Think Sharp

How to Think Like a Computer Scientist

Allen B. Downey

Peter J. Knaggs

5.1.2 / 0.3

Preface

> "As we enjoy great Advantages from the Inventions of others, we should be glad of an Opportunity to serve others by any Invention of ours, and this we should do freely and generously."
>
> —Benjamin Franklin, quoted in *Benjamin Franklin* by Edmund S. Morgan.

Why I wrote this book

This is the fifth edition of a book I started writing in 1999, when I was teaching at Colby College. I had taught an introductory computer science class using the Java programming language, but I had not found a textbook I was happy with. For one thing, they were all too big! There was no way my students would read 800 pages of dense, technical material, even if I wanted them to. And I didn't want them to. Most of the material was too specific—details about any particular language and its libraries that would be obsolete by the end of the semester, and that obscured the material I really wanted to get to.

The other problem I found was that the introduction to object-oriented programming was too abrupt. Many students who were otherwise doing well just hit a wall when we got to objects, whether we did it at the beginning, middle or end.

So I started writing. I wrote a chapter a day for 13 days, and on the 14th day I edited. Then I sent it to be photocopied and bound. When I handed it out on the first day of class, I told the students that they would be expected to read one chapter a week. In other words, they would read it seven times slower than I wrote it.

The philosophy behind it

Here are some of the ideas that make the book the way it is:

- Vocabulary is important. Students need to be able to talk about programs and understand what I am saying. I try to introduce the minimum number of terms, to define them carefully when they are first used, and to organize them in glossaries at the end of each chapter. In my class, I include vocabulary questions on quizzes and exams, and require students to use appropriate terms in short-answer responses.

- To write a program, students have to understand the algorithm, know the programming language, and they have to be able to debug. I think too many books neglect debugging. This book includes an appendix on debugging and an appendix on program development (which can help avoid debugging). I recommend that students read this material early and come back to it often.

- Some concepts take time to sink in. Some of the more difficult ideas in the book, like recursion, appear several times. By coming back to these ideas, I am trying to give students a chance to review and reinforce or, if they missed it the first time, a chance to catch up.

- I try to use the minimum amount of C# to get the maximum amount of programming power. The purpose of this book is to teach programming and some introductory ideas from computer science, not any language in particular. I left out some language features, like the `switch` statement, that are unnecessary, and avoided most of the libraries.

The minimalism of my approach has some advantages. Each chapter is about ten pages, not including the exercises. In my classes I ask students to read each chapter before we discuss it, and I have found that they are willing to do that and their comprehension is good. Their preparation makes class time available for discussion of the more abstract material, in-class exercises, and additional topics that aren't in the book.

But minimalism has some disadvantages. There is not much here that is intrinsically fun. Most of my examples demonstrate the most basic use of a language feature, and many of the exercises involve string manipulation and mathematical ideas. I think some of them are fun, but many of the things that excite students about computer science, like graphics, sound and network applications, are given short shrift.

The problem is that many of the more exciting features involve lots of details and not much concept. Pedagogically, that means a lot of effort for not much payoff. So there is a tradeoff between the material that students enjoy and the material that is most intellectually rich. I leave it to individual teachers to find the balance that is best for their classes. To help, the book includes appendices that cover graphics, keyboard input and file input.

Object-oriented programming

Some books introduce objects immediately; others warm up with a more procedural style and develop object-oriented style more gradually. This book uses the "objects late" approach.

Many of the object-oriented features are motivated by problems with previous languages, and their implementations are influenced by this history. Some of these features are hard to explain if students aren't familiar with the problems they solve.

It wasn't my intention to postpone object-oriented programming. On the contrary, I got to it as quickly as I could, limited by my intention to introduce concepts one at a time, as clearly as possible, in a way that allows students to practice each idea in isolation before adding the next. But I have to admit that it takes some time to get there.

Free books!

Since the beginning, this book has under a license that allows users to copy, distribute and modify the book. Readers can download the book in a variety of formats and read it on screen or print it. Teachers are free to print as many copies as they need. And anyone is free to customize the book for their needs.

People have translated the book into other computer languages (including Python, Eiffel and now C#), and other natural languages (including Spanish, French and German). Many of these derivatives are also available under free licenses.

Motivated by Open Source Software, I adopted the philosophy of releasing the book early and updating it often. I do my best to minimize the number of errors, but I also depend on readers to help out.

The response has been great. I get messages almost every day from people who have read the book and liked it enough to take the trouble to send in a "bug report." Often I can correct an error and post an updated version within a few minutes. I think of the book as a work in progress, improving a little whenever I have time to make a revision, or when readers send feedback.

Oh, the title

I get a lot of grief about the title of the book. Not everyone understands that it is— mostly—a joke. Reading this book will probably not make you think like a computer scientist. That takes time, experience, and probably a few more classes.

But there is a kernel of truth in the title: this book is not about any particular programming language, and it is only partly about programming. If it is successful, this book is about a way of thinking. Computer scientists have an approach to problem-solving, and a way of crafting solutions, that is unique, versatile and powerful. I hope that this book gives you a sense of what that approach is, and that at some point you will find yourself thinking like a computer scientist.

Allen Downey
Needham, Massachusetts
July 13, 2011

Contributors List

When I started writing free books, it didn't occur to me to keep a contributors list. When Jeff Elkner suggested it, it seemed so obvious that I am embarrassed by the omission. This list starts with the 4th Edition, so it omits many people who contributed suggestions and corrections to earlier versions.

If you have additional comments, please send them to:
feedback@greenteapress.com

- Ellen Hildreth used this book to teach Data Structures at Wellesley College, and she gave me a whole stack of corrections, along with some great suggestions.

- Tania Passfield pointed out that the glossary of Chapter 4 has some leftover terms that no longer appear in the text.

- Elizabeth Wiethoff noticed that my series expansion of $\exp(-x^2)$ was wrong. She is also working on a Ruby version of the book!

- Matt Crawford sent in a whole patch file full of corrections!

- Chi-Yu Li pointed out a typo and an error in one of the code examples.

- Doan Thanh Nam corrected an example in Chapter 3.

- Stijn Debrouwere found a math typo.

- Muhammad Saied translated the book into Arabic, and found several errors.

- Marius Margowski found an inconsistency in a code example.

- Guy Driesen found several typos.

- Leslie Klein discovered yet another error in the series expansion of $\exp(-x^2)$, identified typos in the card array figures, and gave helpful suggestions to clarify several exercises.

Finally, I wish to acknowledge Chris Mayfield for his significant contribution to version 5.1 of this book. His careful review lead to over one hundred corrections and improvements throughout. Several new features include embedded hypertext links and cross references, consistent layout of all exercises, and syntax highlighting in code examples.

Contents

3   Void methods   23

4   Conditionals and recursion   35

Chapter 1

The way of the program

The goal of this book is to teach you to think like a computer scientist. I like the way computer scientists think because they combine some of the best features of Mathematics, Engineering, and Natural Science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behaviour of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem-solving**. By that I mean the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called "The way of the program."

On one level, you will be learning to program, which is a useful skill by itself. On another level you will use programming as a means to an end. As we go along, that end will become clearer.

1.1   What is a programming language?

The programming language you will be learning is C#, which is relatively new (Microsoft released the first version in 2000). C# is an example of a **high-level language**; other high-level languages you might have heard of are Java, C or C++, and Perl.

As you might infer from the name "high-level language," there are also **low-level languages**, sometimes called machine language or assembly language. Loosely-speaking, computers can only run programs written in low-level languages. Thus, programs written in a high-level language have to be translated before they can run. This translation takes time, which is a small disadvantage of high-level languages.

The advantages are enormous. First, it is *much* easier to program in a high-level language: the program takes less time to write, it's shorter and easier to read, and it's more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer, and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are only used for a few special applications.

There are two ways to translate a program; **interpreting** and **compiling**. An interpreter is a program that reads a high-level program and does what it says. In effect, it translates the program line-by-line, alternately reading lines and carrying out commands.

A compiler is a program that reads a high-level program and translates it all at once, before running any of the commands. Often you compile the program as a separate step, and then run the compiled code later. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.

C# is both compiled and interpreted. Instead of translating programs into machine language, the compiler generates something called Microsoft Intermediate Language (MSIL) which is a **byte code**. Byte code is easy (and fast) to interpret, like machine language, but it is also portable, like a high-level language. Thus, it is possible to compile a program on one machine, transfer the byte code to another machine, and then interpret the byte code on the other machine. This ability is an advantage of C# over many other high-level languages.
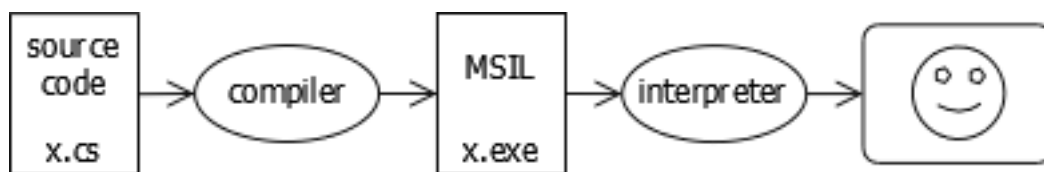
The compiler read the source code and generates Microsoft Intermediate Language (MSIL)

The .Net runtime reads and interprets the MSIL and the result appears on the screen

Although this process may seem complicated, in most program development environments these steps are automated for you. Usually you will only have to write a program and press a button or type a single command to compile and run it. On the other hand, it is useful to know what steps are happening in the background, so if something goes wrong you can figure out what it is.

1.2   What is a program?

A program is a sequence of instructions that specifies how to perform a computation[1]. The computation might be something mathematical, like solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program.

The instructions, which we will call **statements**, look different in different programming languages, but there are a few basic operations most languages perform:

input: Get data from the keyboard, or a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

testing: Check for certain conditions and run the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

That's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of statements that perform these operations. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic operations.

1.3   What is debugging?

For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

There are a three kinds of errors that can occur in a program, and it is useful to distinguish them to track them down more quickly.

1.3.1   Syntax errors

The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. **Syntax** refers to the structure of your program and the rules about that structure.

For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

---

[1]This definition does not apply to all programming languages; for alternatives, see http://en.wikipedia.org/wiki/Declarative_programming.

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e e cummings without spewing error messages.

Compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program.

To make matters worse, there are more syntax rules in C# than there are in English, and the error messages you get from the compiler are often not very helpful. During the first weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

1.3.2 Run-time errors

The second type of error is a run-time error, so-called because the error does not appear until you run the program. In C#, run-time errors occur when the interpreter is running the byte code and something goes wrong.

C# tends to be a **safe** language, which means that the compiler catches a lot of errors. So run-time errors are rare, especially for simple programs.

In C#, run-time errors are called **exceptions**, and in most environments they appear as windows or dialog boxes that contain information about what happened and what the program was doing when it happened. This information is useful for debugging.

1.3.3 Logic errors and semantics

The third type of error is the **logic** or **semantic** error. If there is a logic error in your program, it will compile and run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The semantics, or meaning of the program, are wrong. Identifying logic errors can be tricky because you have to work backwards, looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you will acquire in this class is debugging. Although debugging can be frustrating, it is one of the most interesting, challenging, and valuable parts of programming.

Debugging is like detective work. You are confronted with clues and you have to infer the processes and events that lead to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (From A. Conan Doyle's *The Sign of Four*.)

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should always start with a working program that does *something*, and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux" (from *The Linux Users' Guide* Beta Version 1).

In later chapters I make more suggestions about debugging and other programming practices.

1.4 Formal and natural languages

Natural languages are the languages that people speak, like English, Spanish, and French. They were not designed by people (although people try to impose order on them); they evolved naturally.

Formal languages are languages designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> **Programming languages are formal languages that have been designed to express computations.**

Formal languages have strict rules about syntax. For example, $3 + 3 = 6$ is a syntactically correct mathematical statement, but $3\$ =$ is not. Also, H_2O is a syntactically correct chemical name, but $_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and numbers and chemical elements. One of the problems with $3\$ =$ is that $\$$ is not a legal token in mathematics (at least as far as I know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax rule pertains to the structure of a statement; that is, the way the tokens are arranged. The statement 3$ = is structurally illegal, because you can't have an equals sign at the end of an equation. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this unconsciously). This process is called **parsing**.

