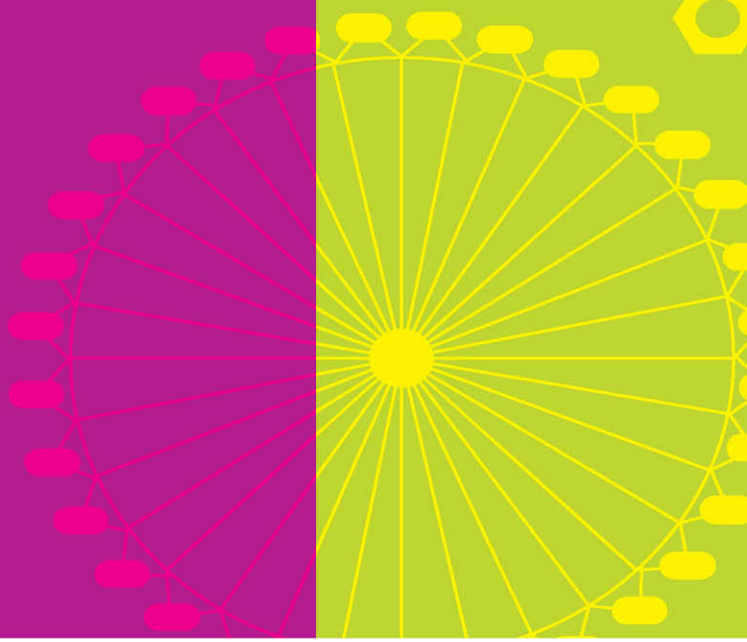
Microsoft

Begin to Code with C#



Rob Miles

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Microsoft

Begin to Code with C#

Rob Miles

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2015942036
ISBN: 978-1-5093-0115-7

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

Acquisitions and Developmental Editor: Devon Musgrave
Project Editor: John Pierce
Editorial Production: Rob Nance and John Pierce
Technical Reviewer: Lance McCarthy; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Copyeditor: John Pierce
Indexer: Christina Palaia, Emerald Editorial Services
Cover: Twist Creative • Seattle

To Mary

Contents at a glance

Part 1: Programming fundamentals

Part 2: Advanced programming

Part 3: Making games

This page intentionally left blank

Contents

Part 3: Making games

Introduction

I think that programming is the most creative thing you can learn how to do. If you learn to paint, you can make pictures. If you learn the violin, you can make music. But if you learn to program, you can create experiences that are entirely new (and you can make pictures and music too if you want to). Once you have started on the programming path, there's no limit to where you can go. There are always new devices, technologies, and marketplaces where you can use your programming skills.

You can think of this book as your first step on a journey to programming enlightenment. The best journeys are undertaken with a destination in mind, and this one is no different. I'd like to describe the destination as "usefulness." By the end of this book you won't be the best programmer in the world (unless I retire, of course), but you will have enough skills and knowledge to write properly useful programs. And maybe you can have at least one of your programs available worldwide for download from the Microsoft Store.

However, before we start off, I'd like to issue a small word of warning. In the same way that a guide would want to tell you about the lions, tigers, and crocodiles that you might encounter if you went on a safari adventure, I feel that I must let you know that our journey might not be all smooth sailing. Programmers have to learn to think slightly differently about problem solving because a computer just doesn't work the same way that we do. Humans can do complex things rather slowly. Computers can do simple things really quickly. It is the job of the programmer to harness the simple abilities of the machine to solve complicated problems. This is what we are going to learn how to do.

The key to success as a programmer is pretty much the same as for lots of other endeavors. If you want to become a world-renowned violin player, you will have to practice a lot. The same is true for programming. You will have to spend quite a bit of time working on your programs to get code-writing skills. But the good news is that, just as a violin player really enjoys making the instrument sing, making a computer do exactly what you want turns out to be a really satisfying experience. And it gets even more enjoyable when you see other people using programs that you have written and finding them useful and fun to use.

How this book fits together

I've organized this book in four parts. Each part builds on the previous one with the aim of turning you into a successful programmer. We start off considering the low-level programing instructions that programs use to tell the computer what to do, and we finish by looking at professional software practices.

Part 1: Coding fundamentals

The first part gets you started. It points you to where you will install and use the programming tools that you will need, and it introduces you to the fundamental elements of the C# programming language that are used by all programs.

Part 2: Advanced programming

Part 2 describes the features of the C# programming language that are used to create more complex applications. It shows you how to break large programs into smaller elements and how you can create custom data types that reflect the specific problem being solved. You'll also find out how programs can maintain data in storage when they are not running.

Part 3: Making games

Making games is great fun. And it turns out that it is also a great way to learn how to use object-oriented programming techniques. In this part, you'll build some playable games and at the same time learn the fundamentals of how to extend programming objects through inheritance and component-based software design.

Part 4: Creating applications

Part 4 is where you find out how to create fully fledged applications. You'll discover how to design graphical user interfaces and how to connect program code to the elements on the display. You'll also learn how modern applications are structured. Part 4 doesn't appear in this printed

book but is available as an ebook, free to download from this book's webpage at https://aka.ms/BeginCodeCSharp/downloads.

How you will learn

In each chapter, I will tell you a bit more about programming. I'll show you how to do something, and then I'll invite you make something of your own by using what you've learned. You'll never be more than a page or so away from doing something or making something unique and personal. In each chapter we will use *Snaps*, prebuilt bits of functionality that I'll show you how to use. After that, it's up to you to make something amazing!

You can read the book straight through if you like, but you'll learn much more if you slow down and work with the practical parts along the way. This book can't really teach you how to program, any more than a book about bicycles can teach you how to ride a bike. You have to put in the time and practice to learn how to do it. But this book will give you the knowledge and confidence to try your hand at programming, and it will also be around to help you if your programming doesn't turn out as you expected. Here are the elements in the book that will help you really learn, by doing!

MAKE SOMETHING HAPPEN

Yes, the best way to learn things is by doing, so you'll find "Make Something Happen" elements throughout the text. These elements offer ways for you to practice your programming skills. Each of them starts with an example and then introduces some steps you can try on your own. Everything you create will run on a Windows PC, tablet, or phone. You can even publish your creations to the whole wide world via the Windows Store.

CODE ANALYSIS

A great way to learn how to program is by looking at code written by other people and working out what it does (and sometimes why it doesn't do what it should). In this book's "Code Analysis" challenges, you'll use your deductive skills to figure out the behavior of a program, fix bugs, and suggest improvements.

WHAT COULD GO WRONG

If you don't already know that programs can fail, you will learn this hard lesson very soon after you start writing your first program. To help you deal with this in advance, I've included "What Could Go Wrong?" elements, which anticipate problems you might have and provide solutions to those problems. For example, when I introduce something new, I'll sometimes spend some time considering how it can fail and what you need to worry about when you use the new feature.

