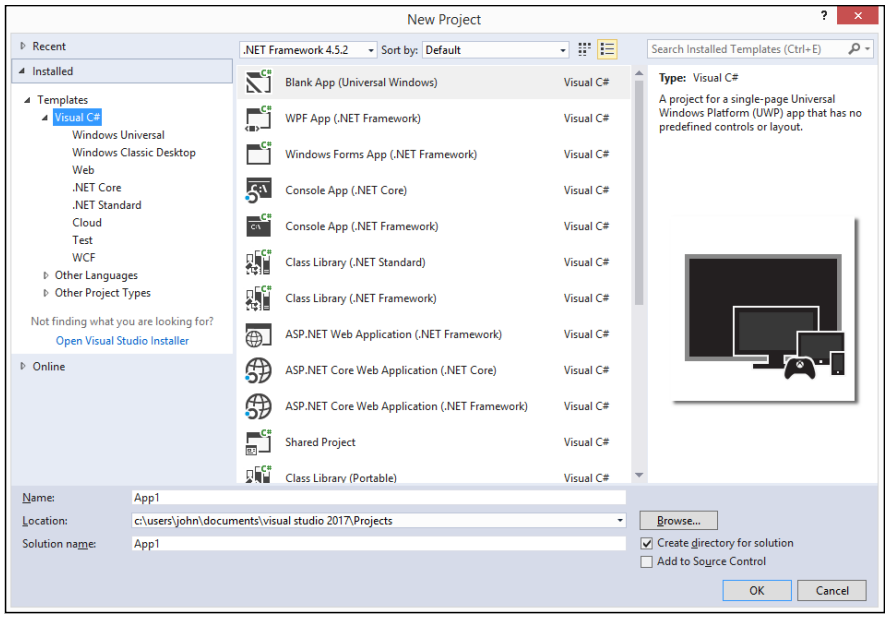

FIGURE 1-2: The Visual Studio App Wizard is eager to create a new program for you.

TECHNICAL STUFF

Visual Studio 2017 provides support for both .NET Framework and .NET Core applications. A .NET Framework application is the same as the C# applications supported in previous versions of Windows; it runs only in Windows and isn't open source. A .NET Core application can run in Windows, Linux, and Mac environments and relies on an open source setup. Although using .NET Core may seem ideal, the .NET Core applications also support only a subset of the .NET Framework features, and you can't add a GUI to them. Microsoft created the .NET Core for these uses:

- Cross platform development
- Microservices
- Docker containers
- High performance and scalable applications
- Side-by-side .NET application support

3. **The default name for your first application is `App1`, but change it this time to** Program1 **by typing in the Name field.**

TIP

The default place to store this file is somewhere deep in your `Documents` directory. For most developers, it's a lot better to place the files where you can actually find them and interact with them as needed, not necessarily where Visual Studio wants them.

4. **Type** C:\CSAIO4D\BK01\CH01 **Location field to change the location of this project.**

5. **Click the OK button.**

   After a bit of disk whirring and chattering, Visual Studio generates a file named `Program.cs`. (If you look in the window labeled Solution Explorer, shown in Figure 1-3, you see some other files; ignore them for now. If Solution Explorer isn't visible, choose View➪Solution Explorer.)

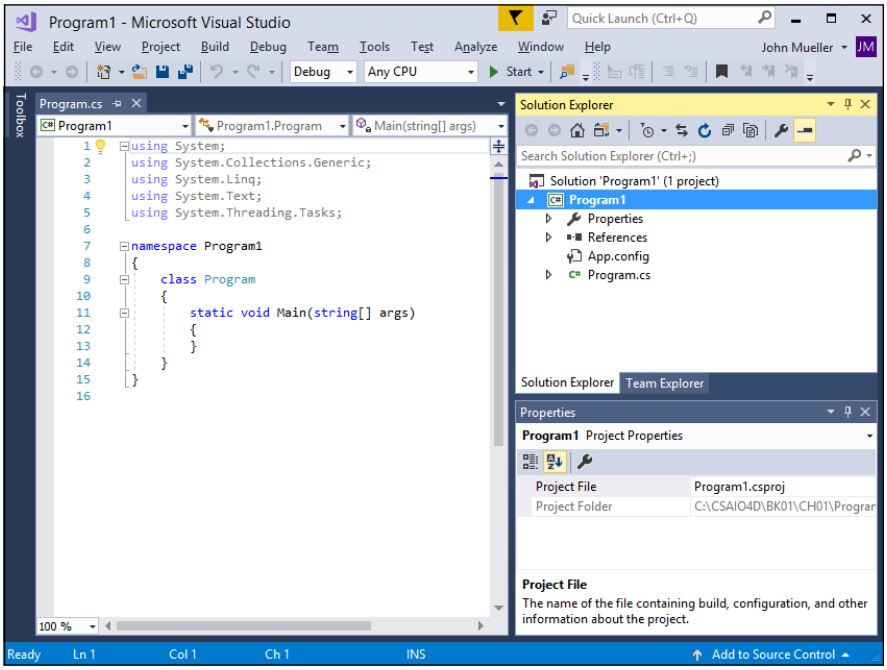

FIGURE 1-3: Visual Studio displays the project you just created.

C# source files carry the extension `.cs`. The name `Program` is the default name assigned for the program file.

The contents of your first console app appear this way (as shown in Figure 1-3):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Program1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

TIP

You can manually change the location of the project with every project. However, you have a simpler way to go. When working with this book, you can change the default program location. To make that happen, follow these steps after you finish creating the project:

1. **Choose Tools➪Options.**

   The Options dialog box opens. You may have to select the Show All Options box.

2. **Choose Projects and Solutions➪General.**

3. **Select the new location in the Projects Location field and click OK.**

   (The examples assume that you have used `C:\CSAIO4D` for this book.)

You can see the Options dialog box in Figure 1-4. Leave the other fields in the project settings alone for now. Read more about customizing Visual Studio in Book 4 and in Online Chapter 2, which you find by going to `www.dummies.com`, searching this book's title, and locating the Downloads tab on the page that appears.

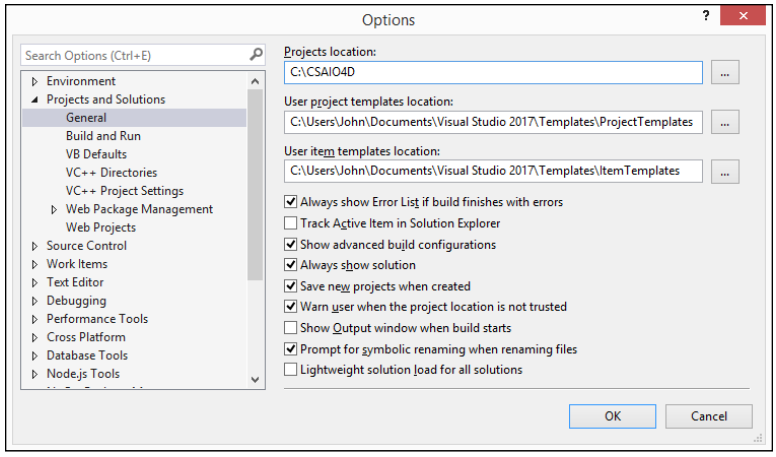

FIGURE 1-4: Changing the default project location.

REMEMBER

Along the left edge of the code window, you see several small plus (+) and minus (–) signs in boxes. Click the + sign next to `using ...`. This expands a *code region*, a handy Visual Studio feature that minimizes clutter. Here are the directives that appear when you expand the region in the default console app:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

Regions help you focus on the code you're working on by hiding code that you aren't. Certain blocks of code — such as the `namespace` block, `class` block, methods, and other code items — get a +/– automatically without a `#region` directive. You can add your own collapsible regions, if you like, by typing `#region` above a code section and `#endregion` after it. It helps to supply a name for the region, such as `Public methods`. This code section looks like this:

```
#region Public methods
... your code
#endregion Public methods
```

REMEMBER

This name can include spaces. Also, you can nest one region inside another, but regions can't overlap.

For now, `using System;` is the only `using` directive you really need. You can delete the others; the compiler lets you know whether you're missing one.

Taking it out for a test drive

Before you try to create your application, open the Output window (if it isn't already open) by choosing View➪Output. To convert your C# program into an executable program, choose Build➪Build Program1. Visual Studio responds with the following message:

```
------ Build started: Project: Program1, Configuration: Debug Any CPU ----
Program1 -> C:\CSAIO4D\BK01\CH01\Program1\Program1\bin\Debug\Program1.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =========
```

The key point here is the `1 succeeded` part on the last line.

TIP

As a general rule of programming, `succeeded` is good; `failed` is bad. The bad — the exceptions — is covered in Chapter 9 of this minibook.

To execute the program, choose Debug➪Start. The program brings up a black console window and terminates immediately. (If you have a fast computer, the appearance of this window is just a flash on the screen.) The program has seemingly done nothing. In fact, this is the case. The template is nothing but an empty shell.


TIP

An alternative command, Debug➪Start Without Debugging, behaves a bit better at this point. Try it out.

Making Your Console App Do Something

Edit the `Program.cs` template file until it appears this way:

```
using System;

namespace Program1
{
  public class Program
  {
    // This is where your program starts.
    static void Main(string[] args)
    {
      // Prompt user to enter a name.
      Console.WriteLine("Enter your name, please:");

      // Now read the name entered.
      string name = Console.ReadLine();

      // Greet the user with the name that was entered.
      Console.WriteLine("Hello, " + name);

      // Wait for user to acknowledge the results.
      Console.WriteLine("Press Enter to terminate...");
      Console.Read();
    }
  }
}
```


TIP

Don't sweat the stuff following the double or triple slashes (`//` or `///`) and don't worry about whether to enter one or two spaces or one or two new lines. However, do pay attention to capitalization.