Although formal and natural languages have features in common—tokens, structure, syntax and semantics—there are differences.

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: To make up for ambiguity and reduce misunderstandings, natural languages are often redundant. Formal languages are more concise.

literalness: Natural languages are full of idiom and metaphor. Formal languages mean exactly what they say.

People who grow up speaking a natural language (everyone) often have a hard time adjusting to formal languages. In some ways the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is common and deliberate.

Prose: The literal meaning of words is more important and the structure contributes more meaning.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, remember that the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.5 The first program

Traditionally the first program people write in a new language is called "hello world" because all it does is display the words "Hello, World." In C#, this program looks like:

```
1  using System;
2
3  namespace ThinkSharp {
4    public class Hello {
5      // main: generate some simple output
6      public static void Main(string[] args) {
7        Console.WriteLine("Hello, world.");
8      }
9    }
10 }
```

This program includes features that are hard to explain to beginners, but it provides a preview of topics we will see in detail later. Note that the line numbers (on the left) are given so we can refer to a specific line, they are not part of the program.

C# programs are made up of **name spaces** and **classes**. A name space is a way of separating names so that different programs can use the same name without getting confused. Your program should all ways be included in a name space. The name space is normally the same name as your program, just without any spaces. So a **name space definition** has the form:

```
namespace ⟨NameSpace⟩ {
    ...
}
```

Here ⟨NameSpace⟩ indicates a name chosen by the programmer. (Lines 3 and 10 from the example.)

A name space is made up of **class definitions**, which have the form:

```
class ⟨ClassName⟩ {
    ...
}
```

Here ⟨ClassName⟩ indicates a name chosen by the programmer. The class name in the example is Hello. (Lines 4 and 9 of our example.)

In turn, a class is made up of a number of **method definitions**. In this case:

```
public static void Main (string[] args) {
    ⟨Statments⟩
}
```

Main is a **method**, which is a named collection of statements. The name Main is special; it marks the place in the program where execution begins. When the program runs, it starts

at the first statement in `Main` and ends when it finishes the last statement.

`Main` can have any number of statements, but the example has one. It is a **write statement**, meaning that it displays a message on the screen. Confusingly, "write" can mean "display something on the screen," "send something to the printer," or "send something to a file." In this book I won't say much about sending things anywhere other than the screen. The write statement ends with a semi-colon (;).

`Console`.`WriteLine` is a method provided by one of C#'s libraries. A **library** is a collection of class and method definitions. In this case the `WriteLine` method from the `Console` class. The `Console` class is defined in the `System` name space, which is why we had to tell the system we are `using` the `System` name space (line 1).

C# uses squiggly-braces (`{` and `}`) to group things together. The outermost squiggly-braces (lines 3 and 10) contain the name space definition, and the inner braces (lines 6 and 8) contain the definition of `Main`.

Line 5 begins with `//`. That means it's a **comment**, which is a bit of English text that you can put in a program, usually to explain what it does. When the compiler sees `//`, it ignores everything from there until the end of the line.

When running the program from an IDE (pressing the F5 key), the program will complete and return to IDE before you have a chance to look at the results. Add the following line, just before the close brace on line 8:

```
Console.ReadLine();
```

This will cause the execution to wait until you press the ENTER or RETURN key before returning to the IDE.

1.6   Glossary

problem-solving: The process of formulating a problem, finding a solution, and expressing the solution.

high-level language: A programming language like C# that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to run. Also called "machine language" or "assembly language."

formal language: Any of the languages people have designed for specific purposes, like representing mathematical ideas or computer programs. All programming languages are formal languages.

natural language: Any of the languages people speak that have evolved naturally.

portability: A property of a program that can run on more than one kind of computer.

interpret: To run a program in a high-level language by translating it one line at a time.

compile: To translate a program in a high-level language into a low-level language, all at once, in preparation for later execution.

source code: A program in a high-level language, before being compiled.

object code: The output of the compiler, after translating the program.

executable: Another name for object code that is ready to run.

byte code: A special kind of object code used for C# programs. Byte code is similar to a low-level language, but it is portable, like a high-level language.

statement: A part of a program that specifies a computation.

write statement: A statement that causes output to be displayed on the screen.

comment: A part of a program that contains information about the program, but that has no effect when the program runs.

method: A named collection of statements.

library: A collection of class and method definitions.

bug: An error in a program.

syntax: The structure of a program.

semantics: The meaning of a program.

parse: To examine a program and analyze the syntactic structure.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to compile).

exception: An error in a program that makes it fail at run-time. Also called a run-time error.

logic error: An error in a program that makes it do something other than what the programmer intended.

debugging: The process of finding and removing any of the three kinds of errors.

1.7 Exercises

Exercise 1.1. Computer scientists have the annoying habit of using common English words to mean something other than their common English meaning. For example, in English, statements and comments are the same thing, but in programs they are different.

The glossary at the end of each chapter is intended to highlight words and phrases that have special meanings in computer science. When you see familiar words, don't assume that you know what they mean!

1. In computer jargon, what's the difference between a statement and a comment?

2. What does it mean to say that a program is portable?

3. What is an executable?

Exercise 1.2. Before you do anything else, find out how to compile and run a C# program (read Appendix A).

1. Type in the "Hello, world" program, then compile and run it.

2. Add a write statement that prints a second message after the "Hello, world!". Something witty like, "How are you?" Compile and run the program again.

3. Add a comment to the program (anywhere), recompile, and run it again. The new comment should not affect the result.

This exercise may seem trivial, but it is the starting place for many of the programs we will work with. To debug with confidence, you have to have confidence in your programming environment. In some environments, it is easy to lose track of which program is executing, and you might find yourself trying to debug one program while you are accidentally running another. Adding (and changing) write statements is a simple way to be sure that the program you are looking at is the program you are running.

Exercise 1.3. It is a good idea to commit as many errors as you can think of, so that you see what error messages the compiler produces. Sometimes the compiler tells you exactly what is wrong, and all you have to do is fix it. But sometimes the error messages are misleading. You will develop a sense for when you can trust the compiler and when you have to figure things out yourself.

1. Remove one of the open squiggly-braces.

2. Remove one of the close squiggly-braces.

3. Instead of `Main`, write `Mian`.

4. Remove the word `static`.

5. Remove the word `public`.

6. Remove the word `Console`.

7. Replace `WriteLine` with `Writeline`.

8. Replace `WriteLine` with `Write`. This one is tricky because it is a logic error, not a syntax error. The statement `Console`.`Write` is legal, but it may or may not do what you expect.

9. Delete one of the parentheses. Add an extra one.

10. Delete the semi-colon. Add an extra one.

Chapter 2

Variables and types

2.1 More writing

You can put as many statements as you want in `Main`; for example, to print more than one line:

```
1  using System;
2
3  namespace ThinkSharp {
4    public class Program {
5      // Generates some simple output.
6      public static void Main(string[] args) {
7        Console.WriteLine("Hello, world."); // print one line
8        Console.WriteLine("How are you?");  // print another
9      }
10   }
11 }
```

As this example demonstrates, you can put comments at the end of a line, as well as on a line by themselves.

The phrases that appear in quotation marks are called **strings**, because they are made up of a sequence (string) of characters. Strings can contain any combination of letters, numbers, punctuation marks, and other special characters.

`WriteLine` is so called because, after each line it adds a special character, called a **newline**, that moves the cursor to the next line of the display. The next time `WriteLine` is invoked, the new text appears on the next line.

To display the output from multiple print statements all on one line, use `Write`:

```
1  using System;
2
3  namespace HelloWorld {
4     public class Program {
5        // Generates some simple output.
6        public static void Main(string[] args) {
7           Console.Write("Goodbye, ");
8           Console.WriteLine("cruel world!");
9        }
10    }
11 }
```

The output appears on a single line as "Goodbye, cruel world!". There is a space between the word "Goodbye" and the second quotation mark. This space appears in the output, so it affects the behaviour of the program.

Spaces that appear outside of quotation marks generally do not affect the behaviour of the program. For example, I could have written:

```
1  using System;
2  namespace HelloWorld {
3  public class Program {
4  public static void Main(string[] args) {
5  Console.Write("Goodbye, ");
6  Console.WriteLine("cruel world!");
7  }
8  }
9  }
```

This program would compile and run just as well as the original. The breaks at the ends of lines (newlines) do not affect the program's behavior either, so I could have written:

```
1  using System; namespace HelloWorld { public class Program { public static
2  void Main(string[] args) { Console.Write("Goodbye, "); Console.WriteLine
3  ("cruel world!"); }}}
```

That would work, too, but the program is getting harder and harder to read. Newlines and spaces are useful for organizing your program visually, making it easier to read the program and locate errors.

2.2   Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a named location that stores a **value**. Values are things that can be written, stored and (as we'll see later) operated on. The strings we have been writing ("Hello, World.", "Goodbye, ", etc.) are values.

To store a value, you have to create a variable. Since the values we want to store are strings, we declare that the new variable is a string:

```
string bob;
```

This statement is a **declaration**, because it declares that the variable named `bob` has the type `string`. Each variable has a type that determines what kind of values it can store. For example, the `int` type can store integers, and the `string` type can store strings.

Some types begin with a capital letter and some with lower-case. We will learn the significance of this distinction later, but for now you should take care to get it right. There is no such type as `Int`, and the compiler will object if you try to make one up.

To create an integer variable, the syntax is `int bob;`, where `bob` is the arbitrary name you made up for the variable. In general, you will want to make up variable names that indicate what you plan to do with the variable. For example, if you saw these variable declarations:

```
string firstName;
string lastName;
int hour, minute;
```

you could guess what values would be stored in them. This example also demonstrates the syntax for declaring multiple variables with the same type: `hour` and `minute` are both integers (`int` type).

2.3 Assignment

Now that we have created variables, we want to store values. We do that with an **assignment statement**.

```
bob = "Hello.";      // give bob the value "Hello."
hour = 11;           // assign the value 11 to hour
minute = 59;         // set minute to 59
```

This example shows three assignments, and the comments show three different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

- When you declare a variable, you create a named storage location.

- When you make an assignment to a variable, you give it a value.

A common way to represent variables on paper is to draw a box with the name of the variable on the outside and the value of the variable on the inside. This figure shows the effect of the three assignment statements:

bob	"Hello."
hour	11
minute	59

As a general rule, a variable has to have the same type as the value you assign it. You cannot store a `string` in `minute` or an integer in `bob`.

On the other hand, that rule can be confusing, because there are many ways that you can convert values from one type to another, and sometimes things are converted automatically. For now you should remember the general rule and we'll talk about exceptions later.

Another source of confusion is that some strings *look* like integers, but they are not. For example, `bob` can contain the string `"123"`, which is made up of the characters 1, 2 and 3, but that is not the same thing as the *number* 123.

```
bob = "123";   // legal
bob = 123;     // not legal
```

2.4   Printing variables

You can print the value of a variable using `WriteLine` or `Write`:

```
using System;

namespace ThinkSharp {
   public class Hello {
      public static void Main(string[] args) {
         string firstLine;
         firstLine = "Hello, again!";
         Console.WriteLine(firstLine);
      }
   }
}
```

This program creates a variable named `firstLine`, assigns it the value `"Hello, again!"` and then prints that value. When we talk about "printing a variable," we mean printing the *value* of the variable. To print the *name* of a variable, you have to put it in quotes. For example: `Console.WriteLine("firstLine");`

For example, you can write

```
string firstLine;
firstLine = "Hello, again!";
Console.Write("The value of firstLine is ");
Console.WriteLine(firstLine);
```

The output of this program is

```
The value of firstLine is Hello, again!
```

I am happy to report that the syntax for printing a variable is the same regardless of the variable's type.

```csharp
int hour, minute;
hour = 11;
minute = 59;
Console.Write("The current time is ");
Console.Write(hour);
Console.Write(":");
Console.Write(minute);
Console.WriteLine(".");
```

The output of this program is `The current time is 11:59.`

WARNING: To put multiple values on the same line, is common to use several `Write` statements followed by a `WriteLine`. But you have to remember the `WriteLine` at the end. In many environments, the output from `Write` is stored without being displayed until `WriteLine` is invoked, at which point the entire line is displayed at once. If you omit `WriteLine`, the program may terminate without displaying the stored output!