PROGRAMMER'S POINTS

I've spent a lot of my time teaching programming. But I've also written many programs and sold a few to paying customers. I've learned some things the hard way that I really wish I'd known right at the start. The aim of "Programmer's Points" is to give you this information up front so that you can start taking a professional view of software development as you learn how to do it.

"Programmer's Points" cover a wide range of issues, from programming to people to philosophy. I'd strongly advise you to read and absorb these points carefully—they can save you a lot of time in the future!

Programs and Snaps

Nobody builds programs from scratch any more. All software is built using pieces of software that have already been built. If one program wants to display text, make a sound, or play some video, it simply asks another program to do it. Every popular computer language is underpinned by a huge library of existing code, and one of the things that a programmer needs to understand is how to use these libraries and software written by other people.

I've created the Snaps library specially for this book. It provides a set of functional behaviors that are easy to use and fit together. You will use the Snaps library in your first programs. Later in the book you'll discover other libraries of functionality that you can use to build programs.

Programs that use Snaps run inside the Snaps engine, which is a self-contained environment in which programs can speak messages, get input from a user, draw images, make sounds, and even find out what the weather is like.

I'll provide examples of how the Snaps work and then leave it up to you to see what you can come up with. The principle we'll follow is this: "If you can't use programming to impress your friends and family, what's the point of it?" I really hope you'll come up with some impressive programs of your own and maybe even publish them for other people to enjoy.

PROGRAMMER'S POINT

Everything is built on someone else's code

It seems fitting that the first Programmer's Point is about how "creatively lazy" a good programmer can be. They'll never write a program if they can find a way to use one that has already been written. (Why reinvent the wheel?) The Snaps that I've provided are an example of this. You'll take a look inside some of them later in the book and discover that they themselves make use of other libraries.

Software and hardware

You'll need a computer and some software to work with the programs in the book. I'm afraid I can't provide you with a computer, but in the first chapter you will find out where you can get Visual Studio 2015 Community Edition, the free software that you'll use to create your programs. You'll also learn where to download the Snaps library and the demonstration code we'll examine and use.

The computer you use must run the 64-bit version of the Windows 10 operating system. Here are the other requirements:

- A 1 Ghz or faster processor, preferably an Intel i5 or better.
- At least 4 gigabytes (GB) of memory (RAM), but preferably 8 GB or more.
- The full Visual Studio 2015 Community installation takes about 8 GB of hard disk space.

There are no special requirements for the graphics display, although a higher resolution screen will enable you to see more when you are writing your programs. The Snaps library works with touchscreens, a mouse,

pen input devices, and the Xbox One and Xbox 360 controllers for the games you'll develop in Part 3.

Visual Studio 2015 Community Edition is a freely available application that can be used to create C# programs on a Windows 10 PC. If you have an earlier version of Visual Studio on your computer already (Visual Studio 2013, for example), I'm afraid that you can't use it with this book. However, the 2015 version of Visual Studio will work quite happily alongside existing installations. In Chapter 1, I provide a link to detailed instructions for how to install Visual Studio and get it going. To make use of Visual Studio, it's best to have a Microsoft account so that a development license can be assigned to you.

Downloads

In every chapter in this book, I'll demonstrate and explain programs that teach you how to begin to program—and that you can then use to create programs of your own. You can download the Snaps library, this book's sample code, installation and setup instructions for Visual Studio, and the ebook for Part 4, "Creating applications," from the following page:

https://aka.ms/BeginCodeCSharp/downloads

Follow the instructions you'll find in Chapter 1 and in the setup document to install the sample programs and code.

Acknowledgments

I really like to write books. Huge thanks to Devon Musgrave and the folks at Microsoft Press for giving me the chance to write another one, to Rob Nance for the wonderful artwork, and to John Pierce and Lance McCarthy for doing such fantastic work on the text. It turns out that the acknowledgment is the olny part of the buk that they don't see, and I must give them both greatful thanks for making sure that all my text reads rightly.

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

https://aka.ms/BeginCodeCSharp/errata

If you discover an error that is not already listed, please submit it to us at the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

You'll also find "author's notes" about this book, including other projects and information about the Snaps library at:

http://www.robmiles.com/begintocode

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

We know you're busy, so we've kept it short with just a few questions. Your answers go directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress.*

Part 1

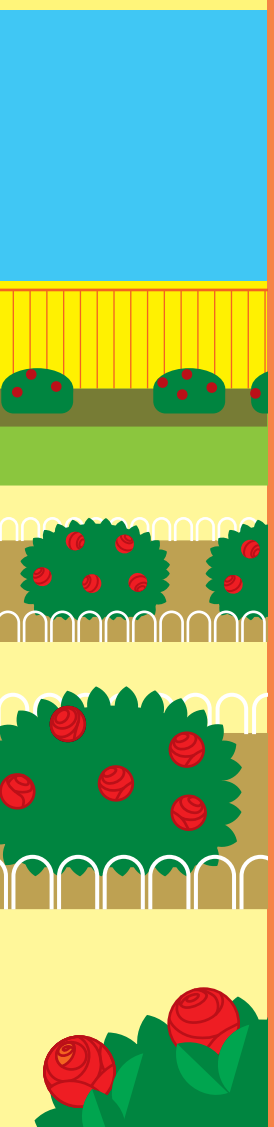
Programming fundamentals

Let's begin traveling toward programming enlightenment. You'll start by installing the programming tools you need. Next you'll discover what a computer actually does and what a programming language is. You'll also take your first small steps in using the C# language to tell a computer to do things for you, and you'll find out how to work with Snaps, small helpers I've created for you to use in your first programs.

The aim of Part 1 is to introduce you to fundamental elements of the C# programming language that are used by all programs. Then, in Part 2, you'll look at how a modern programming language like C# builds on these programming fundamentals to make it easier to create applications.

3

Writing programs



What you will learn

Now that you know a bit about computers, programs, and programmers, you can start to think about writing program code.

In this chapter, you'll closely examine some C# programs to find out how they run. I call these programs "Snaps applications" because they use the Snaps library, a simple collection of programming resources that help you get things done "in a snap." By analyzing how these programs use various Snaps—discrete pieces of programming functionality or behaviors provided by the library—you'll learn some fundamentals of C# programming. Along the way you'll learn more about using Visual Studio to create and manage the code elements in the **BeginToCodeWithCSharp** solution and what to do when the compiler complains that your program doesn't make sense as far as it is concerned.

At the end of this chapter, you will be creating programs that provide simple solutions to some realistic problems.