Choose Build➪Build Program1 to convert this new version of `Program.cs` into the `Program1.exe` program.

From within Visual Studio 2017, choose Debug➪Start Without Debugging. The black console window appears and prompts you for your name. (You may need to activate the console window by clicking it.) Then the window shows `Hello`, followed by the name entered, and displays `Press Enter to terminate ...`. Pressing Enter closes the window.

TECHNICAL STUFF

You can also execute the program from the DOS command line. To do so, open a Command Prompt window and enter the following:

```
CD \C#Programs\Program1\bin\Debug
```

Now enter **Program1** to execute the program. The output should be identical to what you saw earlier. You can also navigate to the `\C#Programs\Program1\bin\Debug` folder in Windows Explorer and then double-click the `Program1.exe` file.

TIP

To open a Command Prompt window, try choosing Tools➪Command Prompt. If that command isn't available on your Visual Studio Tools menu, open a copy of Windows Explorer, locate the folder containing the executable as shown in Figure 1-5, and then choose File➪Open Command Prompt. You see a command prompt where you can execute the program.

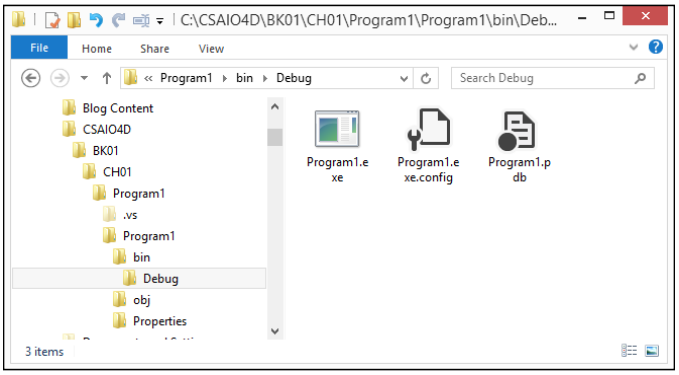

FIGURE 1-5: Windows Explorer provides a quick way to open a command prompt.

Reviewing Your Console Application

In the following sections, you take this first C# console app apart one section at a time to understand how it works.

The program framework

The basic framework for all console applications starts as the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Program1
{
  public class Program
  {
    // This is where your program starts.
    public static void Main(string[] args)
    {
       // Your code goes here.
    }
  }
}
```

The program starts executing right after the statement containing `Main()` and ends at the closed curly brace (}) following `Main()`. (You find the explanation for these statements in due course. Just know that they work as they should for now.)


REMEMBER

The list of `using` directives can come immediately before or immediately after the phrase `namespace Program1 {`. The order doesn't matter. You can apply `using` to lots of things in .NET. You find an explanation for namespaces and `using` in the object-oriented programming chapters in Book 2.

Comments

The template already has lots of lines, and the example code adds several other lines, such as the following (in boldface):

```
// This is where your program starts.
public static void Main(string[] args)
```

C# ignores the first line in this example. This line is known as a *comment.*


TIP

Any line that begins with `//` or `///` is free text, and C# ignores it. Consider `//` and `///` to be equivalent for now.

Why include lines if the computer ignores them? Because comments explain your C# statements. A program, even in C#, isn't easy to understand. Remember that a programming language is a compromise between what computers understand and what humans understand. These comments are useful while you write the

code, and they're especially helpful to the poor sap — possibly you — who tries to re-create your logic a year later. Comments make the job much easier.

TIP

Comment early and often.

The meat of the program

The real core of this program is embedded within the block of code marked with `Main()`, like this:

```
// Prompt user to enter a name.
Console.WriteLine("Enter your name, please:");

// Now read the name entered.
string name = Console.ReadLine();

// Greet the user with the name that was entered.
Console.WriteLine("Hello, " + name);
```

TIP

Save a ton of routine typing with the C# Code Snippets feature. Snippets are great for common statements like `Console.WriteLine`. Press Ctrl+K,X to see a pop-up menu of snippets. (You may need to press Tab once or twice to open the Visual C# folder or other folders on that menu.) Scroll down the menu to `cw` and press Enter. Visual Studio inserts the body of a `Console.WriteLine()` statement with the insertion point between the parentheses, ready to go. When you have a few of the shortcuts, such as `cw`, `for`, and `if`, memorized, use the even quicker technique: Type **cw** and press Tab twice. (Also try selecting some lines of code, pressing Ctrl+K, and then pressing Ctrl+S. Choose something like `if`. An `if` statement *surrounds* the selected code lines.)

The program begins executing with the first C# statement: `Console.WriteLine`. This command writes the character string `Enter your name, please:` to the console.

The next statement reads in the user's answer and stores it in a *variable* (a kind of workbox) named `name`. (See Chapter 2 of this minibook for more on these storage locations.) The last line combines the string `Hello,` with the user's name and outputs the result to the console.

The final three lines cause the computer to wait for the user to press Enter before proceeding. These lines ensure that the user has time to read the output before the program continues:

```
// Wait for user to acknowledge the results.
Console.WriteLine("Press Enter to terminate...");
Console.Read();
```

This step can be important, depending on how you execute the program and depending on the environment. In particular, running your console app inside Visual Studio, or from Windows Explorer, makes the preceding lines necessary — otherwise, the console window closes so fast you can't read the output. If you open a console window and run the program from there, the window stays open regardless.

Introducing the Toolbox Trick

The key part of the program you create in the preceding section consists of the final two lines of code:

```
// Wait for user to acknowledge the results.
Console.WriteLine("Press Enter to terminate...");
Console.Read();
```

The easiest way to re-create those key lines in each future console application you write is described in the following sections.

Saving code in the Toolbox

The first step is to save those lines in a handy location for future use: the Toolbox window. With your `Program1` console application open in Visual Studio, follow these steps:

1. **In the `Main()` method of class `Program`, select the lines you want to save — in this case, the three lines mentioned previously.**
2. **Make sure the Toolbox window is open on the left. (If it isn't, open it by choosing View➪Toolbox.)**
3. **Drag the selected lines into the General tab of the Toolbox window and drop them. (Or copy the lines and paste them into the Toolbox.)**

   The Toolbox stores the lines there for you in perpetuity. Figure 1-6 shows the lines placed in the Toolbox.

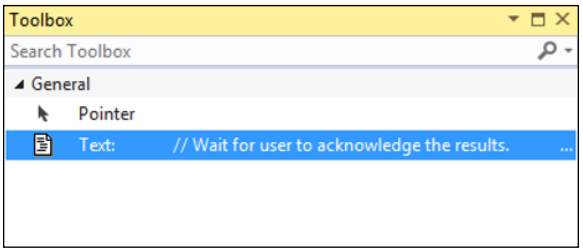

FIGURE 1-6: Setting up the Toolbox with some handy saved text for future use.

Reusing code from the Toolbox

Now that you have your template text stored in the Toolbox, you can reuse it in all console applications you write henceforth. Here's how to use it:

1. **In Visual Studio, create a new console application as described in the section "Creating the source program," earlier in this chapter.**
2. **Click in the editor at the spot where you'd like to insert some Toolbox text.**
3. **With the `Program.cs` file open for editing, make sure the Toolbox window is open.**

   If it isn't, see the procedure in the preceding "Saving code in the Toolbox" section.
4. **In the General tab of the Toolbox window (other tabs may be showing), find the saved text you want to use and double-click it.**

   The selected item is inserted at the insertion point in the editor window.

With that boilerplate text in place, you can write the rest of your application above those lines. That's it. You now have a finished console app. Try it for about 30 seconds. Then you can check out Chapter 2 of this minibook.

IN THIS CHAPTER

- Using C# variables, such as integers, as a storage locker
- Declaring other types of variables — dates, characters, strings
- Handling numeric constants
- Changing types and letting the compiler figure out the type

Chapter 2

Living with Variability — Declaring Value-Type Variables

The most fundamental of all concepts in programming is that of the variable. A C# variable is like a small box in which you can store things, particularly numbers, for later use. (The term *variable* is borrowed from the world of mathematics.)

Unfortunately for programmers, C# places several limitations on variables — limitations that mathematicians don't have to consider. However, these limits are in place for a reason. They make it easier for C# to understand what you mean by a particular kind of variable and for you to find mistakes in your code. This chapter takes you through the steps for declaring, initializing, and using variables. It also introduces several of the most basic data types in C#.

Declaring a Variable

Mathematicians work with numbers in a precise manner, but in a way that C# could never understand. The mathematician is free to introduce the variables as needed to present an idea in a particular way. Mathematicians use algorithms, a set of procedural steps used to solve a problem, in a way that makes sense to other mathematicians to model real-world needs. Algorithms can appear quite complex, even to other humans, much less C#. For example, the mathematician may say this:

```
x = y² + 2y + y
if k = y + 1 then
x = k²
```

Programmers must define variables in a particular way that's more demanding than the mathematician's looser style. A programmer must tell C# the kind of value that a variable contains and then tell C# specifically what to place in that variable in a manner that C# understands. For example, a C# programmer may write the following bit of code:

```
int n;
n = 1;
```

The first line means, "Carve off a small amount of storage in the computer's memory and assign it the name *n*." This step is analogous to reserving one of those storage lockers at the train station and slapping the label *n* on the side. The second line says, "Store the value 1 in the variable `n`, thereby replacing whatever that storage location already contains." The train-locker equivalent is, "Open the train locker, rip out whatever happens to be in there, and shove a 1 in its place."

REMEMBER

The equals symbol (=) is called the *assignment operator.*

TECHNICAL STUFF

The mathematician says, "*n* equals 1." The C# programmer says in a more precise way, "Store the value 1 in the variable `n`." (Think about the train locker, and you see why that's easier for C# to understand.) C# operators, such as the assignment operator, tell the computer what you want to do. In other words, operators are verbs and not descriptors. The assignment operator takes the value on its right and stores it in the variable on the left. You discover more about operators in Chapter 4 of this minibook.

What's an int?

In C#, each variable has a fixed type. When you allocate one of those train lockers, you have to pick the size you need. If you pick an integer locker, for instance, you can't turn around and hope to stuff the entire state of Texas in it — maybe Rhode Island, but not Texas.

For the example in the preceding section of this chapter, you select a locker that's designed to handle an integer — C# calls it an `int`. Integers are the counting numbers 1, 2, 3, and so on, plus 0 and the negative numbers –1, –2, –3, and so on.


REMEMBER

Before you can use a variable, you must *declare* it, which means creating a variable with a specific name (label) using code and optionally assigning a value to that variable. After you declare a variable as `int`, it can hold integer values, as this example demonstrates:

```
// Declare a variable named n – an empty train locker.
int n;
// Declare an int variable m and initialize it with the value 2.
int m = 2;
// Assign the value stored in m to the variable n.
n = m;
```

The first line after the comment is a *declaration* that creates a little storage area, `n`, designed to hold an integer value. The initial value of `n` is not specified until it is *assigned* a value, so this locker is essentially empty. The second declaration not only declares an `int` variable `m` but also *initializes* it with a value of 2, all in one shot.


REMEMBER

The term *initialize* means to assign an initial value. To initialize a variable is to assign it a value for the first time. You don't know for sure what the value of a variable is until it has been initialized. Nobody knows. It's always an error to use a variable before you initialize it.

The final statement in the program assigns the value stored in `m`, which is 2, to the variable `n`. The variable `n` continues to contain the value 2 until it is assigned a new value. (The variable `m` doesn't lose its value when you assign its value to `n`. It's like cloning `m`.)

Rules for declaring variables

You can initialize a variable as part of the declaration, like this:

```
// Declare another int variable and give it the initial value of 1.
int p = 1;
```

This is equivalent to sticking a 1 into that `int` storage locker when you first rent it, rather than opening the locker and stuffing in the value later.


TIP

Initialize a variable when you declare it. In most (but not all) cases, C# initializes the variable for you — but don't rely on it to do that. For example, C# does place a 0 into an uninitialized `int` variable, but the compiler will still display an error if you try to use the variable before you initialize it. You may declare variables anywhere (well, almost anywhere) within a program.


WARNING

However, you may not use a variable until you declare it and set it to some value. Thus the last two assignments shown here are *not* legal:

```
// The following is illegal because m is not assigned
// a value before it is used.
int n
int m;
n = m;
// The following is illegal because p has not been
// declared before it is used.
p = 2;
int p;
```

Finally, you cannot declare the same variable twice in the same scope (a function, for example).

Variations on a theme: Different types of int

Most simple numeric variables are of type `int`. However, C# provides a number of twists to the `int` variable type for special occasions.

All integer variable types are limited to whole numbers. The `int` type suffers from other limitations as well. For example, an `int` variable can store values only in the range from roughly −2 billion to 2 billion.

A distance of 2 billion inches is greater than the circumference of the Earth. In case 2 billion isn't quite large enough for you, C# provides an integer type called `long` (short for `long int`) that can represent numbers almost as large as you can imagine. The only problem with a `long` is that it takes a larger train locker: A `long` consumes 8 bytes (64 bits) — twice as much as a garden-variety 4-byte (32-bit) `int`. C# provides several other integer variable types, as shown in Table 2-1.

TABLE 2-1 Size and Range of C# Integer Types

Type	Bytes	Range of Values	In Use
`sbyte`	1	–128 to 127	`sbyte sb = 12;`
`byte`	1	0 to 255	`byte b = 12;`
`short`	2	–32,768 to 32,767	`short sh = 12345;`
`ushort`	2	0 to 65,535	`ushort ush = 62345;`
`int`	4	–2 billion to 2 billion	`int n = 1234567890;`
`uint`	4	0 to 4 billion (exact values listed in the Cheat Sheet on this book's website)	`uint un = 3234567890U`
`long`	8	-10^{20} to 10^{20} — "a whole lot"	`long l = 123456789012L`
`Ulong`	8	0 to 2×10^{20}	`long ul = 123456789012UL`

As explained in the section entitled "Declaring Numeric Constants," later in this chapter, fixed values such as 1 also have a type. By default, a simple constant such as 1 is assumed to be an `int`. Constants other than an `int` must be marked with their variable type. For example, 123U is an unsigned integer, `uint`.

Most integer variables are called *signed*, which means they can represent negative values. Unsigned integers can represent only positive values, but you get twice the range in return. As you can see from Table 2-1, the names of most unsigned integer types start with a `u`, while the signed types generally don't have a prefix.

TIP

You don't need any unsigned integer versions in this book.

Representing Fractions

Integers are useful for most calculations. However, many calculations involve fractions, which simple integers can't accurately represent. The common equation for converting from Fahrenheit to Celsius temperatures demonstrates the problem, like this:

```
// Convert the temperature 41 degrees Fahrenheit.
int fahr = 41;
int celsius = (fahr - 32) * (5 / 9)
```

This equation works just fine for some values. For example, 41 degrees Fahrenheit is 5 degrees Celsius.

Okay, try a different value: 100 degrees Fahrenheit. Working through the equation, 100–32 is 68; 68 times 5/9 is 37 when using integers. However, a closer answer is 37.78. Even that's wrong because it's really 37.777 . . . with the 7s repeating forever.


REMEMBER

An `int` can represent only integer numbers. The integer equivalent of 37.78 is 37. This lopping off of the fractional part of a number to get it to fit into an integer variable is called *integer truncation.*


TECHNICAL STUFF

Truncation is not the same thing as *rounding.* Truncation lops off the fractional part. Goodbye, Charlie. Rounding picks the closest integer value. Thus, truncating 1.9 results in 1. Rounding 1.9 results in 2.

For temperatures, 37 may be good enough. It's not like you wear short-sleeve shirts at 37.7 degrees but pull on a sweater at 37 degrees. But integer truncation is unacceptable for many, if not most, applications.

Actually, the problem is much worse than that. An `int` can't handle the ratio 5/9 either; it always yields the value 0. Consequently, the equation as written in this example calculates `celsius` as 0 for all values of `fahr`.

Handling Floating-Point Variables

The limitations of an `int` variable are unacceptable for some applications. The range generally isn't a problem — the double-zillion range of a 64-bit-long integer should be enough for almost anyone. However, the fact that an `int` is limited to whole numbers is a bit harder to swallow.

In some cases, you need numbers that can have a nonzero fractional part. Mathematicians call these *real numbers.* (Somehow that always seemed like a ridiculous name for a number. Are integer numbers somehow unreal?)


REMEMBER

Note that a real number *can* have a nonzero fractional part — that is, 1.5 is a real number, but so is 1.0. For example, 1.0 + 0.1 is 1.1. Just keep that point in mind as you read the rest of this chapter.

Fortunately, C# understands real numbers. Real numbers come in two flavors: floating-point and decimal. Floating-point is the most common type. You can find a description of the decimal type in the section "Using the Decimal Type: Is It an Integer or a Float?" later in this chapter.

Declaring a floating-point variable

A floating-point variable carries the designation `float`, and you declare one as shown in this example:

```
float f = 1.0;
```

After you declare it as `float`, the variable `f` is a `float` for the rest of its natural instructions.

Table 2-2 describes the two kinds of floating-point types. All floating-point variables are signed. (There's no such thing as a floating-point variable that can't represent a negative value.)

TABLE 2-2 Size and Range of Floating-Point Variable Types

Type	Bytes	Range of Values	Accuracy to Number of Digits	In Use
`float`	8	$1.5 * 10^{-45}$ to $3.4 * 10^{38}$	6 to 7	`float f = 1.2F;`
`double`	16	$5.0 * 10^{-324}$ to $1.7 * 10^{308}$	15 to 16	`double d = 1.2;`

REMEMBER

You might think that `float` is the default floating-point variable type, but actually the `double` is the default in C#. If you don't specify the type for, say, 12.3, C# calls it a `double`.

The Accuracy column in Table 2-2 refers to the number of significant digits that such a variable type can represent. For example, $\frac{5}{9}$ is actually 0.555 . . . with an unending sequence of 5s. However, a `float` variable is said to have six significant digits of accuracy — which means that numbers after the sixth digit are ignored. Thus $\frac{5}{9}$ may appear this way when expressed as a `float`:

```
0.5555551457382
```

Here you know that all the digits after the sixth 5 are untrustworthy.

The same number — $\frac{5}{9}$ — may appear this way when expressed as a `double`:

```
0.55555555555557823
```

The `double` packs a whopping 15 to 16 significant digits.

TIP

Use `double` variable types unless you have a specific reason to do otherwise. For example, here's the equation for converting from Fahrenheit to Celsius temperatures using floating-point variables:

```
double celsius = (fahr - 32.0) * (5.0 / 9.0)
```

Examining some limitations of floating-point variables

You may be tempted to use floating-point variables all the time because they solve the truncation problem so nicely. Sure, they use up a bit more memory. But memory is cheap these days, so why not? But floating-point variables also have limitations, which you discover in the following sections.

Counting

You can't use floating-point variables as counting numbers. Some C# structures need to count (as in 1, 2, 3, and so on). You know that 1.0, 2.0, and 3.0 are counting numbers just as well as 1, 2, and 3, but C# doesn't know that. For example, given the accuracy limitations of floating-points, how does C# know that you aren't *actually* saying 1.000001?

REMEMBER

Whether you find that argument convincing, you can't use a floating-point variable when counting things.

Comparing numbers

You have to be careful when comparing floating-point numbers. For example, 12.5 may be represented as 12.500001. Most people don't care about that little extra bit on the end. However, the computer takes things extremely literally. To C#, 12.500000 and 12.500001 are not the same numbers.

So, if you add 1.1 to 1.1, you can't tell whether the result is 2.2 or 2.200001. And if you ask, "Is `doubleVariable` equal to 2.2?" you may not get the results you expect. Generally, you have to resort to some bogus comparison like this: "Is the absolute value of the difference between `doubleVariable` and 2.2 less than .000001?" In other words, "within an acceptable margin of error."

TECHNICAL STUFF

Modern processors play a trick to make this problem less troublesome than it otherwise may be: They perform floating-point arithmetic in an especially long `double` format — that is, rather than use 64 bits, they use a whopping 80 bits (or 128-bits in newer processors). When rounding off an 80-bit `float` into a 64-bit `float`, you (almost) always get the expected result, even if the 80-bit number was off a bit or two.

Calculation speed

Integers are always faster than floats to use because integers are less complex. Just as you can calculate the value of something using whole numbers a lot faster than using those pesky decimals, so can processors work faster with integers faster.

TECHNICAL STUFF

Intel processors perform integer math using an internal structure called a general-purpose register that can work only with integers. These same registers are used for counting. Using general-purpose registers is extremely fast. Floating-point numbers require use of a special area that can handle real numbers called the Arithmetic Logic Unit (ALU) and special floating-point registers that don't work for counting. Each calculation takes longer because of the additional handling that floating-point numbers require.

Unfortunately, modern processors are so complex that you can't know precisely how much time you save by using integers. Just know that using integers is generally faster, but that you won't actually see a difference unless you're performing a long list of calculations.

Not-so-limited range

In the past, a floating-point variable could represent a considerably larger range of numbers than an integer type. It still can, but the range of the `long` is large enough to render the point moot.

WARNING

Even though a simple `float` can represent a very large number, the number of significant digits is limited to about six. For example, `123,456,789F` is the same as `123,456,000F`. (For an explanation of the `F` notation at the end of these numbers, see "Declaring Numeric Constants," later in this chapter.)

Using the Decimal Type: Is It an Integer or a Float?

As explained in previous sections of this chapter, both the integer and floating-point types have their problems. Floating-point variables have rounding problems associated with limits to their accuracy, while `int` variables just lop off the fractional part of a variable. In some cases, you need a variable type that offers the best of two worlds:

- Like a floating-point variable, it can store fractions.
- Like an integer, numbers of this type offer exact values for use in computations — for example, 12.5 is really 12.5 and not 12.500001.

Fortunately, C# provides such a variable type, called `decimal`. A `decimal` variable can represent a number between 10^{-28} and 10^{28} — which represents a lot of zeros! And it does so without rounding problems.

Declaring a decimal

Decimal variables are declared and used like any variable type, like this:

```
decimal m1 = 100;   // Good
decimal m2 = 100M;  // Better
```

The first declaration shown here creates a variable `m1` and initializes it to a value of `100`. What isn't obvious is that 100 is actually of type `int`. Thus, C# must convert the `int` into a `decimal` type before performing the initialization. Fortunately, C# understands what you mean — and performs the conversion for you.

The declaration of `m2` is the best. This clever declaration initializes `m2` with the `decimal` constant 100M. The letter *M* at the end of the number specifies that the constant is of type `decimal`. No conversion is required. (See the section "Declaring Numeric Constants," later in this chapter.)

Comparing decimals, integers, and floating-point types

The `decimal` variable type seems to have all the advantages and none of the disadvantages of `int` or `double` types. Variables of this type have a very large range, they don't suffer from rounding problems, and 25.0 is 25.0 and not 25.00001.

The `decimal` variable type has two significant limitations, however. First, a `decimal` is not considered a counting number because it may contain a fractional value. Consequently, you can't use them in flow-control loops, as explained in Chapter 5 of this minibook.

The second problem with `decimal` variables is equally serious or even more so. Computations involving `decimal` values are significantly slower than those involving either simple integer or floating-point values. On a crude benchmark test of 300,000,000 adds and subtracts, the operations involving `decimal` variables were approximately 50 times slower than those involving simple `int` variables. The relative computational speed gets even worse for more complex operations. Besides that, most computational functions, such as calculating sines or exponents, are not available for the `decimal` number type.

Clearly, the `decimal` variable type is most appropriate for applications such as banking, in which accuracy is extremely important but the number of calculations is relatively small.

Examining the bool Type: Is It Logical?

Finally, a logical variable type, one that can help you get to the truth of the matter. The Boolean type `bool` can have two values: `true` or `false`.

WARNING

Former C and C++ programmers are accustomed to using the `int` value 0 (zero) to mean `false` and nonzero to mean `true`. That doesn't work in C#.

You declare a `bool` variable this way:

```
bool thisIsABool = true;
```

No conversion path exists between `bool` variables and any other types. In other words, you can't convert a `bool` directly into something else. (Even if you could, you shouldn't because it doesn't make any sense.) In particular, you can't convert a `bool` into an `int` (such as `false` becoming 0) or a `string` (such as `false` becoming the word "false").

Checking Out Character Types

A program that can do nothing more than spit out numbers may be fine for mathematicians, accountants, insurance agents with their mortality figures, and folks calculating cannon-shell trajectories. (Don't laugh. The original computers were built to generate tables of cannon-shell trajectories to help artillery gunners.) However, for most applications, programs must deal with letters as well as numbers.

C# treats letters in two distinctly different ways: individual characters of type `char` (usually pronounced *char*, as in singe or burn) and strings of characters — a type called, cleverly enough, `string`.

The char variable type

The `char` variable is a box capable of holding a single character. A character constant appears as a character surrounded by a pair of single quotation marks, as in this example:

```
char c = 'a';
```

You can store any single character from the Roman, Hebrew, Arabic, Cyrillic, and most other alphabets. You can also store Japanese katakana and hiragana characters, as well as many Japanese and Chinese kanjis.

In addition, `char` is considered a counting type. That means you can use a `char` type to control the looping structures described in Chapter 5 of this minibook. Character variables do not suffer from rounding problems.

WARNING

The character variable includes no font information. So you may store in a `char` variable what you think is a perfectly good kanji (and it may well be) — but when you view the character, it can look like garbage if you're not looking at it through the eyes of the proper font.

Special chars

Some characters within a given font are not printable, in the sense that you don't see anything when you look at them on the computer screen or printer. The most obvious example of this is the space, which is represented by the character `' '` (single quotation mark, space, single quotation mark). Other characters have no letter equivalent — for example, the tab character. C# uses the backslash to flag these characters, as shown in Table 2-3.

TABLE 2-3 Special Characters

Character Constant	Value
`'\n'`	New line
`'\t'`	Tab
`'\0'`	Null character
`'\r'`	Carriage return
`'\\'`	Backslash

The string type

Another extremely common variable type is the `string`. The following examples show how you declare and initialize `string` variables:

```
// Declare now, initialize later.
string someString1;
someString1 = "this is a string";
// Or initialize when declared – preferable.
string someString2 = "this is a string";
```

A `string` constant, often called a `string` *literal,* is a set of characters surrounded by double quotation marks. The characters in a `string` can include the special characters shown in Table 2-3. A `string` cannot be written across a line in the C# source file, but it can contain the newline character, as the following examples show (see boldface):

```
// The following is not legal.
string someString = "This is a line
and so is this";
// However, the following is legal.
string someString = "This is a line\nand so is this";
```

When written out with `Console.WriteLine`, the last line in this example places the two phrases on separate lines, like this:

```
This is a line
and so is this
```

A `string` is not a counting type. A `string` is also not a value-type — no "string" exists that's intrinsic (built in) to the processor. A computer processor understands only numbers, not letters. The letter *A* is actually the number 65 to the processor. Only one of the common operators works on `string` objects: The `+` operator concatenates two strings into one. For example:

```
string s = "this is a phrase"
         + " and so is this";
```

These lines of code set the `string` variable `s` equal to this character string:

```
"this is a phrase and so is this"
```

WARNING

The `string` with no characters, written `""` (two double quotation marks in a row), is a valid `string`, called an empty `string` (or sometimes a null `string`). A null `string` (`""`) is different from a null `char` (`'\0'`) and from a `string` containing any amount of space, even one (`" "`).

TIP

Best practice is to initialize strings using the `String.Empty` value, which means the same thing as `""` and is less prone to misinterpretation:

```
string mySecretName = String.Empty;  // A property of the String type
```

By the way, all the other data types in this chapter are *value types*. The `string` type, however, is not a value type, as explained in the following section. Chapter 3 of this minibook goes into much more detail about the `string` type.

What's a Value Type?

TECHNICAL STUFF

The variable types described in this chapter are of fixed length — again with the exception of `string`. A fixed-length variable type always occupies the same amount of memory. So if you assign `a = b`, C# can transfer the value of `b` into `a` without taking extra measures designed to handle variable-length types. In addition, these kinds of variables are stored in a special location called the *stack* as actual values. You don't need to worry about the stack; you just need to know that it exists as a location in memory. This characteristic is why these types of variables are called *value types.*

REMEMBER

The types `int`, `double`, and `bool`, and their close derivatives (like unsigned `int`) are intrinsic variable types built right into the processor. The intrinsic variable types plus `decimal` are also known as value types because variables store the actual data. The `string` type is neither — because the variable actually stores a sort of "pointer" to the string's data, called a *reference*. The data in the string is actually off in another location. Think of a reference type as you would an address for a house. Knowing the address tells you the location of the house, but you must actually go to the address to find the physical house.

The programmer-defined types explained in Chapter 8 of this minibook, known as reference types, are neither value types nor intrinsic. The `string` type is a reference type, although the C# compiler does accord it some special treatment because `string` types are so widely used.

Comparing string and char

Although strings deal with characters, the `string` type is amazingly different from the `char`. Of course, certain trivial differences exist. You enclose a character with single quotation marks, as in this example:

```
'a'
```

On the other hand, you put double quotation marks around a string:

```
"this is a string"
"a"   // So is this -- see the double quotes?
```

The rules concerning strings are not the same as those concerning characters. For one thing, you know right up front that a `char` is a single character, and that's it. For example, the following code makes no sense, either as addition or as concatenation:

```
char c1 = 'a';
char c2 = 'b';
char c3 = c1 + c2
```

Actually, this bit of code almost compiles — but with a completely different meaning from what was intended. These statements convert `c1` into an `int` consisting of the numeric value of `c1`. C# also converts `c2` into an `int` and then adds the two integers. The error occurs when trying to store the results back into `c3` — numeric data may be lost storing an `int` into the smaller `char`. In any case, the operation makes no sense.

A string, on the other hand, can be any length. So concatenating two strings, as shown here, *does* make sense:

```
string s1 = "a";
string s2 = "b";
string s3 = s1 + s2;  // Result is "ab"
```

As part of its library, C# defines an entire suite of string operations. You find these operations described in Chapter 3 of this minibook.

TIP

NAMING CONVENTIONS

Programming is hard enough without programmers making it harder. To make your C# source code easier to wade through, adopt a naming convention and stick to it. As much as possible, your naming convention should follow that adopted by other C# programmers:

- **The names of things other than variables start with a capital letter, and variables start with a lowercase letter.** Make these names as descriptive as possible — which often means that a name consists of multiple words. These words should be capitalized but butted up against each other with no underscore between them — for example, `ThisIsALongName`. Names that start with a capital are *Pascal-cased,* from the way a 1970s-era language called Pascal named things.
- **The names of variables start with a lowercase letter.** A typical variable name looks like this: `thisIsALongVariableName`. This variable naming style is called *camel-casing* because it has humps in the middle.

Prior to the .NET era, it was common among Windows programmers to use a convention in which the first letter of the variable name indicated the type of the variable. Most of these letters were straightforward: `f` for `float`, `d` for `double`, `s` for `string`, and so on. The only one that was even the slightest bit different was `n` for `int`. One exception to this rule existed: For reasons that stretch way back into the Fortran programming language of the 1960s, the single letters `i`, `j`, and `k` were also used as common names for an `int`, and they still are in C#. This style of naming variables was called Hungarian notation, after Charles Simonyi, a famous Microsoftie who went to the International Space Station as a space tourist. (Martha Stewart packed his sack lunch.)

Hungarian notation has fallen out of favor, at least in .NET programming circles. With recent Visual Studio versions, you can simply rest the cursor on a variable in the debugger to have its data type revealed in a tooltip box. That makes the Hungarian prefix a bit less useful, although a few folks still hold out for Hungarian.

Calculating Leap Years: DateTime

What if you had to write a program that calculates whether this year is a leap year?

The algorithm looks like this:

```
It's a leap year if
  year is evenly divisible by 4
  and, if it happens to be evenly divisible by 100,
    it's also evenly divisible by 400
```

You don't have enough tools yet to tackle that in C#. But you could just ask the `DateTime` type (which is a value type, like `int`):

```
DateTime thisYear = new DateTime(2011, 1, 1);
bool isLeapYear = DateTime.IsLeapYear(thisYear.Year);
```

The result for 2016 is `true`, but for 2017, it's `false`. (For now, don't worry about that first line of code, which uses some things you haven't gotten to yet.)

With the `DateTime` data type, you can do something like 80 different operations, such as pull out just the month; get the day of the week; add days, hours, minutes, seconds, milliseconds, months, or years to a given date; get the number of days in a given month; and subtract two dates.

The following sample lines use a convenient property of `DateTime` called `Now` to capture the present date and time, and one of the numerous `DateTime` methods that let you convert one time into another:

```
DateTime thisMoment = DateTime.Now;
DateTime anHourFromNow = thisMoment.AddHours(1);
```

You can also extract specific parts of a `DateTime`:

```
int year = DateTime.Now.Year;                  // For example, 2007
DayOfWeek dayOfWeek = DateTime.Now.DayOfWeek;  // For example, Sunday
```

If you print that `DayOfWeek` object, it prints something like "Sunday." And you can do other handy manipulations of `DateTime`s:

```
DateTime date = DateTime.Today;                // Get just the date part.
TimeSpan time = thisMoment.TimeOfDay;          // Get just the time part.
TimeSpan duration = new TimeSpan(3, 0, 0, 0);  // Specify length in days.
DateTime threeDaysFromNow = thisMoment.Add(duration);
```

The first two lines just extract portions of the information in a `DateTime`. The next two lines add a *duration* (length of time) to a `DateTime`. A duration differs from a moment in time; you specify durations with the `TimeSpan` class, and moments with `DateTime`. So the third line sets up a `TimeSpan` of three days, zero hours, zero minutes, and zero seconds. The fourth line adds the three-day duration to the `DateTime` representing right now, resulting in a new `DateTime` whose day component is three greater than the day component for `thisMoment`.

Subtracting a `DateTime` from another `DateTime` (or a `TimeSpan` from a `DateTime`) returns a `DateTime`:

```
TimeSpan duration1 = new TimeSpan(1, 0, 0);  // One hour later.
// Since Today gives 12:00:00 AM, the following gives 1:00:00 AM:
DateTime anHourAfterMidnight = DateTime.Today.Add(duration1);
Console.WriteLine("An hour after midnight will be {0}", anHourAfterMidnight);
DateTime midnight = anHourAfterMidnight.Subtract(duration1);
Console.WriteLine("An hour before 1 AM is {0}", midnight);
```

The first line of the preceding code creates a `TimeSpan` of one hour. The next line gets the date (actually, midnight this morning) and adds the one-hour span to it, resulting in a `DateTime` representing 1:00 a.m. today. The next-to-last line subtracts a one-hour duration from 1:00 a.m. to get 12:00 a.m. (midnight).

Declaring Numeric Constants

There are very few absolutes in life; however, C# does have an absolute: Every expression has a value and a type. In a declaration such as `int n`, you can easily see that the variable `n` is an `int`. Further, you can reasonably assume that the type of a calculation `n + 1` is an `int`. However, what type is the constant `1`?

The type of a constant depends on two things: its value and the presence of an optional descriptor letter at the end of the constant. Any integer type less than 2 billion is assumed to be an `int`. Numbers larger than 2 billion are assumed to be `long`. Any floating-point number is assumed to be a `double`.

Table 2-4 demonstrates constants that have been declared to be of a particular type. The case of these descriptors is not important; `1U` and `1u` are equivalent.

TABLE 2-4 **Common Constants Declared along with Their Types**

Constant	Type
`1`	`int`
`1U`	`unsigned int`
`1L`	`long int` (avoid lowercase *l*; it's too much like the digit 1)
`1.0`	`double`
`1.0F`	`float`

Constant	Type
`1M`	`decimal`
`true`	`bool`
`false`	`bool`
`'a'`	`char`
`'\n'`	`char` (the character newline)
`'\x123'`	`char` (the character whose numeric value is hex 123)[1]
`"a string"`	`string`
`""`	`string` (an empty string); same as `String.Empty`

[1] *"hex" is short for hexadecimal (numbers in base 16 rather than in base 10).*

Changing Types: The Cast

Humans don't treat different types of counting numbers differently. For example, a normal person (as distinguished from a C# programmer) doesn't think about the number 1 as being signed, unsigned, short, or long. Although C# considers these types to be different, even C# realizes that a relationship exists between them. For example, this bit of code converts an `int` into a `long`:

```
int intValue = 10;
long longValue;
longValue = intValue;  // This is OK.
```

An `int` variable can be converted into a `long` because any possible value of an `int` can be stored in a `long` — and because they are both counting numbers. C# makes the conversion for you automatically without comment. This is called an *implicit* type conversion.

A conversion in the opposite direction can cause problems, however. For example, this line is illegal:

```
long longValue = 10;
int intValue;
intValue = longValue;  // This is illegal.
```

TIP

Some values that you can store in a `long` don't fit in an `int` (4 billion, for example). If you try to shoehorn such a value into an `int`, C# generates an error because data may be lost during the conversion process. This type of bug is difficult to catch.

But what if you know that the conversion is okay? For example, even though `longValue` is a `long`, maybe you know that its value can't exceed 100 in this particular program. In that case, converting the `long` variable `longValue` into the `int` variable `intValue` would be okay.

You can tell C# that you know what you're doing by means of a cast:

```
long longValue = 10;
int intValue;
intValue = (int)longValue;  // This is now OK.
```

In a *cast*, you place the name of the type you want in parentheses and put it immediately in front of the value you want to convert. This cast forces C# to convert the `long` named `longValue` into an `int` and assumes that you know what you're doing. In retrospect, the assertion that you know what you're doing may seem overly confident, but it's often valid.

A counting number can be converted into a floating-point number automatically, but converting a floating-point into a counting number requires a cast:

```
double doubleValue = 10.0;
long longValue = (long)doubleValue;
```

All conversions to and from a `decimal` require a cast. In fact, all numeric types can be converted into all other numeric types through the application of a cast. Neither `bool` nor `string` can be converted directly into any other type.

Built-in C# methods can convert a number, character, or Boolean into its string equivalent, so to speak. For example, you can convert the `bool` value `true` into the `string` "true"; however, you cannot consider this change a direct conversion. The `bool true` and the `string` "true" are completely different things.

Letting the C# Compiler Infer Data Types

So far in this book — well, so far in this chapter — when you declared a variable, you *always* specified its exact data type, like this:

```
int i = 5;
string s = "Hello C#";
double d = 1.0;
```

You're allowed to offload some of that work onto the C# compiler, using the `var` keyword:

```
var i = 5;
var s = "Hello C# 4.0";
var d = 1.0;
```

Now the compiler *infers* the data type for you — it looks at the stuff on the right side of the assignment to see what type the left side is.

For what it's worth, Chapter 3 of this minibook shows how to calculate the type of an expression like the ones on the right side of the assignments in the preceding example. Not that you need to do that — the compiler mostly does it for you. Suppose, for example, you have an initializing expression like this:

```
var x = 3.0 + 2 - 1.5;
```

The compiler can figure out that `x` is a `double` value. It looks at `3.0` and `1.5` and sees that they're of type `double`. Then it notices that `2` is an `int`, which the compiler can convert *implicitly* to a `double` for the calculation. All the additional terms in `x`'s initialization expression end up as `double` types. So the *inferred type* of `x` is `double`.

But now, you can simply utter the magic word `var` and supply an initialization expression, and the compiler does the rest:

```
var aVariable = <initialization expression here>;
```

If you've worked with a scripting language such as JavaScript or VBScript, you may have gotten used to all-purpose-in-one data types. VBScript calls them `Variant` data types — and a `Variant` can be anything at all. But does `var` in C# signify a `Variant` type? Not at all. The object you declare with `var` definitely has a C# data type, such as `int`, `string`, or `double`. You just don't have to declare what it is.

What's really lurking in the variables declared in this example with `var`? Take a look at this:

```
var aString = "Hello C# 3.0";
Console.WriteLine(aString.GetType().ToString());
```

The mumbo jumbo in that `WriteLine` statement calls the `String.GetType()` method on `aString` to get its C# type. Then it calls the resulting object's

ToString() method to display the object's type. Here's what you see in the console window:

```
System.String
```

The output from this code proves that the compiler correctly inferred the type of aString.


TIP

Most of the time, the best practice is to not use var. Save it for when it's necessary. Being explicit about the type of a variable is clearer to anyone reading your code than using var.

You see examples later in which var is definitely called for, and you use it part of the time throughout this book, even sometimes where it's not strictly necessary. You need to see it used, and use it yourself, to internalize it.


TIP

You can see var used in other ways: with arrays and collections of data, in Chapter 6 of this minibook, and with anonymous types, in Book 2. Anonymous? Bet you can't wait.

What's more, a type in C# 4.0 and later is even more flexible than var: The dynamic type takes var a step further.

The var type causes the compiler to infer the type of the variable based on expected input. The dynamic keyword does this at runtime, using a set of tools called the Dynamic Language Runtime. You can find more about the dynamic type in Chapter 6 of Book 3.

IN THIS CHAPTER

- Pulling and twisting a string with C#
- Matching searching, trimming, splitting, and concatenating strings
- Parsing strings read into the program
- Formatting output strings manually or using the `String.Format()` method

Chapter 3
Pulling Strings

For many applications, you can treat a `string` like one of the built-in value-type variable types such as `int` or `char`. Certain operations that are otherwise reserved for these intrinsic types are available to strings:

```
Int i = 1;          // Declare and initialize an int.
string s = "abc";   // Declare and initialize a string.
```

In other respects, as shown in the following example, a `string` is treated like a user-defined class (Book 2 discusses classes):

```
string s1 = new String();
string s2 = "abcd";
int lengthOfString = s2.Length;
```

Which is it — a value type or a class? In fact, `String` is a class for which C# offers special treatment because strings are so widely used in programs. For example, the keyword `string` is synonymous with the class name `String`, as shown in this bit of code:

```
String s1 = "abcd"; // Assign a string literal to a String obj.
string s2 = s1;     // Assign a String obj to a string variable.
```

In this example, `s1` is declared to be an object of class `String` (spelled with an uppercase *S*) whereas `s2` is declared as a simple `string` (spelled with a lowercase *s*). However, the two assignments demonstrate that `string` and `String` are of the same (or compatible) types.

In fact, this same property is true of the other intrinsic variable types, to a more limited extent. Even the lowly `int` type has a corresponding class `Int32`, `double` has the class `Double`, and so on. The distinction here is that `string` and `String` truly are the same thing.

The rest of the chapter covers `Strings` and `strings` and all the tasks you can accomplish by using them.

The Union Is Indivisible, and So Are Strings

You need to know at least one thing that you didn't learn before the sixth grade: You can't change a `string` object after creating it. Even though you may see text that speaks of modifying a string, C# doesn't have an operation that modifies the actual `string` object. Plenty of operations appear to modify the `string` that you're working with, but they always return the modified `string` as a new object instead. The new string contains the modified text and has the same name as the existing string, but it really is a new string.

For example, the operation `"His name is " + "Randy"` changes neither of the two strings, but it generates a third string, `"His name is Randy"`. One side effect of this behavior is that you don't have to worry about someone modifying a `string` that you create. Consider this example program:

```
using System;

// ModifyString -- The methods provided by class String do
//    not modify the object itself. (s.ToUpper() doesn't
//    modify 's'; rather it returns a new string that has
//    been converted.)
namespace ModifyString
{
  class Program
  {
    public static void Main(string[] args)
    {
      // Create a student object.
      Student s1 = new Student();
      s1.Name = "Jenny";
```

```
      // Now make a new object with the same name.
      Student s2 = new Student();
      s2.Name = s1.Name;

      // "Changing" the name in the s1 object does not
      // change the object itself because ToUpper() returns
      // a new string without modifying the original.
      s2.Name = s1.Name.ToUpper();
      Console.WriteLine("s1 - " + s1.Name + ", s2 - " + s2.Name);

      // Wait for user to acknowledge the results.
      Console.WriteLine("Press Enter to terminate...");
      Console.Read();
    }
  }

  // Student -- You just need a class with a string in it.
  class Student
  {
    public String Name;
  }
}
```

Book 2 fully discusses classes, but for now, you can see that the `Student` class contains a data variable called `Name`, of type `String`. The `Student` objects `s1` and `s2` are set up so the student `Name` data in each points to the same string data. `ToUpper()` converts the string `s1.Name` to all uppercase characters. Normally, this would be a problem because both `s1` and `s2` point to the same object. However, `ToUpper()` does not change `Name` — it creates a new, independent uppercase string and stores it in the object `s2`. Now the two `Student`s don't point to the same string data. Here's some sample output from this program:

```
s1 - Jenny, s2 - JENNY
Press Enter to terminate...
```

This property of strings is called *immutability* (meaning unchangeability).

The immutability of strings is also important for `string` constants. A string such as `"this is a string"` is a form of a `string` constant, just as `1` is an `int` constant. A compiler may choose to combine all accesses to the single constant `"this is a string"`. Reusing `string` constants can reduce the *footprint* of the resulting program (its size on disc or in memory) but would be impossible if anyone could modify the string.

Performing Common Operations on a String

C# programmers perform more operations on strings than Beverly Hills plastic surgeons do on Hollywood hopefuls. Virtually every program uses the addition operator that's used on `string`s, as shown in this example:

```
string name = "Randy";
Console.WriteLine("His name is " + name); // + means concatenate.
```

The `String` class provides this special operator. However, the `String` class also provides other, more direct methods for manipulating strings. You can see the complete list by looking up "String class" in the Visual Studio Help Index, and you'll meet many of the usual suspects in this chapter. Among the string-related tasks I cover here are the ones described in this list:

- Comparing strings — for equality or for tasks like alphabetizing
- Changing and converting strings in various ways: replacing part of a string, changing case, and converting between strings and other things
- Accessing the individual characters in a string
- Finding characters or substrings inside a string
- Handling input from the command line
- Managing formatted output
- Working efficiently with strings using the `StringBuilder`

Comparing Strings

It's common to need to compare two strings. For example, did the user input the expected value? Or maybe you have a list of strings and need to alphabetize them. Best practice calls for avoiding the standard == and != comparison operators and to use the built-in comparison functions because strings can have nuances of difference between them, and these operators don't always work as expected. In addition, using the comparison functions makes the kind of comparison you want clearer and makes your code easier to maintain. The article at `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/how-to-compare-strings` provides some additional details on this issue, but the following sections tell you all you need to know about comparing two strings.

Equality for all strings: The Compare() method

Numerous operations treat a string as a single object — for example, the `Compare()` method. `Compare()`, with the following properties, compares two strings as though they were numbers:

- If the left string is *greater than* the right string, `Compare(`*`left`*`,`*`right`*`)` returns 1.
- If the left string is *less than* the right string, it returns –1.
- If the two strings are equal, it returns 0.

The algorithm works as follows when written in *notational C#* (that is, C# without all the details, also known as *pseudocode*):

```
compare(string s1, string s2)
{
  // Loop through each character of the strings until
  // a character in one string is greater than the
  // corresponding character in the other string.
  foreach character in the shorter string
    if (s1's character > s2's character when treated as a number)
      return 1
    if (s2's character < s1's character)
      return –1
  // Okay, every letter matches, but if the string s1 is longer,
  // then it's greater.
  if s1 has more characters left
    return 1
  // If s2 is longer, it's greater.
  if s2 has more characters left
    return –1
  // If every character matches and the two strings are the same
  // length, then they are "equal."
  return 0
}
```

Thus, `"abcd"` is greater than `"abbd"`, and `"abcde"` is greater than `"abcd"`. More often than not, you don't care whether one string is greater than the other, but only whether the two strings are equal. You *do* want to know which string is bigger when performing a sort.

REMEMBER

The `Compare()` method returns 0 when two strings are identical (as shown by the code in boldface type in the following listing). The following test program uses the equality feature of `Compare()` to perform a certain operation when the program encounters a particular string or strings. `BuildASentence` prompts the user to enter lines of text. Each line is concatenated to the previous line to build a single sentence. This program ends when the user enters the word *EXIT*, *exit*, *QUIT*, or *quit*. (You'll see after the code what the code in bold does.)