2.5 Keywords

A few sections ago, I said that you can make up any name you want for your variables, but that's not quite true. There are certain words that are reserved because they are used by the compiler to parse the structure of your program, and if you use them as variable names, it will get confused. These words, called **keywords**, they are generally displayed in another colour (such as blue) and include `public`, `class`, `void`, `int` and many more.

The complete list is available at from the Microsoft Developer Network (MSDN)[1]. This site, provides documentation for C# that I will be referring to throughout the book.

Rather than memorize the list, I suggest you take advantage of a feature of the IDEs: code highlighting. As you type, parts of your program should appear in different colours. For example, keywords in blue, strings in red, classes in light blue and other code black. If you type a variable name and it turns blue, watch out! You might get some strange behaviour from the compiler.

---

[1]http://msdn.microsoft.com/en-GB/library/x53a06bb.aspx

2.6   Operators

Operators are symbols used to represent computations like addition and multiplication. Most operators do what you expect them to do because they are common mathematical symbols. For example, the operator for addition is +. Subtraction is -, multiplication is * and division is /.

```
1 + 1      hour - 1      hour * 60 + minute      minute / 60
```

Expressions can contain both variable names and numbers. Variables are replaced with their values before the computation is performed.

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, this program:

```csharp
int hour, minute;
hour = 11;
minute = 59;
Console.Write("Number of minutes since midnight: ");
Console.WriteLine(hour * 60 + minute);
Console.Write("Fraction of the hour that has passed: ");
Console.WriteLine(minute / 60);
```

generates this output:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

The first line is expected, but the second line is odd. The value of `minute` is 59, and 59 divided by 60 is 0.98333, not 0. The problem is that C# is performing **integer division**.

When both **operands** are integers (operands are the things operators operate on), the result is also an integer, and by convention integer division always rounds *down*, even in cases like this where the next integer is so close.

An alternative is to calculate a percentage rather than a fraction:

```csharp
Console.Write("Percentage of the hour that has passed: ");
Console.Write(minute * 100 / 60);
```

The result is:

```
Percentage of the hour that has passed: 98
```

Again the result is rounded down, but at least now the answer is approximately correct. To get a more accurate answer, we can use a different type of variable, called floating-point, that can store fractional values. We'll get to that in the next chapter.

2.7   Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the rules of **precedence**. A complete explanation of precedence can get complicated,

but just to get you started:

- Multiplication and division happen before addition and subtraction. So `2 * 3 - 1` yields 5, not 4, and `2 / 3 - 1` yields -1, not 1 (remember that in integer division `2 / 3` is 0).

- If the operators have the same precedence they are evaluated from left to right. So in the expression `minute * 100 / 60`, the multiplication happens first, yielding `5900 / 60`, which in turn yields `98`. If the operations had gone from right to left, the result would be `59 * 1` which is `59`, which is wrong.

- Any time you want to override the rules of precedence (or you are not sure what they are) you can use parentheses. Expressions in parentheses are evaluated first, so `2 * (3 - 1)` is 4. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.

2.8 Operators for `Strings`

In general you cannot perform mathematical operations on `string`s, even if the strings look like numbers. The following are illegal (if we know that `bob` has type `string`)

<div align="center">

`bob - 1`   `"Hello" / 123`   `bob * "Hello"`

</div>

By the way, can you tell by looking at those expressions whether `bob` is an integer or a string? Nope. The only way to tell the type of a variable is to look at the place where it is declared.

Interestingly, the `+` operator *does* work with `string`s, but it might not do what you expect. For `string`s, the `+` operator represents **concatenation**, which means joining up the two operands by linking them end-to-end. So `"Hello, " + "world."` yields the string `"Hello, world."` and `bob + "ism"` adds the suffix *ism* to the end of whatever `bob` is, which is handy for naming new forms of bigotry.

2.9 Composition

So far we have looked at the elements of a programming language—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to multiply numbers and we know how to write them; it turns out we can combine them in a single statement:

```
Console.WriteLine(17 * 3);
```

Any expression involving numbers, strings and variables, can be used inside a write state-ment. We've already seen one example:

```
Console.WriteLine(hour * 60 + minute);
```

But you can also put arbitrary expressions on the right-hand side of an assignment state-ment:

```
int percentage;
percentage = (minute * 100) / 60;
```

This ability may not seem impressive now, but we will see examples where composition expresses complex computations neatly and concisely.

WARNING: The left side of an assignment has to be a *variable* name, not an expression. That's because the left side indicates the storage location where the result will go. Expres-sions do not represent storage locations, only values. So the following is illegal: `minute+1 = hour;`.

2.10   Glossary

variable: A named storage location for values. All variables have a type, which is declared when the variable is created.

value: A number or string (or other thing to be named later) that can be stored in a variable. Every value belongs to a type.

type: A set of values. The type of a variable determines which values can be stored there. The types we have seen are integers (`int`) and strings (`string`).

keyword: A reserved word used by the compiler to parse programs. You cannot use keywords, like `public`, `class` and `void` as variable names.

declaration: A statement that creates a new variable and determines its type.

assignment: A statement that assigns a value to a variable.

expression: A combination of variables, operators and values that represents a single value. Expressions also have types, as determined by their operators and operands.

operator: A symbol that represents a computation like addition, multiplication or string concatenation.

operand: One of the values on which an operator operates.

precedence: The order in which operations are evaluated.

concatenate: To join two operands end-to-end.

composition: The ability to combine simple expressions and statements into compound statements and expressions to represent complex computations concisely.

2.11 Exercises

Exercise 2.1. If you are using this book with a study partner, you might enjoy this exercise: find a partner and play "Stump the Chump":

Start with a program that compiles and runs correctly. One player turns away while the other player adds an error to the program. Then the first player tries to find and fix the error. You get two points if you find the error without compiling the program, one point if you find it using the compiler, and your opponent gets a point if you don't find it.

Exercise 2.2.

1. Create a new program named `Date.cs`. Copy or type in something like the "Hello, World" program and make sure you can compile and run it.

2. Following the example in Section 2.4, write a program that creates variables named `day`, `date`, `month` and `year`. `day` will contain the day of the week and `date` will contain the day of the month. What type is each variable? Assign values to those variables that represent today's date.

3. Print the value of each variable on a line by itself. This is an intermediate step that is useful for checking that everything is working so far.

4. Modify the program so that it prints the date in standard European form: `Saturday 16 July, 2011`.

5. Modify the program again so that the total output is:

```
American format:
Saturday, July 16, 2011
European format:
Saturday 16 July, 2011
```

The point of this exercise is to use string concatenation to display values with different types (`int` and `string`), and to practice developing programs gradually by adding a few statements at a time.

Exercise 2.3.

1. Create a new program called `Time.cs`. From now on, I won't remind you to start with a small, working program, but you should.

2. Following the example in Section 2.6, create variables named `hour`, `minute` and `second`, and assign them values that are roughly the current time. Use a 24-hour clock, so that at 2pm the value of `hour` is 14.

3. Make the program calculate and print the number of seconds since midnight.

4. Make the program calculate and print the number of seconds remaining in the day.

5. Make the program calculate and print the percentage of the day that has passed.

6. Change the values of `hour`, `minute` and `second` to reflect the current time (I assume that some time has elapsed), and check to make sure that the program works correctly with different values.

The point of this exercise is to use some of the arithmetic operations, and to start thinking about compound entities like the time of day that that are represented with multiple values. Also, you might run into problems computing percentages with `ints`, which is the motivation for floating point numbers in the next chapter.

HINT: you may want to use additional variables to hold values temporarily during the computation. Variables like this, that are used in a computation but never printed, are sometimes called intermediate or temporary variables.

Chapter 3

Void methods

3.1 Floating-point

In the last chapter we had some problems dealing with numbers that were not integers. We worked around the problem by measuring percentages instead of fractions, but a more general solution is to use floating-point numbers, which can represent fractions as well as integers. The floating-point type is called `double`, which is short for "double-precision."

You can create floating-point variables and assign values to them using the same syntax we used for the other types. For example:

```
double pi;
pi = 3.14159;
```

It is also legal to declare a variable and assign a value to it at the same time:

```
int x = 1;
string empty = "";
double pi = 3.14159;
```

This syntax is common; a combined declaration and assignment is sometimes called an **initialization**.

Although floating-point numbers are useful, they are a source of confusion because there seems to be an overlap between integers and floating-point numbers. For example, if you have the value 1, is that an integer, a floating-point number, or both?

C# distinguishes the integer value 1 from the floating-point value 1.0, even though they seem to be the same number. They belong to different types, and strictly speaking, you are not allowed to make assignments between types. For example, the following is illegal:

```
int x = 1.1;
```

because the variable on the left is an `int` and the value on the right is a `double`. But it is easy to forget this rule, especially because there are places where C# will automatically convert from one type to another. For example:

```
double y = 1;
```

should technically not be legal, but C# allows it by converting the `int` to a `double` automatically. This leniency is convenient, but it can cause problems; for example:

```
double y = 1 / 3;
```

You might expect the variable `y` to get the value `0.333333`, which is a legal floating-point value, but in fact it gets `0.0`. The reason is that the expression on the right is the ratio of two integers, so C# does *integer* division, which yields the integer value `0`. Converted to floating-point, the result is `0.0`.

One way to solve this problem (once you figure out what it is) is to make the right-hand side a floating-point expression:

```
double y = 1.0 / 3.0;
```

This sets `y` to `0.333333`, as expected.

The operations we have seen so far—addition, subtraction, multiplication, and division— also work on floating-point values, although you might be interested to know that the underlying mechanism is completely different. In fact, most processors have special hardware just for performing floating-point operations.

3.2   Converting from `double` to `int`

As I mentioned, `int`s are converted into `double`s automatically if necessary, because no information is lost in the translation. On the other hand, going from a `double` to an `int` requires rounding off. C# doesn't perform this operation automatically, in order to make sure that you, as the programmer, are aware of the loss of the fractional part of the number.

The simplest way to convert a floating-point value to an integer is to use a **typecast**. Typecasting is so called because it allows you to take a value that belongs to one type and "cast" it into another type (in the sense of moulding or reforming).

The syntax for typecasting is to put the name of the type in parentheses and use it as an operator. For example,

```
double pi = 3.14159;
int x = (int) pi;
```

The `(int)` operator has the effect of converting what follows into an integer, so `x` gets the value 3.

Typecasting takes precedence over arithmetic operations, so in the following example, the value of `pi` gets converted to an integer first, and the result is 60.0, not 62.

```
    double pi = 3.14159;
    double x = (int) pi * 20.0;
```

Converting to an integer always rounds down, even if the fraction part is 0.99999999. These behaviours (precedence and rounding) can make typecasting error-prone.

3.3   Math methods

In mathematics, you have probably seen functions like sin and log, and you have learned to evaluate expressions like $\sin(\pi/2)$ and $\log(1/x)$. First, you evaluate the expression in parentheses, which is called the **argument** of the function. Then you can evaluate the function itself, either by looking it up in a table or by performing various computations.

This process can be applied repeatedly to evaluate more complicated expressions like $\log(1/\sin(\pi/2))$. First we evaluate the argument of the innermost function, then evaluate the function, and so on.

C# provides functions that perform the most common mathematical operations. These functions are called **methods**. The math methods are invoked using a syntax that is similar to the `Write` statements we have already seen:

```
    double root = Math.Sqrt(17.0);
    double angle = 1.5;
    double height = Math.Sin(angle);
```

The first example sets `root` to the square root of 17. The second example finds the sine of the value of `angle`, which is `1.5`. C# assumes that the values you use with `sin` and the other trigonometric functions (`cos`, `tan`) are in *radians*. To convert from degrees to radians, you can divide by 360 and multiply by 2π. Conveniently, C# provides `Math.PI`:

```
    double degrees = 90;
    double angle = degrees * 2 * Math.PI / 360.0;
```

Notice that `PI` is in all capital letters. C# does not recognize `Pi`, `pi`, or `pie`.