C# program structure

Let's take a very detailed look at some Snaps applications to understand their elements and the organization of those elements. The welcome that you witness when you first run the **BeginToCodeWithCSharp** solution isn't complicated, but it's a good place for us to start. We quickly examined the code that creates that experience when we analyzed **MyProgram.cs** in Chapter 2. Take a look now at the file named **Ch03_01_WelcomeProgram.cs**. (In case you've forgotten: use Solution Explorer to navigate through the solution's chapter folders to find the file, and then select the file to show its code in the editor window.)

Notice that the code is almost exactly the same as the code as in **MyProgram.cs**, so this program should give us the same experience, right? Let's check that. Go ahead and run the solution again, select **Chapter 03** in the **Folder** list and **Ch03_01_WelcomeProgram** is the **Snaps apps** list, and then run the app. Yep, same experience, which makes perfect sense. Now let's really break down this program to figure out how it's working. Its code is shown next, and I've indicated each part of the program with a callout. We'll examine these parts line by line in the following sections.

```
using SnapsLibrary;

public class Ch03_01_WelcomeProgram
{
    public void StartProgram()
    {
        SnapsEngine.SetTitleString("Begin to Code with C#");
        SnapsEngine.DisplayString("Welcome to the world of Snaps");
    }
}
```

Identify resources.

Start a class definition.

Declare the StartProgram method.

Set the title and display a message.

Identify resources

```
using SnapsLibrary;
```

I described the *C# compiler* in Chapter 2. This is a program that converts a high-level C# program (like the one we're analyzing) into machine code that can run inside your computer. When you run your C# code, the compiler built into Visual Studio converts the program into machine code so that it can be run. A C# program can contain lines called *directives* that give the compiler instructions. This first line of the program is a `using` directive.

As a programmer, you will frequently want to use prebuilt pieces of software, in the same way that a cook will sometimes use readymade pastry. Readymade C# programs are packaged as libraries of components that can be added to a Visual Studio solution. As I've mentioned, the Snaps library is an example of such a library that I've provided to help you get started. The `using` directive here identifies the library as a resource that has been added to our solution and, as you'll see in a moment, this program is going to use something from it, specifically the `SnapsEngine`. This `using` directive says to the compiler, "If I mention something you haven't seen before, go and look in `SnapsLibrary` to see if you can find it there." This is a bit like saying to our cook, "If you need to use some pastry, take a look in the fridge." The first programs that we're going to write in this book use only items in `SnapsLibrary`. Later on we will create programs that use other libraries.

CODE ANALYSIS

Using the `using` directive

In some "Code Analysis" sections, like this one, you don't need to look at any code to consider some code-related questions.

Question: Does the `using` directive actually fetch the library that a program wants to use?

> **Answer:** No. This might sound confusing, but the `using` directive just tells the compiler where to look for the items that are available for use in a program. The resources available to a program are set up in the Visual Studio project. We can change the `using` directive to direct the compiler to use code from different places. This would be like telling the cook, "If you need to use some pastry, check by the sink" so that he would use a resource from a different location.

Question: If I add lots of `using` directives, will this make my program bigger?

> **Answer:** No. The directive just tells the compiler where to look for things. It doesn't add anything to the size of the program.

Start a class definition

```
public class Ch03_01_WelcomeProgram
```

C# can be called an *object-oriented* programming language. This is because, in the universe of C#, everything is an *object*. Objects in a C# program can be as simple as a single number or as complex as an entire video game. An object can contain other

objects. Anything that is contained within another object is called a *member* of that object.

We can express an object design in the form of a C# *class* definition. A C# class definition can describe data members (values that the object can hold) and behavior members (things you can ask the object to do for you). When you design an object, you write C# that specifies these two things. This line of the program tells the compiler that we are expressing the design of a `class` named `Ch03_01_WelcomeProgram`.

You'll find out much more about classes and objects later in the book.

CODE ANALYSIS

Classes and objects

Question: Is a class definition the only way to define an object?

Answer: No. There are other kinds of C# objects, which you will see later.

Question: Does defining a class actually create an object?

Answer: No. Think of the class as the blueprint or design of an object, just like you might have plans for a treehouse. In the same way as having the plans for a treehouse doesn't actually give you a treehouse, having a class definition doesn't actually give you an object.

Question: Do all classes have to contain both data and behaviors?

Answer: No. Some classes contain just data members, and others contain only behavior members. For example, the Math library, which we haven't seen yet, contains classes that can perform mathematical functions.

Question: When does the program actually make an object based on the class `Ch03_01_WelcomeProgram`?

Answer: This happens automatically. The sequence goes like this: The user is running the **BeginToCodeWithCSharp** application and then selects **Ch03_01_WelcomeProgram** and runs it. The **BeginToCodeWithCsharp** application creates an object based on the `Ch03_01_WelomeProgram` class and then runs the `StartProgram` behavior inside this object.

Declare the `StartProgram` method

```
public void StartProgram()
```

Behaviors in an object are expressed in the form of *methods*. A method is a piece of C# code that is given a name. A program can run the code in a method simply by giving the name of the method—this is known as *calling the method*. You are going to start by calling methods that have already been written (by me), but later you will create methods of your own.

This program's single class—`Ch03_01_WelcomeProgram`—has just a single behavior, a method called `StartProgram`. The declaration `public void StartProgram()` marks the beginning of the `StartProgram` method. (The *method modifier* `public` and the *return type* `void` tell us about the nature of this method, but these are details we don't need to get into at the moment.) The `StartProgram` method is special. It is the entry point for a Snaps application. In other words, to start running a Snaps application, the `StartProgram` method is called.

This program's class does not contain any data members but later we will design some objects that do contain data.

CODE ANALYSIS

Declaring methods in classes

Question: What is the difference between a behavior and a method?

Answer: A behavior is an action that an object can perform. The method is the actual C# code that delivers that behavior.

Question: Can a class contain more than one method?

Answer: Yes. A programmer decides how many behaviors a class should provide, and she writes a method for each one. The demo program we've been looking at has only one behavior: to start the demo. Later on we'll create classes with many methods in them.

Question: How does the `StartProgram` method get used?

Answer: `StartProgram` is a special method, in that it defines the starting point for any Snaps application. While we're working in the Snaps environment provided by the Snaps library, we will always call the `StartProgram` method to start a program running.

Set the title and display a message

```
SnapsEngine.SetTitleString("Begin to Code with C#");
SnapsEngine.DisplayString("Welcome to the world of Snaps");
```

The first of these two lines of code is the first C# *statement* in the `StartProgram` method. Statements are the parts of a program that get things done. A statement might call a method, make a decision, or manipulate some data. Statements are held inside methods and are performed when the method is used. The `StartProgram` method contains only two statements; larger programs will contain many more. The two statements within the `StartProgram` method do indeed call other methods.