```
using System;

// BuildASentence -- The following program constructs sentences
//   by concatenating user input until the user enters one of the
//   termination characters. This program shows when you need to look for
//   string equality.
namespace BuildASentence
{
  public class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Each line you enter will be "
                      + "added to a sentence until you "
                      + "enter EXIT or QUIT");

      // Ask the user for input; continue concatenating
      // the phrases input until the user enters exit or
      // quit (start with an empty sentence).
      string sentence = "";

      for (; ; )
      {
        // Get the next line.
        Console.WriteLine("Enter a string ");
        string line = Console.ReadLine();

        // Exit the loop if line is a terminator.
        string[] terms = { "EXIT", "exit", "QUIT", "quit" };

        // Compare the string entered to each of the
        // legal exit commands.
        bool quitting = false;

        foreach (string term in terms)
        {
          // Break out of the for loop if you have a match.
          if (String.Compare(line, term) == 0)
          {
            quitting = true;
          }
```

```
        }
        if (quitting == true)
        {
          break;
        }

        // Otherwise, add it to the sentence.
        sentence = String.Concat(sentence, line);

        // Let the user know how she's doing.
        Console.WriteLine("\nyou've entered: " + sentence);
      }

      Console.WriteLine("\ntotal sentence:\n" + sentence);

      // Wait for user to acknowledge the results.
      Console.WriteLine("Press Enter to terminate...");
      Console.Read();
    }
  }
}
```

After prompting the user for what the program expects, the program creates an empty initial sentence string called `sentence`. From there, the program enters an infinite loop.


REMEMBER

The controls `while(true)` and `for(;;)` loop forever, or at least long enough for some internal `break` or `return` to break you out. The two loops are equivalent, and in practice, you'll see them both. (Looping is covered in Chapter 5 of this minibook.)

`BuildASentence` prompts the user to enter a line of text, which the program reads using the `ReadLine()` method. Having read the line, the program checks to see whether it is a terminator by using the code in boldface in the preceding example.

The termination section of the program defines an array of strings called `terms` and a `bool` variable `quitting`, initialized to `false`. Each member of the `terms` array is one of the strings you're looking for. Any of these strings causes the program to end.


WARNING

The program must include both `"EXIT"` and `"exit"` because `Compare()` considers the two strings to be different by default. (The way the program is written, these are the only two ways to spell *exit*. Strings such as `"Exit"` and `"eXit"` aren't recognized as terminators.) You can also use other string operations to check for various spellings of exit. You see how to perform this task in the next section.

The termination section loops through each of the strings in the array of target strings. If `Compare()` reports a match to any of the terminator phrases, `quitting` is set to `true`. If `quitting` remains `false` after the termination section and `line` is not one of the terminator strings, it is concatenated to the end of the sentence using the `String.Concat()` method. The program outputs the immediate result so that the user can see what's going on. Iterating through an array is a classic way to look for one of various possible values. (The next section shows you another way, and Book 2 gives you an even cooler way.) Here's a sample run of the `BuildASentence` program:

```
Each line you enter will be added to a
sentence until you enter EXIT or QUIT
Enter a string
Programming with

You've entered: Programming with
Enter a string
 C# is fun

You've entered: Programming with C# is fun
Enter a string
 (more or less)

You've entered: Programming with C# is fun (more or less)
Enter a string
EXIT

Total sentence:
Programming with C# is fun (more or less)
Press Enter to terminate...
```

Would you like your compares with or without case?

The `Compare()` method used in the previous example considers `"EXIT"` and `"exit"` different strings. However, the `Compare()` method has a second version that includes a third argument. This argument indicates whether the comparison should ignore the letter case. A `true` indicates "ignore."

The following version of the lengthy termination section in the `BuildASentence` example sets `quitting` to `true` whether the string passed is uppercase, lowercase, or a combination of the two:

```
// Indicate true if passed either exit or quit,
// irrespective of case.
```

```
  if (String.Compare("exit", source, true) == 0) ||
        (String.Compare("quit", source, true) == 0)
  {
    quitting = true;
  }
}
```

This version is simpler than the previous looping version. This code doesn't need to worry about case, and it can use a single conditional expression because it now has only two options to consider instead of a longer list: any spelling variation of QUIT or EXIT.

What If I Want to Switch Case?

You may be interested in whether all the characters (or just one) in a string are uppercase or lowercase characters. And you may need to convert from one to the other.

Distinguishing between all-uppercase and all-lowercase strings

You can use the `switch` statement (see Chapter 5 of this minibook) to look for a particular string. Normally, you use the `switch` statement to compare a counting number to some set of possible values; however, `switch` does work on `string` objects as well. This version of the termination section in `BuildASentence` uses the `switch` construct:

```
switch(line)
{
  case "EXIT":
  case "exit":
  case "QUIT":
  case "quit":
    return true;
}
return false;
```

This approach works because you're comparing only a limited number of strings. The `for` loop offers a much more flexible approach for searching for string values. Using the case-less `Compare()` in the previous section gives the program greater flexibility in understanding the user.

Converting a string to upper- or lowercase

Suppose you have a string in lowercase and need to convert it to uppercase. You can use the `ToUpper()` method:

```
string lowcase = "armadillo";
string upcase = lowcase.ToUpper();  // ARMADILLO.
```

Similarly, you can convert uppercase to lowercase with `ToLower()`.

What if you want to convert just the first character in a string to uppercase? The following rather convoluted code will do it (but you can see a better way in the last section of this chapter):

```
string name = "chuck";
string properName =
   char.ToUpper(name[0]).ToString() + name.Substring(1, name.Length - 1);
```

The idea in this example is to extract the first `char` in `name` (that's `name[0]`), convert it to a one-character string with `ToString()`, and then tack on the remainder of `name` after removing the old lowercase first character with `Substring()`.

You can tell whether a string is uppercased or lowercased by using this scary-looking `if` statement:

```
if (string.Compare(line.ToUpper(CultureInfo.InvariantCulture),
                   line, false) == 0) ...  // True if line is all upper.
```

Here the `Compare()` method is comparing an uppercase version of `line` to `line` itself. There should be no difference if `line` is already uppercase. The `CultureInfo.InvariantCulture` property tells `Compare()` to perform the comparison without considering culture. You can read more about it at `https://msdn.microsoft.com/library/system.globalization.cultureinfo.invariantculture.aspx`. If you want to ensure that the string contains all lowercase characters, stick a not (`!`) operator in front of the `Compare()` call. Alternatively, you can use a loop, as described in the next section.

Looping through a String

You can access individual characters of a string in a `foreach` loop. The following code steps through the characters and writes each to the console — just another (roundabout) way to write out the string:

```
string favoriteFood = "cheeseburgers";
foreach(char c in favoriteFood)
{
  Console.Write(c);  // Could do things to the char here.
}
Console.WriteLine();
```

You can use that loop to solve the problem of deciding whether `favoriteFood` is all uppercase. (See the previous section for more about case.)

```
bool isUppercase = true;  // Assume that it's uppercase.
foreach(char c in favoriteFood)
{
  if(!char.IsUpper(c))
  {
    isUppercase = false;  // Disproves all uppercase, so get out.
    break;
  }
}
```

At the end of the loop, `isUppercase` will either be `true` or `false`. As shown in the final example in the previous section on switching case, you can also access individual characters in a string by using an array index notation.

REMEMBER

Arrays start with zero, so if you want the first character, you ask for index `[0]`. If you want the third, you ask for index `[2]`.

```
char thirdChar = favoriteFood[2];   // First 'e' in "cheeseburgers"
```


Searching Strings

What if you need to find a particular word, or a particular character, inside a string? Maybe you need its index so that you can use `Substring()`, `Replace()`, `Remove()`, or some other method on it. In this section, you see how to find individual characters or substrings using `favoriteFood` from the previous section.

Can I find it?

The simplest task is finding an individual character with `IndexOf()`:

```
int indexOfLetterS = favoriteFood.IndexOf('s');  // 4.
```

Class `String` also has other methods for finding things, either individual characters or substrings:

- `IndexOfAny()` takes an array of `chars` and searches the string for any of them, returning the index of the first one found.

  ```
  char[] charsToLookFor = { 'a', 'b', 'c' };
  int indexOfFirstFound = favoriteFood.IndexOfAny(charsToLookFor);
  ```

  That call is often written more briefly this way:

  ```
  int index = name.IndexOfAny(new char[] { 'a', 'b', 'c' });
  ```

- `LastIndexOf()` finds not the first occurrence of a character but the last.
- `LastIndexOfAny()` works like `IndexOfAny()`, but starting at the end of the string.
- `Contains()` returns `true` if a given substring can be found within the target string:

  ```
  if(favoriteFood.Contains("ee")) ...            // True
  ```

- And `Substring()` returns the string (if it's there), or empty (if not):

  ```
  string sub = favoriteFood.Substring(6, favoriteFood.Length - 6);
  ```

Is my string empty?

How can you tell if a target string is empty (`""`) or has the value `null`? (`null` means that no value has been assigned yet, not even to the empty string.) Use the `IsNullOrEmpty()` method, like this:

```
bool notThere = string.IsNullOrEmpty(favoriteFood);  // False
```

Notice how you call `IsNullOrEmpty()`: `string.IsNullOrEmpty(s)`. You can set a string to the empty string in these two ways:

```
string name = "";
string name = string.Empty;
```

Getting Input from the Command Line

A common task in console applications is getting the information that the user types when the application prompts for input, such as an interest rate or a name. The console methods provide all input in string format. Sometimes you need to parse the input to extract a number from it. And sometimes you need to process lots of input numbers.

Trimming excess white space

First, consider that in some cases, you don't want to mess with any white space on either end of the string. The term *white space* refers to the characters that don't normally display on the screen — for example, space, newline (or `\n`), and tab (`\t`). You may sometimes also encounter the carriage return character, `\r`.You can use the `Trim()` method to trim off the edges of the string, like this:

```
// Get rid of any extra spaces on either end of the string.
random = random.Trim();
```

Class `String` also provides `TrimFront()` and `TrimEnd()` methods for getting more specific, and you can pass an array of `char`s to include in the trimming process. For example, you might trim a leading currency sign, such as `'$'`. Cleaning up a string can make it easier to parse. The trim methods return a new string.

Parsing numeric input

A program can read from the keyboard one character at a time, but you have to worry about newlines and so on. An easier approach reads a string and then *parses* the characters out of the string.

Parsing characters out of a string is necessary at times, but some programmers abuse this technique. In some cases, they're too quick to jump into the middle of a string and start pulling out what they find there. This is particularly true of C++ programmers because that's the only way they could deal with strings — until the addition of a string class.

The `ReadLine()` method used for reading from the console returns a `string` object. A program that expects numeric input must convert this `string`. C# provides just the conversion tool you need in the `Convert` class. This class provides

a conversion method from `string` to each built-in variable type. Thus, this code segment reads a number from the keyboard and stores it in an `int` variable:

```
string s = Console.ReadLine();  // Keyboard input is string data
int n = Convert.ToInt32(s);     // but you know it's meant to be a number.
```


REMEMBER

The other conversion methods are a bit more obvious: `ToDouble()`, `ToFloat()`, and `ToBoolean()`. `ToInt32()` refers to a 32-bit, signed integer (32 bits is the size of a normal `int`), so this is the conversion method for `int`s. `ToInt64()` handles the size of a `long`.

When `Convert()` encounters an unexpected character type, it can generate unexpected results. Thus, you must know for sure what type of data you're processing and ensure that no extraneous characters are present.

Although you don't know much about methods yet (see Book 2), here's one anyway. The `IsAllDigits()` method returns `true` if the string passed to it consists of only digits. You can call this method prior to converting a string into an integer, assuming that a sequence of nothing but digits is a legal number.

Here's the method:

```
// IsAllDigits -- Return true if all characters
//   in the string are digits.
public static bool IsAllDigits(string raw)
{
  // First get rid of any benign characters at either end;
  // if there's nothing left, you don't have a number.
  string s = raw.Trim();  // Ignore white space on either side.
  if (s.Length == 0) return false;

  // Loop through the string.
  for(int index = 0; index < s.Length; index++)
  {
    // A nondigit indicates that the string probably isn't a number.
    if (Char.IsDigit(s[index]) == false) return false;
  }

  // No nondigits found; it's probably okay.
  return true;
}
```


REMEMBER

To be truly complete, you need to include the decimal point for floating-point variables and include a leading minus sign for negative numbers.

The method `IsAllDigits()` first removes any harmless white space at either end of the string. If nothing is left, the string was blank and could not be an integer. The method then loops through each character in the string. If any of these characters turns out to be a nondigit, the method returns `false`, indicating that the string is probably not a number. If this method returns `true`, the probability is high that you can convert the string into an integer successfully. The following code sample inputs a number from the keyboard and prints it back out to the console.

```
using System;

// IsAllDigits -- Demonstrate the IsAllDigits method.
namespace IsAllDigits
{
  class Program
  {
    public static void Main(string[] args)
    {
      // Input a string from the keyboard.
      Console.WriteLine("Enter an integer number");
      string s = Console.ReadLine();

      // First check to see if this could be a number.
      if (!IsAllDigits(s)) // Call the special method.
      {
        Console.WriteLine("Hey! That isn't a number");
      }
      else
      {
        // Convert the string into an integer.
        int n = Int32.Parse(s);

        // Now write out the number times 2.
        Console.WriteLine("2 * " + n + " = " + (2 * n));
      }

      // Wait for user to acknowledge the results.
      Console.WriteLine("Press Enter to terminate...");
      Console.Read();
    }
    // Place IsAllDigits here...
  }
}
```

The program reads a line of input from the console keyboard. If `IsAllDigits()` returns `false`, the program alerts the user. If not, the program converts the string into a number using an alternative to `Convert.ToInt32(aString)` — the

`Int32.Parse(aString)` call. Finally, the program outputs both the number and two times the number (the latter to prove that the program did, in fact, convert the string as advertised). Here's the output from a sample run of the program:

```
Enter an integer number
1A3
Hey! That isn't a number
Press Enter to terminate...
```


TECHNICAL STUFF

You could let `Convert()` try to convert garbage and handle any exception it may decide to throw. However, a better-than-even chance exists that it won't throw an exception but will just return incorrect results — for example, returning 1 when presented with 1A3. You should validate input data yourself.


TIP

You could instead use `Int32.TryParse(s, n)`, which returns `false` if the parse fails or `true` if it succeeds. If it does work, the converted number is found in the second parameter, an `int` that I named `n`. This method won't throw exceptions. See the next section for an example.

Handling a series of numbers

Often, a program receives a series of numbers in a single line from the keyboard. Using the `String.Split()` method, you can easily break the string into a number of substrings, one for each number, and parse them separately.

The `Split()` method chops a single string into an array of smaller strings using some delimiter. For example, if you tell `Split()` to divide a string using a comma (`,`) as the delimiter, `"1,2,3"` becomes three strings, `"1"`, `"2"`, and `"3"`. (The *delimiter* is whichever character you use to split collections.) The following program uses the `Split()` method to input a sequence of numbers to be summed. The code in bold shows the `Split()` method-specific code.

```
using System;

// ParseSequenceWithSplit; Input a series of numbers separated by commas,
// parse them into integers and output the sum.
namespace ParseSequenceWithSplit
{
  class Program
  {
    public static void Main(string[] args)
    {
      // Prompt the user to input a sequence of numbers.
      Console.WriteLine(
           "Input a series of numbers separated by commas:");
```

```
      // Read a line of text.
      string input = Console.ReadLine();
      Console.WriteLine();

      // Now convert the line into individual segments
      // based upon either commas or spaces.
      char[] dividers = {',', ' '};
      string[] segments = input.Split(dividers);

      // Convert each segment into a number.
      int sum = 0;

      foreach(string s in segments)
      {
        // Skip any empty segments.
        if (s.Length > 0)
        {
          // Skip strings that aren't numbers.
          if (IsAllDigits(s))
          {
            // Convert the string into a 32-bit int.
            int num = 0;
            if (Int32.TryParse(s, out num))
            {
              Console.WriteLine("Next number = {0}", num);
              // Add this number into the sum.
              sum += num;
            }
            // If parse fails, move on to next number.
          }
        }
      }

      // Output the sum.
      Console.WriteLine("Sum = {0}", sum);

      // Wait for user to acknowledge the results.
      Console.WriteLine("Press Enter to terminate...");
      Console.Read();
    }
    // Place IsAllDigits here...
  }
}
```

The `ParseSequenceWithSplit` program begins by reading a string from the keyboard. The program passes the `dividers` array of `char` to the `Split()` method to indicate that the comma and the space are the characters used to separate individual numbers. Either character will cause a split there.

The program iterates through each of the smaller subarrays created by `Split()` using the `foreach` loop statement. The program skips any zero-length subarrays. (This would result from two dividers in a row.) The program next uses the `IsAllDigits()` method to make sure that the string contains a number. (It won't if, for instance, you type `,.3` with an extra nondigit, nonseparator character.) Valid numbers are converted into integers and then added to an accumulator, `sum`. Invalid numbers are ignored. Here's the output of a typical run:

```
Input a series of numbers separated by commas:
1,2, a, 3 4

Next number = 1
Next number = 2
Next number = 3
Next number = 4
Sum = 10
Press Enter to terminate...
```

The program splits the list, accepting commas, spaces, or both as separators. It successfully skips over the `a` to generate the result of `10`. In a real-world program, however, you probably don't want to skip over incorrect input without comment. You almost always want to draw the user's attention to garbage in the input stream.

Joining an array of strings into one string

Class `String` also has a `Join()` method. If you have an array of strings, you can use `Join()` to concatenate all the strings. You can even tell it to put a certain character string between each item and the next in the array:

```
string[] brothers = { "Chuck", "Bob", "Steve", "Mike" };
string theBrothers = string.Join(":", brothers);
```

The result in `theBrothers` is `"Chuck:Bob:Steve:Mike"`, with the names separated by colons. You can put any separator string between the names: `", "`, `"\t"`, `"  "`. The first item is a comma and a space. The second is a tab character. The third is a string of several spaces.

Controlling Output Manually

Controlling the output from programs is an important aspect of string manipulation. Face it: The output from the program is what the user sees. No matter

how elegant the internal logic of the program may be, the user probably won't be impressed if the output looks shabby.

The `String` class provides help in directly formatting string data for output. The following sections examine the `Pad()`, `PadRight()`, `PadLeft()`, `Substring()`, and `Concat()` methods.

Using the Trim() and Pad() methods

In the "Trimming excess white space" section, you see how to use `Trim()` and its more specialized variants, `TrimFront()` and `TrimEnd()`. This section discusses another common method for formatting output. You can use the `Pad` methods, which add characters to either end of a string to expand the string to some predetermined length. For example, you may add spaces to the left or right of a string to left- or right-justify it, or you can add `"*"` characters to the left of a currency number, and so on. The following small `AlignOutput` program uses both `Trim()` and `Pad()` to trim up and justify a series of names (the code specific to `Trim()` and `Pad()`appears in bold):

```
using System;
using System.Collections.Generic;

// AlignOutput -- Left justify and align a set of strings
//    to improve the appearance of program output.
namespace AlignOutput
{
  class Program
  {
    public static void Main(string[] args)
    {
      List<string> names = new List<string> {"Christa  ",
                                             "  Sarah",
                                             "Jonathan",
                                             "Sam",
                                             " Schmekowitz "};

      // First output the names as they start out.
      Console.WriteLine("The following names are of "
                        + "different lengths");

      foreach(string s in names)
      {
        Console.WriteLine("This is the name '" + s + "' before");
      }
      Console.WriteLine();
```