Another useful method in the `Math` class is `Round`, which rounds a floating-point value off to the nearest integer and returns an `int`.

```
    int x = (int) Math.Round(Math.PI * 20.0);
```

In this case the multiplication happens first, before the method is invoked. The result is 63 (rounded up from 62.8319).

3.4   Composition

Just as with mathematical functions, methods can be **composed**, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a method:

```
    double x = Math.Cos(angle + Math.PI/2);
```

This statement takes the value `Math.PI`, divides it by two and adds the result to the value of the variable `angle`. The sum is then passed as an argument to `cos`. (PI is the name of a variable, not a method, so there are no arguments, not even the empty argument `()`).

You can also take the result of one method and pass it as an argument to another:

```
    double x = Math.Exp(Math.Log(10.0));
```

The `log` method always uses base e, so this statement finds the log base e of 10 and then raises e to that power. The result gets assigned to `x`; I hope you know what it is.

3.5   Adding new methods

So far we have used library methods, but it is also possible to add new methods. We have already seen one method definition: `Main`. The method named `Main` is special, but the syntax is the same for other methods:

```
public static void ⟨name⟩(⟨list of parameters⟩) {
    ⟨statements⟩
}
```

You can make up any name you want for your method, except that you can't call it `Main` or any keyword. By convention, methods start with an upper case letter and use "Pascal case," which is a cute name for `JammingWordsTogetherLikeThis`.

The list of parameters specifies what information, if any, you have to provide to use (or **invoke**) the new method.

The parameter for `Main` is `string[] args`, which means that whoever invokes `Main` has to provide an array of strings (we'll get to arrays in Chapter 10). The first couple of methods we are going to write have no parameters, so the syntax looks like this:

```
public static void NewLine() {
    Console.WriteLine("");
}
```

This method is named `NewLine`, and the empty parentheses mean that it takes no parameters. It contains one statement, which prints an empty `string`, indicated by `""`. Printing a `string` with no letters in it may not seem all that useful, but `WriteLine` skips to the next line after it prints, so this statement skips to the next line.

In `Main` we invoke this new method the same way we invoke methods:

```
public static void Main(string[] args) {
    Console.WriteLine("First line.");
    NewLine();
    Console.WriteLine("Second line.");
}
```

The output of this program is

```
First line.

Second line.
```

Notice the extra space between the lines. What if we wanted more space between the lines? We could invoke the same method repeatedly:

```csharp
public static void Main(string[] args) {
    Console.WriteLine("First line.");
    NewLine();
    NewLine();
    NewLine();
    Console.WriteLine("Second line.");
}
```

Or we could write a new method, named `ThreeLine`, that prints three new lines:

```csharp
public static void ThreeLine() {
    NewLine();  NewLine();  NewLine();
}

public static void Main(string[] args) {
    Console.WriteLine("First line.");
    ThreeLine();
    Console.WriteLine("Second line.");
}
```

You should notice a few things about this program:

- You can invoke the same procedure more than once.

- You can have one method invoke another method. In this case, `Main` invokes `ThreeLine` and `ThreeLine` invokes `NewLine`.

- In `ThreeLine` I wrote three statements all on the same line, which is syntactically legal (remember that spaces and new lines usually don't change the meaning of a program). It is usually a good idea to put each statement on its own line, but I sometimes break that rule.

You might wonder why it is worth the trouble to create all these new methods. There are several reasons; this example demonstrates two:

1. Creating a new method gives you an opportunity to give a name to a group of statements. Methods can simplify a program by hiding a complex computation behind a single statement, and by using English words in place of arcane code. Which is clearer, `NewLine` or `Console.WriteLine()`?

2. Creating a new method can make a program smaller by eliminating repetitive code. For example, to print nine consecutive new lines, you could invoke `ThreeLine` three times.

In Section 6.6 we will come back to this question and list some additional benefits of dividing programs into methods.

3.6 Classes and methods

Pulling together the code fragments from the previous section, the class definition looks like this:

```csharp
using System;

namespace ThinkSharp {
  public class Program {

    public static void NewLine() {
      Console.WriteLine("");
    }

    public static void ThreeLine() {
      NewLine();  NewLine();  NewLine();
    }

    public static void Main(string[] args) {
      Console.WriteLine("First line.");
      ThreeLine();
      Console.WriteLine("Second line.");
    }
  }
}
```

Line 4 indicates that this is the class definition for a new class called `Program`. A **class** is a collection of related methods. In this case, the class named `Program` contains three methods, named `NewLine`, `ThreeLine`, and `Main`.

The other class we've seen is the `Math` class. It contains methods named `Sqrt`, `Sin` and others. When we invoke a mathematical method, we have to specify the name of the class (`Math`) and the name of the method. That's why the syntax is slightly different for library methods and the methods we write:

```csharp
Math.Pow(2.0, 10.0);
NewLine();
```

The first statement invokes the `Pow` method in the Math class (which raises the first argument to the power of the second argument). The second statement invokes the `NewLine` method, which is in the class we are writing (i.e., `NewLine`).

If you try to invoke a method from the wrong class, the compiler will generate an error. For example, if you type:

```
Pow(2.0, 10.0);
```

The compiler will say something like, "The name 'Pow' does not exist in the current context." If you have seen this message, you might have wondered why it was looking for `Pow` in your class definition. Now you know.

3.7   Programs with multiple methods

When you look at a class definition that contains several methods, it is tempting to read it from top to bottom, but that is likely to be confusing, because that is not the **order of execution** of the program.

Execution always begins at the first statement of `Main`, regardless of where it is in the program (in this example I deliberately put it at the bottom). Statements are executed one at a time, in order, until you reach a method invocation. Method invocations are like a detour in the flow of execution. Instead of going to the next statement, you go to the first line of the invoked method, execute all the statements there, and then come back and pick up again where you left off.

That sounds simple enough, except that you have to remember that one method can invoke another. Thus, while we are in the middle of `Main`, we might have to go off and execute the statements in `ThreeLine`. But while we are executing `ThreeLine`, we get interrupted three times to go off and execute `NewLine`.

For its part, `NewLine` invokes `WriteLine`, which causes yet another detour. Fortunately, C# is adept at keeping track of where it is, so when `WriteLine` completes, it picks up where it left off in `NewLine`, and then gets back to `ThreeLine`, and then finally gets back to `Main` so the program can terminate.

Technically, the program does not terminate at the end of `Main`. Instead, execution picks up where it left off in the program that invoked `Main`, which is the C# interpreter. The interpreter takes care of things like deleting windows and general clean up, and *then* the program terminates.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution.

3.8 Parameters and arguments

Some of the methods we have used require **arguments**, which are values that you provide when you invoke the method. For example, to find the sine of a number, you have to provide the number. So `Sin` takes a `double` as an argument. To print a string, you have to provide the string, so `WriteLine` takes a `string` as an argument.

Some methods take more than one argument; for example, `Pow` takes two `double`s, the base and the exponent.

When you use a method, you provide arguments. When you write a method, you specify a list of parameters. A **parameter** is a variable that stores an argument. The parameter list indicates what arguments are required.

For example, `WriteTwice` specifies a single parameter, `str`, that has type `string`. I called it `str` to suggest that it is a `string`, but I could have given it any legal variable name.

```
public static void WriteTwice(string str) {
    Console.WriteLine(str);
    Console.WriteLine(str);
}
```

When we invoke `WriteTwice`, we have to provide a single argument with type `string`.

```
    WriteTwice("Don't make me say this twice!");
```

When you invoke a method, the argument you provide are assigned to the parameters. In this example, the argument `"Don't make me say this twice!"` is assigned to the parameter `str`. This processing is called **parameter passing** because the value gets passed from outside the method to the inside.

An argument can be any kind of expression, so if you have a `string` variable, you can use it as an argument:

```
    string argument = "Never say never.";
    WriteTwice(argument);
```

The value you provide as an argument must have the same type as the parameter. For example, if you try this:

```
    WriteTwice(17);
```

You get an error message like "The best overloaded method match for 'ThinkSharp. Program.WriteTwice(string)' has some invalid arguments," which isn't very helpful. The reason is that the compiler is looking for a method named `WriteTwice` that can take an integer argument. Since there isn't one, it looks for the next best thing. However, it will also say something like "Argument 1: cannot convert from 'int' to 'string'," which is a little more useful.

The error bubble has the number 2 at the end indicating two error messages. Hover the mouse over the bubble to see all of the errors.

`Console`.WriteLine can accept any type as an argument. But that is an exception; most methods are not so accommodating.

3.9   Stack diagrams

Parameters and other variables only exist inside their own methods. Within the confines of `Main`, there is no such thing as `str`. If you try to use it, the compiler will complain. Similarly, inside `WriteTwice` there is no such thing as `argument`.

One way to keep track of where each variable is defined is with a **stack diagram**. The stack diagram for the previous example looks like this:

Main | argument | "Never say never." |

WriteTwice | str | "Never say never." |
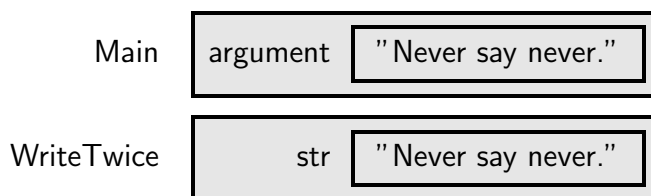
For each method there is a gray box called a **frame** that contains the method's parameters and variables. The name of the method appears outside the frame. As usual, the value of each variable is drawn inside a box with the name of the variable beside it.

3.10   Methods with multiple parameters

The syntax for declaring and invoking methods with multiple parameters is a common source of errors. First, remember that you have to declare the type of every parameter. For example

```
public static void WriteTime(int hour, int minute) {
   Console.Write(hour);
   Console.Write(":");
   Console.WriteLine(minute);
}
```

It might be tempting to write `int hour, minute`, but that format is only legal for variable declarations, not parameter lists.

Another common source of confusion is that you do not have to declare the types of arguments. The following is wrong!

```
   int hour = 11;
   int minute = 59;
   WriteTime(int hour, int minute);   // WRONG!
```

In this case, the system can tell the type of `hour` and `minute` by looking at their declarations. It is not necessary to include the type when you pass them as arguments. The correct syntax is `WriteTime(hour, minute)`.

3.11   Methods that return values

Some of the methods we are using, like the `Math` methods, return values. Other methods, like `WriteLine` and `NewLine`, perform an action but they don't return a value. That raises some questions:

- What happens if you invoke a method and you don't do anything with the result (i.e. you don't assign it to a variable or use it as part of a larger expression)?

- What happens if you use a `WriteLine` method as part of an expression, like `Console.WriteLine("boo!") + 7`?

- Can we write methods that return values, or are we stuck with things like `NewLine` and `WriteTwice`?

The answer to the third question is "yes, you can write methods that return values," and we'll see how in a couple of chapters. I leave it up to you to answer the other two questions by trying them out. In fact, any time you have a question about what is legal or illegal, a good way to find out is to ask the compiler.

3.12   Glossary

initialization: A statement that declares a new variable and assigns a value to it at the same time.

floating-point: A type of variable (or value) that can contain fractions as well as integers. The floating-point type we will use is `double`.

class: A named collection of methods. So far, we have used the `Math` class and the `Console` class, and we have written classes named `Hello` and `Program`.

method: A named sequence of statements that performs a useful function. Methods may or may not take parameters, and may or may not return a value.

parameter: A piece of information a method requires before it can run. Parameters are variables: they contain values and have types.

argument: A value that you provide when you invoke a method. This value must have the same type as the corresponding parameter.

frame: A structure (represented by a gray box in stack diagrams) that contains a method's parameters and variables.

invoke: Cause a method to execute.

3.13 Exercises

Exercise 3.1. Draw a stack frame that shows the state of the program in Section 3.10 when `Main` invokes `WriteTime` with the arguments `11` and `59`.

Exercise 3.2. The point of this exercise is to practice reading code and to make sure that you understand the flow of execution through a program with multiple methods.

1. What is the output of the following program? Be precise about where there are spaces and where there are newlines.

   HINT: Start by describing in words what `Ping` and `Baffle` do when they are invoked.