Each statement in a method is performed in sequence, starting with the first one and then moving on to the next. There are several types of statements that you can use, and you'll find out about these as you learn the C# language. The semicolon (;) character marks the end of each statement.

This first statement sets the title of our program to "Begin to Code with C#". It uses the `SnapsEngine` class to do this. The `SnapsEngine` class is part of the Snaps library—the resource we identified in the first line of this program—and the class provides lots of behaviors that we can use in our programs. You can think of `SnapsEngine` as a kind of "program butler" that can do things for programs that you write.

Each `SnapsEngine` behavior is provided as a C# method that our programs can call. In this example, you can see how to use the `SetTitleString` method in the `SnapsEngine`. Then, in the same way that a "Get me a drink" command to a butler needs to be accompanied by the type of drink you want, the `SetTitleString` method needs to be given the string of text to be used as the title of the program. A C# string is given in parentheses after the name of the method that we're calling. Information added to a call of a method is called an *argument* to the method.

Regarding the string itself, the double quotation mark characters (`"`) in the statement mark the start and end of the string—the string starts immediately after the first double quotation mark and ends immediately before the second one. It's a convention in C# that whenever you want to specify a string of text, you enclose it in double quotation marks like this. If we added spaces in the string text—for example, `" Welcome to Snaps "`—those spaces would also be displayed in the program's title (although a user might not notice them).

The second statement works in the same way as the previous one. It calls a method in the `SnapsEngine` class that displays a string as a message on the screen of the Snaps application (rather than setting a string as a title on the screen). When you saw the `DisplayString` method name, did you expect to see quotation marks and string text within the method's parentheses? Good!

CODE ANALYSIS

Calling methods in classes

Question: Where is the `SetTitleString` method declared?

> **Answer:** The `SetTitleString` method is declared in the `SnapsEngine` class in exactly the same way as the `StartProgram` method is declared in the `Ch03_01_WelcomeProgram` class. Later you will discover how to create your own methods in classes.

Question: What happens if I don't give `SetTitleString` a string to work on?

> **Answer:** The design of `SetTitleString` specifies that a string will be supplied when it is called. The compiler will complain that a program is invalid if the program doesn't provide a string argument to the method call.

Question: Why do we have to put parentheses around the string that we're providing to `SetTitleString`? Surely the compiler can figure out that the string to be displayed will start with a double quotation mark character.

> **Answer:** The reason why we need to include the parentheses is to tell the compiler the start and end of the list of arguments being fed into the method. `SetTitleString` has only one item being fed into it, but other methods might have lots of items. If you look at the text of the program, you'll find that the `StartProgram` method has been specified to accept an "empty" list of arguments, which means that it doesn't work on any items. The designers of the C# language have used different characters to define the limits of (or *delimit*) different elements of the program. As we've seen, strings are *delimited* by double quotation mark characters. Lists of arguments are delimited by open and close parentheses: `(` and `)`. The contents of a class and the body of a method are delimited by curly brackets: `{` and `}`. As you might expect, the compiler is very careful to make sure that the use of these delimiters "makes sense," and it will reject any program that has mismatched delimiters.

You can think of the two statements we just analyzed—which set the screen's title and display a message—as the "payload" of the sample program. The rest of the code around those statements provides the structure around those actions. To write larger programs, you just have to replicate this structure and add more statements. Now that you know how a simple program fits together, you can start to make your own, using the Snaps applications as a starting point. For example, you could make a program that displays two message strings rather than a title and a message, like I've done with **Ch03_02_MoreStatements.cs:**

```
using SnapsLibrary;
```

```
public class Ch03_02_MoreStatements
{
    public void StartProgram()
    {
        SnapsEngine.DisplayString("Hello world");
        SnapsEngine.DisplayString("Goodbye chickens");
    }
}
```

First statement

Second statement

In this program, `SetTitleString` isn't called and two statements call the `DisplayString` method so that the program displays one message followed by a second message. You can put a very large number of statements in a program. You could write a program that displays the Gettysburg Address (or any other long text) one string at a time simply by adding more statements. It's important to remember that each statement is obeyed in order when the program runs. The preceding program will always display "Hello world" before it displays "Goodbye chickens". (Let me point out just one more time that the program doesn't display the double quotation marks you see in the previous sentence because no quotation marks appear in the string's text itself between the double quotation marks that delimit it. The double quotation marks I use in this paragraph are there only for clarity's sake, as I describe the text the program displays.)

So when you display a string via the `DisplayString` method, it replaces the string that was displayed by a previous call of `DisplayString`, if any. In our example, this is why "Hello world" is replaced by "Goodbye chickens". Later you'll discover how to build up multiple lines of text on the screen. Also, you can use `DisplayString` to display very long messages if you want to; the text is automatically wrapped if it extends over the edge of the screen. If you display a message that is extremely long, you'll find that it extends off the bottom of the screen and the user won't be able to read all of it.

Extra Snaps

Every now and then I will introduce other Snaps—behaviors enabled by the Snaps library—that you can play with. You can use these in your programs just like the programs we've been analyzing use `DisplayString`.

SpeakString

You can make programs that speak text instead of displaying it. Here's an example:

```
using SnapsLibrary;

class Ch03_03_Speaking
{
    public void StartProgram()
    {
        SnapsEngine.SpeakString("Hi there. I'm your friendly computer.");
    }
}
```

The `SpeakString` method is used in the same way as the `DisplayString` method, but it causes the computer to speak the text provided instead of displaying it on the screen. This is a useful method because it makes it easy to create programs that can talk.

CODE ANALYSIS

Speak and display

Let's take a look at some code and try to work out why it doesn't do what it should. Let's say that your younger brother wrote this program. He wanted something that displays "Computer Running" and then says "Computer Running," but he complains that the visual message doesn't appear until after the computer has finished speaking.

```
using SnapsLibrary;
class Ch03_04_DoubleOutput
{
    public void StartProgram()
    {
        SnapsEngine.SpeakString("Computer Running");
        SnapsEngine.DisplayString("Computer Running");
    }
}
```

Question: Why does the message appear on the screen after the computer finishes speaking?

> **Answer:** When you are trying to work out what a program does, it is often useful to "behave like the computer" and work through the statements one at a time in sequence. The computer speaks before the message is displayed because it strictly follows the sequence of the statements. The `DisplayString` method doesn't run until after the `SpeakString` method has completed. This problem is fixed by reversing the order of the statements. Take a look at **Ch03_05_DoubleOututFixed.cs** in Visual Studio to see that.

Creating new program files

Programming is very creative, and you'll create your own programs as we go through the book. What I'm really hoping is that you'll have your own ideas for programs and build those along with the ones that I suggest. Each new program that you create will be a new Snaps application that other learners can analyze or use.