2. Draw a stack diagram that shows the state of the program the first time `Ping` is invoked.

```
5  public static void Zoop() {
6      Baffle();
7      Console.Write("You wugga ");
8      Baffle();
9  }
10
11 public static void Main(string[] args) {
12     Console.Write("No, I ");
13     Zoop();
14     Console.Write("I ");
15     Baffle();
16 }
17
18 public static void Baffle() {
19     Console.Write("wug");
20     Ping();
21 }
22
23 public static void Ping() {
24     Console.WriteLine(".");
25 }
```

Exercise 3.3. The point of this exercise is to make sure you understand how to write and invoke methods that take parameters.

1. Write the first line of a method named `Zool` that takes three parameters: an `int` and two `string`s.

2. Write a line of code that invokes `zool`, passing as arguments the value `11`, the name of your first pet, and the name of the street you grew up on.

Exercise 3.4. The purpose of this exercise is to take code from a previous exercise and encapsulate it in a method that takes parameters. You should start with a working solution to Exercise 2.2.

1. Write a method called `WriteEuropean` that takes the day, date, month and year as parameters and that prints them in European format.

2. Test your method by invoking it from `Main` and passing appropriate arguments. The output should look something like this (except that the date might be different):

    ```
    Saturday 16 July, 2011
    ```

3. Once you have debugged `WriteEuropean`, write another method called `WriteAmerican` that prints the date in American format (Saturday, July 16, 2011).

Chapter 4

Conditionals and recursion

4.1 The modulus operator

The modulus operator works on integers (and integer expressions) and yields the *remainder* when the first operand is divided by the second. The modulus operator is a percent sign, %. The syntax is the same as for other operators:

```
int quotient = 7 / 3;
int remainder = 7 % 3;
```

The first operator, integer division, yields 2. The second operator yields 1. Thus, 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if x % y is zero, then x is divisible by y.

Also, you can use the modulus operator to extract the rightmost digit or digits from a number. For example, x % 10 yields the rightmost digit of x (in base 10). Similarly x % 100 yields the last two digits.

4.2 Conditional execution

To write useful programs, we almost always need to check conditions and change the behaviour of the program accordingly. **Conditional statements** give us this ability. The simplest form is the if statement:

```
if ( x > 0 ) {
   Console.WriteLine("x is positive");
}
```

The expression in parentheses is called the condition. If it is true, then the statements in braces get executed. If the condition is not true, nothing happens.

The condition can contain any of the comparison operators, sometimes called **relational operators**:

```
x == y              // x equals y
x != y              // x is not equal to y
x > y               // x is greater than y
x < y               // x is less than y
x >= y              // x is greater than or equal to y
x <= y              // x is less than or equal to y
```

Although these operations are probably familiar to you, the syntax is a little different from mathematical symbols like $=$, $\neq$ and $\leq$. A common error is to use a single = instead of a double ==. Remember that = is the assignment operator, and == is a comparison operator. Also, there is no such thing as =< and => means something very different.

The two sides of a condition operator have to be the same type. You can only compare `int`s to `int`s and `double`s to `double`s.

The operators == and != also work with `string`s, but the other relational operators don't do anything at all. We will see how to compare strings Section 7.10.

4.3 Alternative execution

A second form of conditional execution is alternative execution, in which there are two possibilities, and the condition determines which one gets executed. The syntax looks like:

```
if ( x % 2 == 0 ) {
    Console.WriteLine("x is even");
} else {
    Console.WriteLine("x is odd");
}
```

If the remainder when `x` is divided by 2 is zero, then we know that `x` is even, and this code prints a message to that effect. If the condition is false, the second statement is executed. Since the condition must be true or false, exactly one of the alternatives will be executed.

As an aside, if you think you might want to check the parity (evenness or oddness) of numbers often, you might want to "wrap" this code up in a method, as follows:

```
public static void WriteParity(int x) {
    if ( x % 2 == 0 ) {
        Console.WriteLine("x is even");
    } else {
        Console.WriteLine("x is odd");
    }
}
```

Now you have a method named `WriteParity` that will write an appropriate message for any integer you care to provide. In `Main` you would invoke this method like this:

```
WriteParity(17);
```

Always remember that when you *invoke* a method, you do not have to declare the types of the arguments you provide. The compiler can figure out what type they are. You should resist the temptation to write things like:

```
int number = 17;
WriteParity(int number);          // WRONG!!!
```

4.4 Chained conditionals

Sometimes you want to check for a number of related conditions and choose one of several actions. One way to do this is by **chaining** a series of `if`s and `else`s:

```
if ( x > 0 ) {
   Console.WriteLine("x is positive");
} else if ( x < 0 ) {
   Console.WriteLine("x is negative");
} else {
   Console.WriteLine("x is zero");
}
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and squiggly-braces lined up, you are less likely to make syntax errors and more likely to find them if you do.

4.5 Nested conditionals

In addition to chaining, you can also nest one conditional within another. We could have written the previous example as:

```
if ( x == 0 ) {
   Console.WriteLine("x is zero");
} else {
   if ( x > 0 ) {
      Console.WriteLine("x is positive");
   } else {
      Console.WriteLine("x is negative");
   }
}
```

There is now an outer conditional that contains two branches. The first branch contains a simple `WriteLine` statement, but the second branch contains another conditional statement, which has two branches of its own. Those two branches are both `WriteLine` statements, but they could have been conditional statements as well.

Indentation helps make the structure apparent, but nevertheless, nested conditionals get difficult to read very quickly. Avoid them when you can.

On the other hand, this kind of **nested structure** is common, and we will see it again, so you better get used to it. I find it useful to add a comment on each branch describing the condition, try adding the following comments to this code:

```
// x is zero
// x is positive
// x is negative
```

where would you add these comments to this code?

4.6   The return statement

The `return` statement allows you to terminate the execution of a method before you reach the end. One reason to use it is if you detect an error condition:

```csharp
public static void WriteLogarithm(double x) {
    if ( x <= 0.0 ) {
        Console.WriteLine("Positive numbers only, please.");
        return;
    }

    double result = Math.Log(x);
    Console.WriteLine("The log of x is " + result);
}
}
```

This defines a method named `WriteLogarithm` that takes a `double` named `x` as a parameter. It checks whether `x` is less than or equal to zero, in which case it prints an error message and then uses `return` to exit the method. The flow of execution immediately returns to the caller and the remaining lines of the method are not executed.

I used a floating-point value on the right side of the condition because there is a floating-point variable on the left.

4.7   Type conversion

You might wonder how you can get away with an expression like `"The log of x is "`
`+ result`, since one of the operands is a `string` and the other is a `double`. In this case
the system is being smart on our behalf, automatically converting the `double` to a `string`
before it does the string concatenation.

Whenever you try to "add" two expressions, if one of them is a `string`, the system converts
the other to a `string` and then perform string concatenation. What do you think happens
if you perform an operation between an integer and a floating-point value?

4.8   Recursion

I mentioned in the last chapter that it is legal for one method to invoke another, and we
have seen several examples. I neglected to mention that it is also legal for a method to
invoke itself. It may not be obvious why that is a good thing, but it turns out to be one
of the most magical and interesting things a program can do.

For example, look at the following method:

```
public static void Countdown(int n) {
   if ( n == 0 ) {
      Console.WriteLine("Blastoff!");
   } else {
      Console.WriteLine(n);
      Countdown(n-1);
   }
}
```

The name of the method is `Countdown` and it takes a single integer as a parameter. If the
parameter is zero, it prints the word "Blastoff." Otherwise, it prints the number and then
invokes a method named `Countdown`—itself—passing `n-1` as an argument.

What happens if we invoke this method, in `Main`, like this:

```
countdown(3);
```

The execution of `Countdown` begins with `n=3`, and since `n` is not zero, it prints the value 3,
and then invokes itself...

> The execution of `Countdown` begins with `n=2`, and since `n` is not zero, it prints
> the value 2, and then invokes itself...
>
> > The execution of `Countdown` begins with `n=1`, and since `n` is not zero,
> > it prints the value 1, and then invokes itself...
> >
> > > The execution of `Countdown` begins with `n=0`, and since `n` is
> > > zero, it prints the word "Blastoff!" and then returns.

The `Countdown` that got `n=1` returns.

The `Countdown` that got `n=2` returns.

The `Countdown` that got `n=3` returns.

And then you're back in `Main`. So the total output looks like:

```
3
2
1
Blastoff!
```

As a second example, let's look again at the methods `NewLine` and `ThreeLine`.

```
public static void NewLine() {
    Console.WriteLine("");
}

public static void ThreeLine() {
    NewLine();  NewLine();  NewLine();
}
```

Although these work, they would not be much help if we wanted to print 2 newlines, or 106. A better alternative would be

```
public static void NLines(int n) {
    if ( n > 0 ) {
        Console.WriteLine();
        NLines(n-1);
    }
}
```

This program similar to `Countdown`; as long as `n` is greater than zero, it prints a newline and then invokes itself to print `n-1` additional newlines. The total number of newlines that get printed is $1 + (n - 1)$, which usually comes out to roughly `n`.

When a method invokes itself, that's called **recursion**, and such methods are **recursive**.

4.9 Stack diagrams for recursive methods

In the previous chapter we used a stack diagram to represent the state of a program during a method invocation. The same kind of diagram can make it easier to interpret a recursive method.

Remember that every time a method gets called it creates a new frame that contains a new version of the method's parameters and variables.

The following figure is a stack diagram for countdown, called with `n = 3`:

```
Main       ┌──────────┐
           │          │
           └──────────┘
Countdown  ┌──────────┐
           │  n │ 3 │ │
           └──────────┘
Countdown  ┌──────────┐
           │  n │ 2 │ │
           └──────────┘
Countdown  ┌──────────┐
           │  n │ 1 │ │
           └──────────┘
Countdown  ┌──────────┐
           │  n │ 0 │ │
           └──────────┘
```
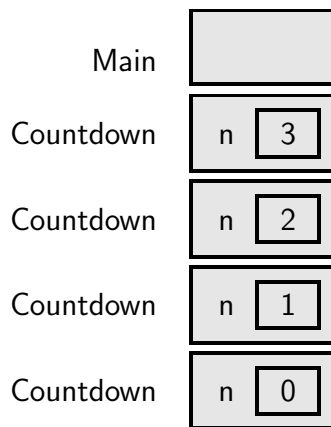
There is one frame for `Main` and four frames for `Countdown`, each with a different value for the parameter `n`. The bottom of the stack, `Countdown` with `n=0` is called the **base case**. It does not make a recursive call, so there are no more frames for `Countdown`.

The frame for `Main` is empty because `Main` does not have any parameters or variables.

4.10 Glossary

modulus: An operator that works on integers and yields the remainder when one number is divided by another. This is denoted with a percent sign (`%`).

conditional: A block of statements that may or may not be executed depending on some condition.

chaining: A way of joining several conditional statements in sequence.

nesting: Putting a conditional statement inside one or both branches of another conditional statement.

typecast: An operator that converts from one type to another. It appears as a type name in parentheses, like (`int`).

recursion: The process of invoking the same method you are currently executing.

base case: A condition that causes a recursive method *not* to make a recursive call.

4.11 Exercises

Exercise 4.1. Draw a stack diagram that shows the state of the program in Section 4.8 after `Main` invokes `NLines` with the parameter `n = 4`, just before the last invocation of `NLines` returns.

Exercise 4.2. This exercise reviews the flow of execution through a program with multiple methods. Read the following code and answer the questions below.