You can create a new Snaps app by using the **MyProgram.cs** program file as a starting point. Begin (like we always do) by opening the **BeginToCodeWithSharp** solution file, and then in Solution Explorer find the file in the **My Snaps apps** folder in the **BeginToCodeWithCSharp** project. Right-click the file in Solution Explorer to open the context menu, and then select **Copy**, as shown in **Figure 3-1**.

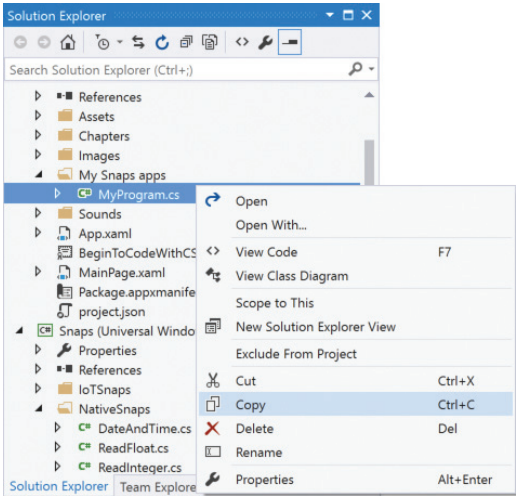


Figure 3-1 Copying a program.

Now paste this copy into the **My Snaps apps** folder by right-clicking the folder and selecting **Paste**, as shown in **Figure 3-2**.

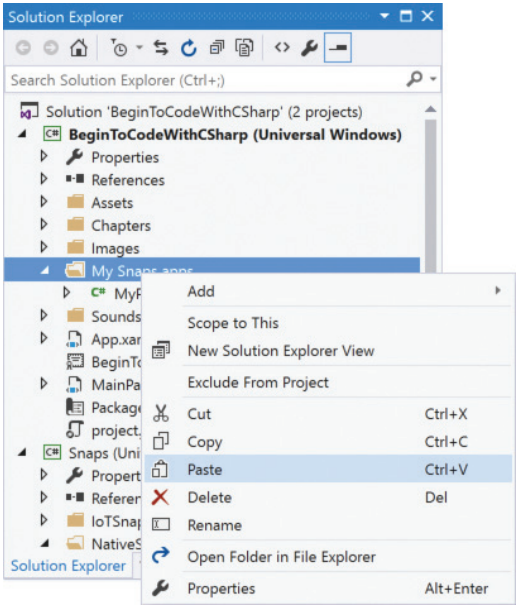


Figure 3-2 Pasting the program.

Figure 3-3 shows the copy, called **MyProgram - Copy.cs**, in the folder.

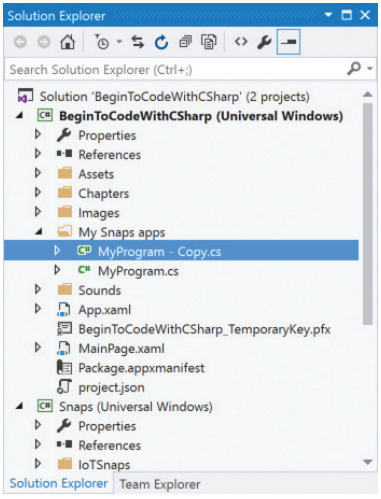


Figure 3-3 The copied program appears in the folder.

Let's rename this new file to reflect the new Snaps app you're going to build. Right-click the file (the one that includes "Copy" in its name) in Solution Explorer to open the context menu again, and select **Rename**. (I won't show this step because I'm sure you know what to do!) Now you can enter the new name for your application, as shown in **Figure 3-4**.

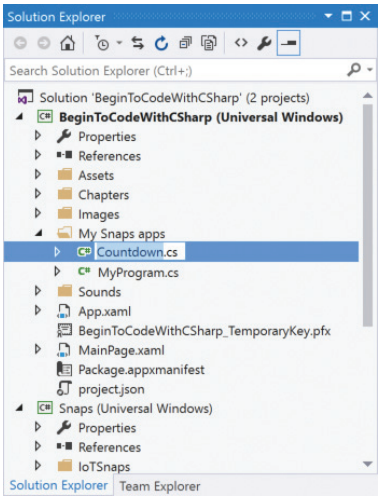


Figure 3-4 Entering the new name.

Change the name of the program to "Countdown". Be very careful not to remove ".cs" at the end of the name. If you remove this part of the file name, Visual Studio will not know the file is a C# program and will not work correctly when you try to run the program. When you have finished entering the name, press Enter. You now have a copy of the original program in a file called **Countdown.cs**. The reason I chose this name will become apparent soon.

The next thing we need to do is rename the class that holds our program. Click the **Countdown.cs** file in Solution Explorer so that its code appears in the editor window, as shown in **Figure 3-5**.

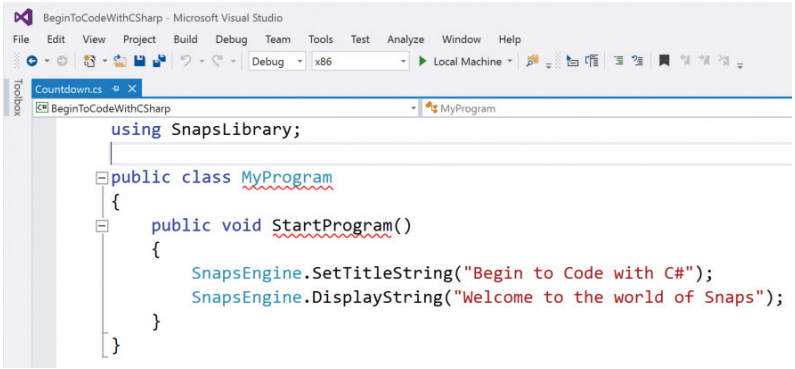


Figure 3-5 The Countdown.cs file open in the Visual Studio editor.

Looking at **Figure 3-5**, you can see that Visual Studio is trying to tell us something. The wavy red lines indicate that Visual Studio thinks some elements of the program's code are wrong. Visual Studio is unhappy in this case because our **BeginToCode-WithCSharp** solution contains two versions of the `MyProgram` class—the original in **MyProgram.cs** and now another in **Countdown.cs**. We can fix this problem by giving the class a new name.

In **Figure 3-6**, I've changed the name of the class to `Countdown` and also changed what the program does by altering one statement (the one that calls `SetTitleString`) and by deleting the other statement (the one that was calling `DisplayString`). The program now just sets its title to "Countdown". You can put whatever you want in the string, of course, but be sure that it has a double quotation mark at each end; other-wise, your program won't compile.

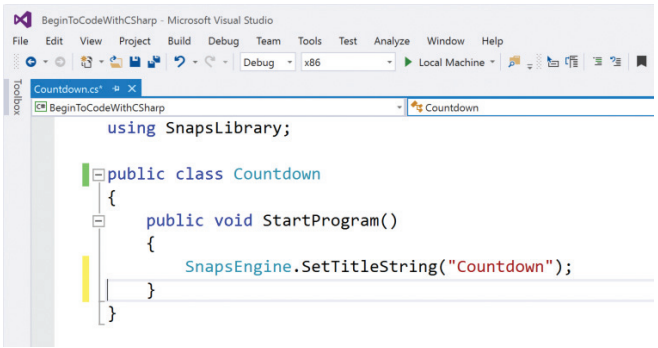


Figure 3-6 Defining a `Countdown` class.

Visual Studio is happy now because we removed the duplicate of the `MyProgram` class. You should now be able to run the program by using the run button (the green arrow).

CODE ANALYSIS

Class names and file names

A C# solution can be spread over a large number of separate program files. It is worth giving some thought to how this works.

Question: Why do we have to change the name of the class when we have already changed the name of the file?

> **Answer:** To answer this question, you have to understand the difference between *logical* and *physical* names in a program. You can think of the names of the files that hold our programs as physical names because a file name is connected to an actual file that is stored on the computer. However, the names of the elements in a program are not tied to the physical file that holds the program's text. They exist in a "logical" namespace that is defined by the programmer.
>
> When the C# compiler is compiling a program, it reads all the source files and builds up a list of all the different items that are defined in the program. This is the logical namespace of the program. Each of the items in this logical namespace must have a unique name. If we create two items with the same name, the compiler will complain, and that is what happened earlier when we copied the **MyProgram.cs** file. After the copy, there were two classes with the name `MyProgram`. We fixed the problem by changing the name of one of the items to a new, unique name.

Question: Does the name of a program's source file (the physical name) and the name of a class (the logical name) in that source file have to match?

> **Answer:** No. It is often convenient to make the two names match because it can make it easier to find particular items, but the C# compiler does not enforce this.

Question: What would have happened if the program already contained a class named `Countdown` and we added another one?

> **Answer:** You can probably guess what would happen. The compiler would complain because it doesn't like having two items with the same name.

By the way, perhaps you were expecting the **Countdown** app to run immediately when you clicked the run button? Whenever the **BeginToCodeWithCSharp** application is first run, the Snaps environment looks for a class named `MyProgram` and then calls the `StartProgram` method in that class. This means that whenever you start the **BeginToCodeWithCSharp** application, it will first run the original program: **MyProgram.cs**. Then you use the **Folder** and **Snaps apps** lists to select other apps you want to run.

You can follow this copy, paste, and revise process each time you want to make a new application and add it to our Snaps environment. Or, now that you know

that **MyProgram** is the app that runs automatically when the Snaps environment starts up, here's a tip that can make things easier: start by editing the content of the **MyProgram.cs** file. This way, the code you've created will run without you having to find and select the new app in the environment (like we just had to do to run the **Countdown** app).

Remember: as long as the class in **MyProgram.cs** is called `MyProgram`, this program will run first in the environment. When you finish building your new app in the **MyProgram.cs** file, you can copy and paste the program code into a new source file (a new **.cs** file), give that new source file a unique name, and rename the new program's class so that Visual Studio won't wave red lines at you and prevent your program from compiling. And, at this point, if you really want the **MyProgram** app to function as it has in these first three chapters—setting the same title and displaying the same message we've seen in these chapters—you know how to get it back to that state.

Is it obvious now why I've called this source file **MyProgram.cs**? It's ready for you to use to build lots of programs!

MAKE SOMETHING HAPPEN

Build a Countdown announcer

This "Make Something Happen" is quite momentous. It represents a very important milestone on your journey toward programming enlightenment. Up until now you've been modifying or fixing existing programs, which is a great way to get started, but at some point you're going to have to create your own program from scratch. That time is now. If you think about it, even Bill Gates had to start somewhere. But I'm fairly sure that his first program wasn't able to speak to its users. Making computers speak was very difficult at the time Bill Gates was learning to write code, but he would have felt the same sense of excitement as you are about to.

After you build this application, you'll have written your first program. You can make the program more personal by using whatever messages you want to, and in the next section you'll discover some more Snaps that you can use to make the program even more interesting.

You should already have an "empty" app named **Countdown**. At the moment it does almost nothing—it only sets a title string in the state we last saw it—but now you're going to write your own statements to give it life. You can use the `SpeakString`, `DisplayString`, and `SetTitleString` methods provided by the `SnapsEngine` class to create your program.

All you have to do is make a program that counts down from 10 to 0. A clue: your program will contain at least 10 statements. Improve the program so that it displays the numbers on the screen as well as speaks them. This should double the number of statements in your program.

WHAT COULD GO WRONG

Compilation errors

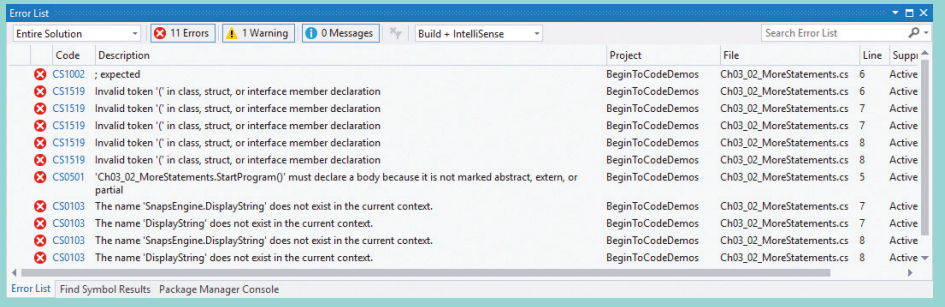
Before a program can run, it must be checked by the compiler. You can think of this process as a bit like the preflight checks performed on aircraft. Before a flight, the captain must walk around the plane, count the wings, ensure that all the tires have air in them, and be sure that the craft is safe to fly. In the same way, the compiler performs preflight checks on a program before it can run. If the program doesn't adhere to the rules of C#, the compiler will generate errors that you, the programmer, need to fix.

Unfortunately, the compiler is much pickier about errors than humans are. I can walk up to someone and ask "What you doing?" I'll get an answer, even though the question I asked is not properly formed English. However, if I try to compile the following program, I will get errors:

```
using SnapsLibrary;

public class BadBrackets
{
    public void StartProgram()
    (
        SnapsEngine.SpeakString("Hello world");
        SnapsEngine.SpeakString("Goodbye chickens");
    )
}
```

This code looks very similar to a program that we know works, but there are two tiny mistakes in the text. The bad news is that they generate 11 highly confusing errors, as shown in this screenshot.