```csharp
public class Buzz {

    public static void Baffle(string blimp) {
        Console.WriteLine(blimp);
        Zippo("ping", -5);
    }

    public static void Zippo(string quince, int flag) {
        if ( flag < 0 ) {
            Console.WriteLine(quince + " zoop");
        } else {
            Console.WriteLine("ik");
            Baffle(quince);
            Console.WriteLine("boo-wa-ha-ha");
        }
    }

    public static void Main(string[] args) {
        Zippo("rattle", 13);
    }
}
```

1. Write the number 1 next to the first *statement* of this program that will be executed. Be careful to distinguish things that are statements from things that are not.

2. Write the number 2 next to the second statement, and so on until the end of the program. If a statement is executed more than once, it might end up with more than one number next to it.

3. What is the value of the parameter `blimp` when `Baffle` gets invoked?

4. What is the output of this program?

Exercise 4.3. The first verse of the song "99 Bottles of Beer" is:

> 99 bottles of beer on the wall, 99 bottles of beer, ya' take one down, ya' pass it around, 98 bottles of beer on the wall.

Subsequent verses are identical except that the number of bottles gets smaller by one in each verse, until the last verse:

> No bottles of beer on the wall, no bottles of beer, ya' can't take one down, ya' can't pass it around, 'cause there are no more bottles of beer on the wall!

And then the song (finally) ends.

Write a program that prints the entire lyrics of "99 Bottles of Beer." Your program should include a recursive method that does the hard part, but you might want to write additional methods to separate the major functions of the program.

As you develop your code, test it with a small number of verses, like "3 Bottles of Beer."

The purpose of this exercise is to take a problem and break it into smaller problems, and to solve the smaller problems by writing simple methods.

Exercise 4.4. What is the output of the following program?

```csharp
public class Narf {

    public static void Zoop(string fred, int bob) {
        Console.WriteLine(fred);
        if ( bob == 5 ) {
            Ping("not ");
        } else {
            Console.WriteLine("!");
        }
    }

    public static void Main(string[] args) {
        int bizz = 5;
        int buzz = 2;
        Zoop("just for", bizz);
        Clink(2*buzz);
    }

    public static void Clink(int fork) {
        Console.WriteLine("It's ");
        Zoop("breakfast ", fork) ;
    }

    public static void Ping(string strangStrung) {
        Console.WriteLine("any " + strangStrung + "more ");
    }
}
```

Exercise 4.5. Fermat's Last Theorem says that there are no integers a, b, and c such that

$$a^n + b^n = c^n$$

except in the case when $n = 2$.

Write a method named `CheckFermat` that takes four integers as parameters—`a`, `b`, `c` and `n`—and that checks to see if Fermat's theorem holds. If n is greater than 2 and it turns out to be true that $a^n + b^n = c^n$, the program should print "Holy smokes, Fermat was wrong!" Otherwise the program should print "No, that doesn't work."

You should use the `Math.Pow` library function that takes two integers as arguments and raises the first argument to the power of the second. For example:

```
int x = Math.Pow(2, 3);
```

would assign the value 8 to x, because $2^3 = 8$.

Chapter 5

Value methods

5.1 Return values

Some of the methods we have used, like the `Math` functions, produce results. That is, the effect of invoking the method is to generate a new value, which we usually assign to a variable or use as part of an expression. For example:

```
double e = Math.Exp(1.0);
double height = radius * Math.Sin(angle);
```

But so far all our methods have been **void**; that is, methods that return no value. When you invoke a void method, it is typically on a line by itself, with no assignment:

```
Countdown(3);
NLines(3);
```

In this chapter we write methods that return things, which I call **value** methods. The first example is `Area`, which takes a `double` as a parameter, and returns the area of a circle with the given radius:

```
public static double Area(double radius) {
    double area = Math.PI * radius * radius;
    return area;
}
```

The first thing you should notice is that the beginning of the method definition is different. Instead of `public static void`, which indicates a void method, we see `public static double`, which means that the return value from this method is a `double`. I still haven't explained what `public static` means, but be patient.

The last line is a new form of the `return` statement that includes a return value. This statement means, "return immediately from this method and use the following expression as the return value." The expression you provide can be arbitrarily complicated, so we could have written this method more concisely:

```
    public static double Area(double radius) {
        return Math.PI * radius * radius;
    }
```

On the other hand, **temporary** variables like `area` often make debugging easier. In either case, the type of the expression in the `return` statement must match the return type of the method. In other words, when you declare that the return type is `double`, you are making a promise that this method will eventually produce a `double`. If you try to `return` with no expression, or an expression with the wrong type, the compiler will take you to task.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
    public static double AbsoluteValue(double x) {
        if ( x < 0 ) {
            return -x;
        } else {
            return x;
        }
    }
```

Since these return statements are in an alternative conditional, only one will be executed. Although it is legal to have more than one return statement in a method, you should keep in mind that as soon as one is executed, the method terminates without executing any subsequent statements.

Code that appears after a `return` statement, or any place else where it can never be executed, is called **dead code**. Some compilers warn you if part of your code is dead.

If you put return statements inside a conditional, then you have to guarantee that *every possible path* through the program hits a return statement. For example:

```
    public static double AbsoluteValue(double x) {
        if ( x < 0 ) {
            return -x;
        } else if ( x > 0 ) {
            return x;
        }                               // WRONG!!
    }
```

This program is not legal because if `x` is 0, neither condition is true and the method ends without hitting a return statement. A typical compiler message would be "Absolute-Value(double): not all code paths return a value," which is a confusing message since there are already two of them.

5.2 Program development

At this point you should be able to look at complete methods and tell what they do. But it may not be clear yet how to go about writing them. I am going to suggest a method called **incremental development**.

As an example, imagine you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the usual definition,

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a `Distance` method should look like. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the two points are the parameters, and it is natural to represent them using four `double`s, although we will see later that there is a `Point` object that we could use. The return value is the distance, which will have type `double`.

Already we can write an outline of the method:

```
public static double Distance
                        (double x1, double y1, double x2, double y2) {
    return 0.0;
}
```

The statement `return 0.0;` is a place-keeper that is necessary to compile the program. Obviously, at this stage the program doesn't do anything useful, but it is worthwhile to try compiling it so we can identify any syntax errors before we add more code.

To test the new method, we have to invoke it with sample values. Somewhere in `Main` I would add:

```
double dist = Distance(1.0, 2.0, 4.0, 6.0);
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result will be 5 (the hypotenuse of a 3-4-5 triangle). When you are testing a method, it is useful to know the right answer.

Once we have checked the syntax of the method definition, we can start adding lines of code one at a time. After each incremental change, we recompile and run the program. If there is an error at any point, we have a good idea where to look: in the last line we added.

The next step is to find the differences $x_2 - x_1$ and $y_2 - y_1$. I store those values in temporary variables named `dx` and `dy`[1].

---

[1]A mathematician would want to name these values Δx and Δy, where Δ stands for difference. While I can use Δ in the names for these variables, I find it much easer to write `d`.

```
public static double Distance
                        (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    Console.WriteLine("dx is " + dx);
    Console.WriteLine("dy is " + dy);
    return 0.0;
}
```

I added write statements so we can check the intermediate values before proceeding. They should be 3.0 and 4.0.

When the method is finished I remove the write statements. Code like that is called **scaffolding**, because it is helpful for building the program, but it is not part of the final product.

The next step is to square `dx` and `dy`. We could use the `Math.Pow` method, but it is simpler to multiply each term by itself.

```
public static double Distance
                        (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx*dx + dy*dy;
    Console.WriteLine("dsquared is " + dsquared);
    return 0.0;
}
```

Again, I would compile and run the program at this stage and check the intermediate value (which should be 25.0).

Finally, we can use `Math.Sqrt` to compute and return the result.

```
public static double Distance
                        (double x1, double y1, double x2, double y2) {
double dx = x2 - x1;
double dy = y2 - y1;
double dsquared = dx*dx + dy*dy;
double result = Math.sqrt(dsquared);
return result;
}
```

In `Main`, we can write and check the value of the result.

As you gain more experience programming, you might write and debug more than one line at a time. Nevertheless, incremental development can save you a lot of time. The key aspects of the process are:

- Start with a working program and make small, incremental changes. At any point, if there is an error, you will know exactly where it is.

- Use temporary variables to hold intermediate values so you can write and check them.

- Once the program is working, you can remove scaffolding and consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

5.3 Composition

Once you define a new method, you can use it as part of an expression, and you can build new methods using existing methods. For example, what if someone gave you two points, the centre of the circle and a point on the perimeter, and asked for the area of the circle?

Let's say the centre point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we have a method, `Distance` that does that.

```
double radius = Distance(xc, yc, xp, yp);
```

The second step is to find the area of a circle with that radius, and return it.

```
double area = Area(radius);
return area;
```

Wrapping that all up in a method, we get:

```
public static double CircleArea
                    (double xc, double yc, double xp, double yp) {
    double radius = Distance(xc, yc, xp, yp);
    double area = Area(radius);
    return area;
}
```

The temporary variables `radius` and `area` are useful for development and debugging, but once the program is working we can make it more concise by composing the method invocations:

```
public static double CircleArea
                    (double xc, double yc, double xp, double yp) {
    return Area(Distance(xc, yc, xp, yp));
}
```

5.4 Overloading

You might have noticed that `CircleArea` and `Area` perform similar functions—finding the area of a circle—but take different parameters. For `Area`, we have to provide the radius;

for `CircleArea` we provide two points.

If two methods do the same thing, it is natural to give them the same name. Having more than one method with the same name, which is called **overloading**, is legal *as long as each version takes different parameters.* So we could rename `CircleArea`:

```
public static double Area (double x1, double y1, double x2, double y2) {
    return Area(Distance(xc, yc, xp, yp));
}
```

When you invoke an overloaded method, the system knows which version you want by looking at the arguments that you provide. If you write:

```
double x = Area(3.0);
```

it goes looking for a method named `Area` that takes one `double` as an argument, and so it uses the first version, which interprets the argument as a radius. If you write:

```
double x = Area(1.0, 2.0, 4.0, 6.0);
```

It uses the second version of `Area`. And notice that the second version of `Area` actually invokes the first.

Many methods are overloaded, meaning that there are different versions that accept different numbers or types of parameters. For example, there are versions of `Write` and `WriteLine` that accept a single parameter of any type. In the `Math` class, there is a version of `Abs` that works on `double`s, and there is also a version for `int`s.

Although overloading is a useful feature, it should be used with caution. You might get yourself nicely confused if you are trying to debug one version of a method while accidentally invoking a different one.

And that reminds me of one of the cardinal rules of debugging: **make sure that the version of the program you are looking at is the version of the program that is running!**

Some day you may find yourself making one change after another in your program, and seeing the same thing every time you run it. This is a warning sign that you are not running the version of the program you think you are. To check, add a `Write` statement (it doesn't matter what you write) and make sure the behaviour of the program changes accordingly.

5.5   Boolean expressions

Most of the operations we have seen produce results that are the same type as their operands. For example, the `+` operator takes two `int`s and produces an `int`, or two `double`s and produces a `double`, etc.

The exceptions we have seen are the **relational operators**, which compare `int`s and `double`s and return either `true` or `false`. `true` and `false` are special values, and together they make up a type called **boolean**. You might recall that when I defined a type, I said it was a set of values. In the case of `int`s, `double`s and `string`s, those sets are pretty big. For booleans (`bool`s), there are only two values.

Boolean expressions and variables work just like other types of expressions and variables:

```
bool flag;
flag = true;
bool testResult = false;
```

The first example is a simple variable declaration; the second example is an assignment, and the third example is an initialization.

The values `true` and `false` are keywords, so they may appear in a different colour.

The result of a conditional operator is a boolean, so you can store the result of a comparison in a variable:

```
bool evenFlag = (n % 2 == 0);   // true if n is even
bool positiveFlag = (x > 0);    // true if x is positive
```

and then use it as part of a conditional statement later:

```
if ( evenFlag ) {
    Console.WriteLine("n was even when I checked it");
}
```

A variable used in this way is called a **flag** because it flags the presence or absence of some condition.

5.6   Logical operators

There are three **logical operators**: AND, OR and NOT, which are denoted by the symbols `&&`, `||` and `!`. The semantics of these operators is similar to their meaning in English. For example `x > 0 && x < 10` is true only if `x` is greater than zero AND less than 10.

`evenFlag || n%3 == 0` is true if *either* of the conditions is true, that is, if `evenFlag` is true OR the number is divisible by 3.

Finally, the NOT operator inverts a boolean expression, so `! evenFlag` is true if `evenFlag` is false—if the number is odd.

Logical operators can simplify nested conditional statements. For example, can you rewrite this code using a single conditional?