The hard part about this state of affairs is that none of these messages actually tell you what you did wrong (and some of them look really scary). The compiler is a very clever program, but it's not smart enough to say, "You've used parentheses where you should have used curly

brackets." Update this code so that the statements are preceded by an open curly bracket ({) and followed by a closed curly bracket (}), and the program will run. When you mark the start and end of parts of a program, you must always use curly brackets. Parentheses are used for something else.

PROGRAMMER'S POINT

A good programmer has to be able deal with details

Humans are incredibly good at dealing with noise. We can pick out our name from background chatter and recognize our mother in a sea of faces. Computer programs have to work very hard to extract meaning from data. The compiler will become confused by one tiny, incorrect detail in a program. This means that to become a great programmer, you'll need to learn how to examine things in great detail, in some cases character by character, to discover what is wrong with them.

The best way to deal with mistakes like this is, of course, not to make them in the first place. But because we are human, this is impossible. Here are my tips for dealing with compilation errors:

1. Start from a program that *compiles*, or runs successfully. (Remember: compilers take our high-level code and generate the machine code that enables a computer to perform the actions we want it to perform. This is why we say that a program that runs successfully without errors compiles.) Visual Studio provides software wizards that can be used to make a program that doesn't do much but that does compile.
2. Compile often (in Visual Studio with the run button). If the number of changes you have made since the last successful compilation is small, you can isolate the error to just a few places.
3. Look for the three classic compilation mistakes:
   a. Missing something—for example, not putting a semicolon at the end of a statement.
   b. Using the wrong character—for example, using [rather than }.
   c. Spelling something incorrectly—for example, writing "startProgram" rather than "StartProgram". In the world of C#, it matters whether you use capital letters or lowercase letters.
4. Don't expect the error to be where the compiler has detected it. Some mistakes—for example, a missing curly bracket—may be detected many lines further down the program.
5. Use the color highlighting to help you. Words that are part of C# are shown in blue. Strings of text are red. If a word is not the color you think it should be, you might have typed it incorrectly.
6. Fix all the errors that you can see, and then compile again. Sometimes the compiler becomes confused and reports errors on lines that are sensible. Once you have fixed all the errors you can see, compile again and see if that works.

7. Use Undo and Redo. Visual Studio contains a very powerful editor with an Undo button (or Ctrl+Z) and a Redo button (Ctrl+Y), which you can use to step backward and forward through the changes you have made to your code. You can use these commands and the wavy red lines Visual Studio uses to highlight errors to find out where the mistakes are.

CODE ANALYSIS

Find the compilation errors

This program produces 20 errors when it is compiled. See if you can find all the mistakes.

```
using SnapsLibrary;

public Class MyProgram
{
    public void StartProgram()
    {
        SnapsEngine.SetTitleString("Begin to Code with C#");
        SnapsEngine.DisplayString(Welcome to the world of Snaps");
    }
}
```

Here are the errors:

```
using SnapsLibrary;

public Class MyProgram
{
    public void StartProgram()
    {
        SnapsEngine.SetTitleString("Begin to Code with C#");

        SnapsEngine.DisplayString(Welcome to the world of Snaps");
    }
}
```

Class should use a lowercase c

Missing the double quotation mark before Welcome

If you fix these two mistakes, we have a program that compiles just fine.

Extra Snaps

At the end of some chapters, I will introduce extra Snaps that you can play with. You can use these in your programs just like you used the `SpeakString` Snap earlier.

Delay

You might want to make your program delay for a while with the `Delay` Snap:

```
using SnapsLibrary;

class Ch03_06_TenSecondTimer
{
    public void StartProgram()
    {
        SnapsEngine.DisplayString("Start");
        SnapsEngine.Delay(10);
        SnapsEngine.DisplayString("End");
    }
}
```

Delays 10 seconds

This program displays "Start", pauses for 10 seconds, and then displays "End". The `Delay` method is different from `DisplayString` in the type of data you provide to it. You give the `DisplayString` method the string that you want the program to display. You give the `Delay` method the number of seconds you want the program to pause. This number can be a fraction if you want the program to pause for less than a second:

```
SnapsEngine.SpeakString("Tick");
SnapsEngine.Delay(0.5);
SnapsEngine.SpeakString("Tock");
```

Delays the program for half a second

You can use `Delay` to make a program look like it is thinking about something or to give the user time to read some information on the screen.

SetTextColor

This Snap lets you set the color of the text in the message on the screen:

```
using SnapsLibrary;

class Ch03_07_BlueText
{
    public void StartProgram()
    {
        SnapsEngine.SetTextColor(SnapsColor.Blue);
        SnapsEngine.DisplayString("Blue Monday");
    }
}
```

Built-in Snaps color that represents the color blue

You can also call this method to change the color of the text already on the screen.

```
using SnapsLibrary;

class Ch03_08_DelayedBlueText
{
    public void StartProgram()
    {
        SnapsEngine.DisplayString("Blue Monday");
        SnapsEngine.Delay(2);
        SnapsEngine.SetTextColor(SnapsColor.Blue);
    }
}
```

This program displays "Blue Monday" in the default color to start with. After two seconds, it changes the text's color to blue.

SetTitleColor

This Snap lets you set the color of the text in the title message on the screen:

```
using SnapsLibrary;

class Ch03_09_GreenSystemStarting
{
    public void StartProgram()
    {
        SnapsEngine.SetTitleColor(SnapsColor.Green);
```

```
        SnapsEngine.SetTitleString("System Starting");
    }
}
```

This program sets the title text to green and then displays "System Starting" as the title of the page. Generally, it's best to set the color of titles and messages before they are displayed; otherwise, they will "flick" into the requested color once they come into view. Reverse the order of the statements in **Ch03_09_GreenSystemStarting.cs** to see what I mean. This effect was minimized in **Ch03_08_DelayedBlueText.cs** because of the delay.

SetBackgroundColor

This Snap lets you set the background color of the screen. You can use this to indicate alarms or other conditions.

```
using SnapsLibrary;

class Ch03_10_RedScreen
{
    public void StartProgram()
    {
        SnapsEngine.SetBackgroundColor(SnapsColor.Red);
    }
}
```

Creating your own colors

The Snaps library includes a number of built-in colors that you can use in your programs. You can see these `SnapsColor` values in the examples we've been looking at: `SnapsColor.Blue`, `SnapsColor.Green`, and `SnapsColor.Red`. However, you might want to use colors that are not in the library. For example, I like the color lilac. When you describe a color to a computer, you have to use numbers because, as we know, computers only really work with numeric values. To describe a particular color, we can use three values: the amount of red, the amount of green, and the amount of blue in that color. In the case of Snaps (and lots of other computer platforms, including Windows), each of the numbers that describes a color level is in the range 0 to 255.

You can go online and look up the amount of red, green, and blue in particular colors. It turns out that lilac is made up of 200 red, 162 green, and 200 blue. Here's how you use these kinds of values in the Snaps that deal with colors:

```
using SnapsLibrary;

class Ch03_11_LilacScreen
{
    public void StartProgram()
    {
        SnapsEngine.SetBackgroundColor(red:200,green:162,blue:200);
    }
}
```

Amount of red, green, and blue to make the color lilac.

The `SetBackbroundColor` method can be given one or three items to work on. It can be given one `SnapsColor` value, or it can be given values for red, green, and blue. Each of the color intensity values are identified by name, which makes it easier for the programmer to see which of the values is being used for which purpose.

When a method is designed, the programmer has to decide how much information the method needs to do its work and what form the information should take. In the case of `SetBackgroundColor`, this version of the method needs to be told the amount of red, green, and blue to be used. The items supplied to the method are given as a list in which each item is separated from the next by a comma. If you omit an item or list too many, the compiler will complain when it tries to create the program.

```
SnapsEngine.SetBackgroundColor(red:255,green:255);
```

```
Error  1    No overload for method 'SetBackgroundColor' takes 2 arguments
```

The compiler doesn't like this statement because `SetBackgroundColor` in the Snaps library hasn't been created (by me) to accept only two items.

WHAT COULD GO WRONG

Bad color schemes

You will not get any errors if you write a program that displays red text on a red background, but what will your program's users say? I personally like using default colors (that is, the ones that you get when you start the program running). If you want to show your creative side, you

can pick other colors, but make sure you test your color scheme on many different devices because some machines can display colors much better than others. You should also make sure to check your proposed color scheme with your customer, if you have one, because colors are one thing that customers have very strong opinions about. Also, different people see different colors with varying degrees of success. Don't assume that others see colors the way you do!

MAKE SOMETHING HAPPEN

Build an egg timer

You can now use your programming skills to make a program that will time how long to cook an egg. By using the `Delay` method from the Snaps library, you can make the program pause while the egg is cooking and then announce when the egg is ready. My tests indicate that to get a perfect egg, you should cook it for five minutes (or 300 seconds). This code serves as a good starting point—copy this code rather than copying or editing **MyProgram.cs** when you make your egg timer:

```
using SnapsLibrary;

class Ch03_12_EggTimerStart
{
    public void StartProgram()
    {
        SnapsEngine.SetTitleString("Egg Timer");
        SnapsEngine.DisplayString("There are five minutes left");
        SnapsEngine.Delay(60);
        SnapsEngine.DisplayString("There are four minutes left");
    }
}
```

I think this is another important milestone for you as a developer. Unlike the countdown timer you created before, this program has all the makings of a proper product. Your mom would find this program useful. The Windows Store has quite a few products that work as timers, and there's no reason why a timer that you've made could not be one of them.

You could add extra features to your timer to do things like change the screen color when the egg is nearly ready and even provide a 30-second warning before the timer expires—and maybe a "ten, nine, eight" style countdown right at the end. You could also make the timer speak how much time is left as well as display it.

You can also use this design to make timers that could be useful in lots of other situations. Here are four that I can think of:

- Your best friend has discovered a passion for developing her own photographs and wants a timer she can use in the dark. The timer should just announce how many seconds have gone by every five seconds.
- You and your coworkers have started a quiz club and want to control how long each team has to answer a question. Each team gets 10 seconds.
- Your brother has a game where each player has to use a toothpick to eat as many baked beans as they can in thirty seconds (I didn't say it was a sensible game), and he needs a timer for that.
- Your mom is into exercise and needs something to time each stage of her workout and tell her what the next activity is. There are five activities: jogging in place, push-ups, jumping jacks, stand and sit, and squat thrusts. Each activity should be performed for 30 seconds, followed by a 10-second rest period.

Try your hand at making these timers and any other ones that you might think of. In the next chapter, you'll discover how a program can get input from a user so that you can make even better timers that allow the user to set the length of time the timer should run.

What you have learned

In this chapter, you've become more familiar with the Visual Studio environment in which you're creating your programs. You've seen that a C# program is expressed as a sequence of statements that are performed in order when the program runs. You've also seen the high-level C# that you and I have written converted into lower-level computer instructions by a program called the compiler. Sometimes the code has compiled, so the program runs successfully, and sometime the code hasn't compiled because of errors.

You've seen that the compiler ensures that the program conforms to the rules of the C# language. The compiler will reject programs that don't have statements that are completely correct. Whereas a human reader will tolerate missing or incorrect punctuation, the compiler will reject anything that does not obey the rules of the programming language.

The programs that we have written so far make use of a set of Snaps provided by the Snaps library that let us do things such as speak messages, display colors, and delay the execution of the program for a while. These components are provided as methods

that are passed data to tell them what to do. For example, the `SpeakString` method is given the text of the string that is to be spoken.

Here are some questions you might like to ponder about programs, statements, and compilers.

Does the user of the program need to have a copy of Visual Studio to run the program?

No. Visual Studio can produce a program file that users can run without Visual Studio.

Do I have to know how every Visual Studio command works?

No. You can get along by working with just a few of the buttons to start with. You will discover more features as you go through the book.

Is the compiler incompetent because it is confused by invalid program code?

You might think that the compiler is a bit silly, because sometimes it does things like complain when it has seen the wrong character. You would be forgiven for wondering why the compiler doesn't just substitute the right character and keep going. However, it turns out there is a very good reason for the compiler not to do this. If the compiler inserts things that it thinks are missing, it is making an assumption about what you, the programmer, were actually trying to do. We have already seen that assumptions are dangerous. It is much safer for the compiler to insist that you express exactly and correctly what you want the program to do.

Can any C# method accept any number of things to tell it what to do?

No. Each method is custom-made to accept a specific set of information. The `Delay` method needs to be told how many seconds to delay for. The `SpeakString` method needs a string of text to speak. The compiler knows what a method was built to accept, and it will feed only that kind of data into it. If you attempt to feed a string to `Delay`, the program will not compile.

Are the statements in a program always performed in the order they are written in the program?

Yes. You can think of a program as a story or recipe or a sequence of instructions. It would be meaningless for the steps to be performed in any order other than the one that has been set out.

Are the Snaps part of the C# language?

No. The Snaps library and these methods have been provided to help you learn how to program and to create simple applications. They are not part of C#, but they were created with C#. You will learn about other library classes and methods supplied with C# a little later in the book.

Index

Symbols and numbers

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X