```
if ( x > 0 ) {
    if ( x < 10 ) {
        Console.WriteLine("x is a positive single digit.");
    }
}
```

5.7 Boolean methods

Methods can return boolean values just like any other type, which is often convenient for hiding tests inside methods. For example:

```
public static bool IsSingleDigit(int x) {
    if ( x >= 0 && x < 10 ) {
        return true;
    } else {
        return false;
    }
}
```

The name of this method is `IsSingleDigit`. It is common to give boolean methods names that sound like yes/no questions. The return type is `bool`, which means that every return statement has to provide a boolean expression.

The code itself is straightforward, although it is longer than it needs to be. Remember that the expression `x >= 0 && x < 10` has type boolean, so there is nothing wrong with returning it directly and avoiding the `if` statement altogether:

```
public static bool IsSingleDigit(int x) {
    return (x >= 0 && x < 10);
}
```

In `Main` you can invoke this method in the usual ways:

```
bool bigFlag = !IsSingleDigit(17);
Console.WriteLine(IsSingleDigit(2));
```

The first line sets `bigFlag` to `true` only if 17 is *not* a single-digit number. The second line prints `true` because 2 is a single-digit number.

The most common use of boolean methods is inside conditional statements

```
if ( IsSingleDigit(x) ) {
    Console.WriteLine("x is little");
} else {
    Console.WriteLine("x is big");
}
```

5.8 More recursion

Now that we have methods that return values, we have a **Turing complete** programming language, which means that we can compute anything computable, for any reasonable definition of "computable." This idea was developed by Alonzo Church and Alan Turing, so it is known as the Church-Turing thesis[2].

---

[2]You can read more about it at http://en.wikipedia.org/wiki/Turing_thesis

To give you an idea of what you can do with the tools we have learned, let's look at some methods for evaluating recursively-defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

recursive: an adjective used to describe a method that is recursive.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the mathematical function **factorial**, you might get something like:

$$0! = 1$$
$$n! = n \cdot (n - 1)!$$

(Factorial is usually denoted with the symbol !, which is not to be confused with the logical operator ! which means NOT.) This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n - 1$. So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, we get 3! equal to 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a method to evaluate it. The first step is to decide what the parameters are and what the return type is. Since factorial is defined for integers, the method takes an integer as a parameter and returns an integer:

```
public static int Factorial(int n) {
}
```

If the argument happens to be zero, return 1:

```
public static int Factorial(int n) {
   if ( n == 0 ) {
      return 1;
   }
}
```

That's the base case.

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n - 1$, and then multiply it by n.

```
public static int Factorial(int n) {
   if ( n == 0 ) {
      return 1;
   } else {
      int recurse = Factorial(n - 1);
      int result = n * recurse;
      return result;
   }
}
```

The flow of execution for this program is similar to `Countdown` from Section 4.8. If we invoke `Factorial` with the value 3:

Since 3 is not zero, we take the second branch and calculate the factorial of $n - 1$...

> Since 2 is not zero, we take the second branch and calculate the factorial of $n - 1$...
>
> > Since 1 is not zero, we take the second branch and calculate the factorial of $n - 1$...
> >
> > > Since 0 *is* zero, we take the first branch and return the value 1 immediately without making any more recursive invocations.
> >
> > The return value (1) gets multiplied by `n`, which is 1, and the result is returned.
>
> The return value (1) gets multiplied by `n`, which is 2, and the result is returned.

The return value (2) gets multiplied by `n`, which is 3, and the result, 6, is returned to `Main`, or whoever invoked `Factorial(3)`.

Here is what the stack diagram looks like for this sequence of method invocations:
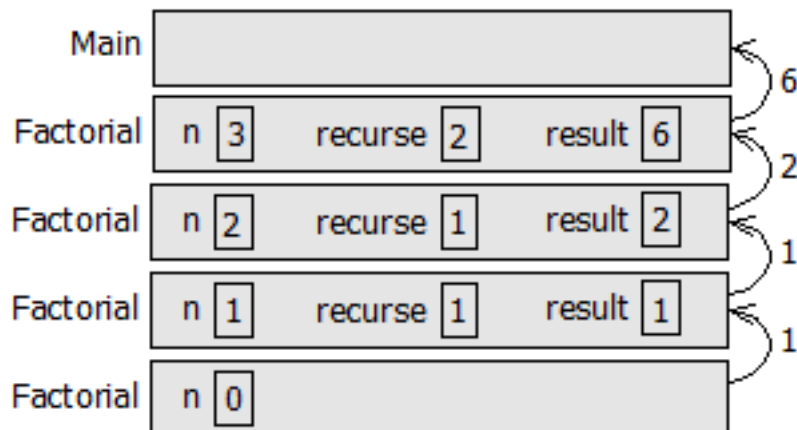
The return values are shown being passed back up the stack.

Notice that in the last frame `recurse` and `result` do not exist because when `n=0` the branch that creates them does not execute.

5.9   Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become disorienting. An alternative is what I call the "leap of faith." When you come to a method invocation, instead of following the flow of execution, you *assume* that the method works correctly and returns the appropriate value.

In fact, you are already practising this leap of faith when you use library methods. When you invoke `Math`.`Cos` or `Console`.`WriteLine`, you don't examine the implementations of those methods. You just assume that they work.

You can apply the same logic to your own methods. For example, in Section 5.7 we wrote a method called `IsSingleDigit` that determines whether a number is between 0 and 9. Once we convince ourselves that this method is correct—by testing and examination of the code—we can use the method without ever looking at the code again.

The same is true of recursive programs. When you get to the recursive invocation, instead of following the flow of execution, you should *assume* that the recursive invocation works, and then ask yourself, "Assuming that I can find the factorial of $n - 1$, can I compute the factorial of n?" Yes, you can, by multiplying by n.

Of course, it is strange to assume that the method works correctly when you have not even finished writing it, but that's why it's called a leap of faith!

5.10   One more example

The second most common example of a recursively-defined mathematical function is `Fibonacci`, which has the following definition:

$$fibonacci(0) = 1$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2);$$

Translated into program code, this is

```
public static int Fibonacci(int n) {
   if ( n == 0 || n == 1 ) {
      return 1;
   } else {
      return Fibonacci(n - 1) + Fibonacci(n - 2);
   }
}
```

If you try to follow the flow of execution here, even for small values of `n`, your head explodes. But according to the leap of faith, if we assume that the two recursive invocations work correctly, then it is clear that we get the right result by adding them together.

5.11   Glossary

return type: The part of a method declaration that indicates what type of value the method returns.

return value: The value provided as the result of a method invocation.

dead code: Part of a program that can never be executed, often because it appears after a `return` statement.

scaffolding: Code that is used during program development but is not part of the final version.

void: A special return type that indicates a void method; that is, one that does not return a value.

overloading: Having more than one method with the same name but different parameters. When you invoke an overloaded method, the system knows which version to use by looking at the arguments you provide.

boolean: A type of variable that can contain only the two values `true` and `false`.

flag: A variable (usually `bool`) that records a condition or status information.

conditional operator: An operator that compares two values and produces a boolean that indicates the relationship between the operands.

logical operator: An operator that combines boolean values and produces boolean values.

5.12   Exercises

Exercise 5.1. Write a method named `IsDivisible` that takes two integers, `n` and `m` and that returns `true` if `n` is divisible by `m` and `false` otherwise.

Exercise 5.2. Many computations can be expressed concisely using the "multadd" operation, which takes three operands and computes `a * b + c`. Some processors even provide a hardware implementation of this operation for floating-point numbers.

1. Create a new program called `Multadd.cs`.

2. Write a method called `Multadd` that takes three `double`s as parameters and that returns their multadditionization.

3. Write a `Main` method that tests `Multadd` by invoking it with a few simple parameters, like `1.0, 2.0, 3.0`.

4. Also in `Main`, use `multadd` to compute the following values:

$$\sin \frac{\pi}{4} + \frac{\cos \frac{\pi}{4}}{2}$$
$$\log 10 + \log 20$$

5. Write a method called `Yikes` that takes a double as a parameter and that uses `multadd` to calculate

$$xe^{-x} + \sqrt{1 - e^{-x}}$$

HINT: the `Math.Exp` method can be used for raising e to a power.

In the last part, you get a chance to write a method that invokes a method you wrote. Whenever you do that, it is a good idea to test the first method carefully before you start working on the second. Otherwise, you might find yourself debugging two methods at the same time, which can be difficult.

One of the purposes of this exercise is to practice pattern-matching: the ability to recognize a specific problem as an instance of a general category of problems.

Exercise 5.3. If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

> "If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can."

Write a method named `IsTriangle` that takes three integers as arguments, and returns either `true` or `false`, depending on whether you can or cannot form a triangle from sticks with the given lengths.

The point of this exercise is to use conditional statements to write a value method.

Exercise 5.4. What is the output of the following program? The purpose of this exercise is to make sure you understand logical operators and the flow of execution through value methods.

```csharp
public static void Main(string[] args) {
    bool flag1 = IsHoopy(202);
    bool flag2 = IsFrabjuous(202);
    Console.WriteLine(flag1);
    Console.WriteLine(flag2);
    if ( flag1 && flag2 ) {
        Console.WriteLine("ping!");
    }
    if ( flag1 || flag2 ) {
        Console.WriteLine("pong!");
    }
}
```

```csharp
public static bool IsHoopy(int x) {
   bool hoopyFlag;
   if ( x % 2 == 0 ) {
      hoopyFlag = true;
   } else {
      hoopyFlag = false;
   }
   return hoopyFlag;
}


public static bool IsFrabjuous(int x) {
   bool frabjuousFlag;
   if ( x > 0 ) {
      frabjuousFlag = true;
   } else {
      frabjuousFlag = false;
   }
   return frabjuousFlag;
}
```

Exercise 5.5. The point of this exercise is to use a stack diagram to understand the execution of a recursive program.

```csharp
public class Prod {

   public static void Main(string[] args) {
      Console.WriteLine(Prod(1, 4));
   }

   public static int Prod(int m, int n) {
      if ( m == n ) {
         return n;
      } else {
         int recurse = Prod(m, n-1);
         int result = n * recurse;
         return result;
      }
   }
}
```

1. Draw a stack diagram showing the state of the program just before the last instance of `Prod` completes. What is the output of this program?

2. Explain in a few words what `Prod` does.

3. Rewrite `Prod` without using the temporary variables `recurse` and `result`.

Exercise 5.6. The purpose of this exercise is to translate a recursive definition into a method. The Ackerman function is defined for non-negative integers as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Write a method called `Ack` that takes two `int`s as parameters and that computes and returns the value of the Ackerman function.

Test your implementation of Ackerman by invoking it from `Main` and printing the return value.

WARNING: the return value gets very big very quickly. You should try it only for small values of m and n (not bigger than 2).

Exercise 5.7.

1. Create a program called `Recurse.cs` and type in the following methods:

   ```
   // first: returns the first character of the given string
   public static char First(string s) {
      return s[0];
   }

   // rest: returns a new string that contains all but the
   // first letter of the given string
   public static string Rest(string s) {
      return s.substring(1);
   }

   // length: returns the length of the given string
   public static int Length(string s) {
      return s.Length;
   }
   ```

2. Write some code in `Main` that tests each of these methods. Make sure they work, and make sure you understand what they do.

3. Write a method called `WriteString` that takes a `string` as a parameter and that prints the letters of the string, one on each line. It should be a `void` method.

4. Write a method called `WriteBackward` that does the same thing as `WriteString` but that prints the string backward (one character per line).

5. Write a method called `ReverseString` that takes a `string` as a parameter and that returns a new `string` as a return value. The new string should contain the same letters as the parameter, but in reverse order. For example, the output of the following code

```
    string backwards = ReverseString("Allen Downey");
    Console.WriteLine(backwards);
```
should be

```
yenwoD nellA
```

Exercise 5.8. Write a recursive method called `Power` that takes a double `x` and an integer `n` and that returns x^n.

HINT: a recursive definition of this operation is $x^n = x \cdot x^{n-1}$. Also, remember that anything raised to the zeroeth power is 1.

Optional challenge: you can make this method more efficient, when `n` is even, using $x^n = \left(x^{n/2}\right)^2$.

Exercise 5.9. (This exercise is based on page 44 of Ableson and Sussman's *Structure and Interpretation of Computer Programs.*)

The following technique is known as Euclid's Algorithm because it appears in Euclid's *Elements* (Book 7, ca. 300 BC). It may be the oldest non-trivial algorithm[3]. It is used to calculate the greatest common divisor of two numbers.

The process is based on the observation that, if r is the remainder when a is divided by b, then the common divisors of a and b are the same as the common divisors of b and r. Thus we can use the equation

$$gcd(a, b) = gcd(b, r)$$

to successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example,

$$gcd(36, 20) = gcd(20, 16) = gcd(16, 4) = gcd(4, 0) = 4$$

implies that the GCD of 36 and 20 is 4. It can be shown that for any two starting numbers, this repeated reduction eventually produces a pair where the second number is 0. Then the GCD is the other number in the pair.

Write a method called `GCD` that takes two integer parameters and that uses Euclid's algorithm to compute and return the greatest common divisor of the two numbers.

---

[3]For a definition of "algorithm," jump ahead to Section 9.13.

Chapter 6

Iteration and loops

6.1 Multiple assignment

You can make more than one assignment to the same variable; the effect is to replace the old value with the new.

```
int liz = 5;
Console.Write(liz);
liz = 7;
Console.Write(liz);
```

The output of this program is 57, because the first time we print `liz` her value is 5, and the second time her value is 7.

This kind of **multiple assignment** is the reason I described variables as a *container* for values. When you assign a value to a variable, you change the contents of the container, as shown in the figure:

| `int liz = 5` | liz | 5 |
| `liz = 7` | liz | X̶ 7 |

When there are multiple assignments to a variable, it is especially important to distinguish between an assignment statement and a statement of equality. Because we use the = symbol for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

First of all, equality is commutative, and assignment is not. For example, in mathematics if $a = 7$ then $7 = a$. But in programming `a = 7;` is a legal assignment statement, and `7 = a;` is not.

Furthermore, in mathematics, a statement of equality is true for all time. If $a = b$ now, then a will always equal b. In programming, an assignment statement can make two variables equal, but they don't have to stay that way!

```
int a = 5;
int b = a;       // a and b are now equal
a = 3;           // a and b are no longer equal
```

The third line changes the value of `a` but it does not change the value of `b`, so they are no longer equal.

Although multiple assignment is frequently useful, you should use it with caution. If the values of variables change often, it can make the code difficult to read and debug.

6.2   The `while` statement

Computers are often used to automate repetitive tasks. Repeating tasks without making errors is something that computers do well and people do poorly.

We have already seen methods like `Countdown` and `Factorial` that use recursion to perform repetition. This process is also called **iteration**. There are language features that make it easier to write these methods. In this chapter we are going to look at the `while` statement. Later on we will check out the `for` (in Section 10.4) and `foreach` (in Section 10.5) statements.

Using a `while` statement, we can rewrite `Countdown`:

```
public static void Countdown(int n) {
    while ( n > 0 ) {
        Console.WriteLine(n);
        n = n-1;
    }
    Console.WriteLine("Blastoff!");
}
```

You can almost read a `while` statement like English. What this means is, "While `n` is greater than zero, print the value of `n` and then reduce the value of `n` by 1. When you get to zero, print the word 'Blastoff!'"

More formally, the flow of execution for a `while` statement is as follows:

1. Evaluate the condition in parentheses, yielding `true` or `false`.

2. If the condition is false, exit the `while` statement and continue execution at the next statement.

3. If the condition is true, execute the statements between the squiggly-braces, and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. The statements inside the loop are called the **body** of the loop. If the condition is false the first time through the loop, the statements inside the loop are never executed.

The body of the loop should change the value of one or more variables so that, eventually, the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite** loop. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of `Countdown`, we can prove that the loop terminates if `n` is positive. In other cases it is not so easy to tell:

```
public static void Sequence(int n) {
    while ( n != 1 ) {
        Console.WriteLine(n);
        if ( n % 2 == 0 ) {      // n is even
            n = n / 2;
        } else {                 // n is odd
            n = n*3 + 1;
        }
    }
}
```

The condition for this loop is `n != 1`, so the loop will continue until `n` is 1, which will make the condition false.

At each iteration, the program prints the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by two. If it is odd, the value is replaced by $3n + 1$. For example, if the starting value (the argument passed to `sequence`) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program will terminate. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of `n` will be even every time through the loop, until we get to 1. The previous example ends with such a sequence, starting with 16.

Particular values aside, the interesting question is whether we can prove that this program terminates for *all* values of n. So far, no one has been able to prove it *or* disprove it! For more information, see the Collatz Conjecture[1].

---

[1]http://en.wikipedia.org/wiki/Collatz_conjecture

6.3 Tables

One of the things loops are good for is generating and printing tabular data. For example, before computers were readily available, people had to calculate logarithms, sines and cosines, and other common mathematical functions by hand.

To make that easier, there were books containing long tables where you could find the values of various functions. Creating these tables was slow and boring, and the results were full of errors.

When computers appeared on the scene, one of the initial reactions was, "This is great! We can use the computers to generate the tables, so there will be no errors." That turned out to be true (mostly), but short sighted. Soon thereafter computers were so pervasive that the tables became obsolete.

Well, almost. For some operations, computers use tables of values to get an approximate answer, and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the original Intel Pentium used to perform floating-point division[2].

Although a "log table" is not as useful as it once was, it still makes a good example of iteration. The following program prints a sequence of values in the left column and their logarithms in the right column:

```
double x = 1.0;
while ( x < 10.0 ) {
    Console.WriteLine(x + "   " + Math.Log(x));
    x = x + 1.0;
}
```

The output of this program is

```
1   0
2   0.693147180559945
3   1.09861228866811
4   1.38629436111989
5   1.6094379124341
6   1.79175946922805
7   1.94591014905531
8   2.07944154167984
9   2.19722457733622
```

Looking at these values, can you tell what base the `Log` method uses?

Since powers of two are important in computer science, we often want logarithms with respect to base 2. To compute them, we can use the formula:

$$\log_2 x = log_e x / log_e 2$$

---

[2]See http://en.wikipedia.org/wiki/Pentium_FDIV_bug.

Changing the `WriteLine` statement to

```
Console.WriteLine(x + "   " + Math.Log(x) / Math.Log(2.0));
```

yields

```
1  0
2  1
3  1.58496250072116
4  2
5  2.32192809488736
6  2.58496250072116
7  2.8073549220576
8  3
9  3.16992500144231
```

We can see that 1, 2, 4 and 8 are powers of two, because their logarithms base 2 are round numbers. If we wanted to find the logarithms of other powers of two, we could modify the program like this:

```
double x = 1.0;
while ( x < 100.0 ) {
    Console.WriteLine(x + "   " + Math.Log(x) / Math.Log(2.0));
    x = x * 2.0;
}
```

Now instead of adding something to `x` each time through the loop, which yields an arithmetic sequence, we multiply `x` by something, yielding a **geometric** sequence. The result is:

```
1   0
2   1
4   2
8   3
16  4
32  5
64  6
```

Log tables may not be useful any more, but for computer scientists, knowing the powers of two is! When you have an idle moment, you should memorize the powers of two up to 65536 (that's 2^{16}).

6.4 Two-dimensional tables

A two-dimensional table consists of rows and columns that make it easy to find values at the intersections. A multiplication table is a good example. Let's say you want to print a multiplication table for the values from 1 to 6.

A good way to start is to write a simple loop that prints the multiples of 2, all on one line.

```
int i = 1;
while ( i <= 6 ) {
    Console.Write(2 * i + "   ");
    i = i + 1;
}
Console.WriteLine();
```

The first line initializes a variable named `i`, which is going to act as a counter, or **loop variable**. As the loop executes, the value of `i` increases from 1 to 6; when `i` is 7, the loop terminates. Each time through the loop, we print the value `2 * i` and three spaces. Since we use `Console.Write`, the output appears on a single line.

In some environments the output from `Write` gets stored without being displayed until `WriteLine` is invoked. If the program terminates, and you forget to invoke `WriteLine`, you may never see the stored output.

The output of this program is:

```
2   4   6   8   10   12
```

So far, so good. The next step is to **encapsulate** and **generalize**.

6.5 Encapsulation and generalization

Encapsulation means taking a piece of code and wrapping it up in a method, allowing you to take advantage of all the things methods are good for. We have seen two examples of encapsulation, when we wrote `WriteParity` in Section 4.3 and `IsSingleDigit` in Section 5.7.

Generalization means taking something specific, like printing multiples of 2, and making it more general, like printing the multiples of any integer.

Here's a method that encapsulates the loop from the previous section and generalizes it to print multiples of `n`.

```
public static void WriteMultiples (int n) {
    int i = 1;
    while ( i <= 6 ) {
        Console.Write(n * i + "   ");
        i = i + 1;
    }
    Console.WriteLine();
}
```

To encapsulate, all I had to do was add the first line, which declares the name, parameter, and return type. To generalize, all I had to do was replace the value 2 with the parameter `n`.

If I invoke this method with the argument 2, I get the same output as before. With argument 3, the output is:

```
3   6   9   12   15   18
```

and with argument 4, the output is

```
4   8   12   16   20   24
```

By now you can probably guess how we are going to print a multiplication table: we'll invoke `WriteMultiples` repeatedly with different arguments. In fact, we are going to use another loop to iterate through the rows.

```
int i = 1;
while ( i <= 6 ) {
    WriteMultiples(i);
    i = i + 1;
}
```

First of all, notice how similar this loop is to the one inside `WriteMultiples`. All I did was replace the print statement with a method invocation.

The output of this program is

```
1   2    3    4    5    6
2   4    6    8    10   12
3   6    9    12   15   18
4   8    12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
```

which is a (slightly sloppy) multiplication table. If the sloppiness bothers you, there is a library method `String.Format` that give you more control over the format of the output, but I'm not going to get into that here.

In Visual Studio you can ask for help on any library function by placing the cursor on the function name and pressing the F1 key. This will open a web browser with the MSDN page describing the function.

If you have auto-complete enabled in Xarmarin Studio, a summary of the function appears next to the function as you type the function name in the auto-complete suggestion list. You can also press the F1 key to view the Xarmarin help file.

6.6   Methods and encapsulation

In Section 3.5 I listed some of the reasons methods are useful. Here are several more:

- By giving a name to a sequence of statements, you make your program easier to read and debug.

- Dividing a long program into methods allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.

- Methods facilitate both recursion and iteration.

- Well-designed methods are often useful for many programs. Once you write and debug one, you can reuse it.

To demonstrate encapsulation again, I'll take the code from the previous section and wrap it up in a method:

```
public static void WriteMultTable() {
    int i = 1;
    while ( i <= 6 ) {
        WriteMultiples(i);
        i = i + 1;
    }
}
```

The development process I am demonstrating is called **encapsulation and generalization**. You start by adding code to `Main` or another method. When you get it working, you extract it and wrap it up in a method. Then you generalize the method by adding parameters.

Sometimes you don't know when you start writing exactly how to divide the program into methods. This process lets you design as you go along.

6.7 Local variables

You might wonder how we can use the same variable `i` in both `WriteMultiples` and `WriteMultTable`. Didn't I say that you can only declare a variable once? And doesn't it cause problems when one of the methods changes the value of the variable?

The answer to both questions is "no," because the `i` in `WriteMultiples` and the `i` in `WriteMultTable` are *not the same variable*. They have the same name, but they do not refer to the same storage location, and changing the value of one has no effect on the other.

Variables declared inside a method definition are called **local variables** because they only exist inside the method. You cannot access a local variable from outside its "home" method, and you are free to have multiple variables with the same name, as long as they are not in the same method.

Although it can be confusing, there are good reasons to reuse names. For example, it is common to use the names `i`, `j` and `k` as loop variables. If you avoid using them in one method just because you used them somewhere else, you make the program harder to read.

6.8 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the 6x6 table. You could add a parameter to `WriteMultTable`:

```
public static void WriteMultTable(int high) {
    int i = 1;
    while ( i <= high ) {
        WriteMultiples(i);
        i = i + 1;
    }
}
```

I replaced the value 6 with the parameter `high`. If I invoke `WriteMultTable` with the argument 7, I get

```
1   2   3   4    5    6
2   4   6   8    10   12
3   6   9   12   15   18
4   8   12  16   20   24
5   10  15  20   25   30
6   12  18  24   30   36
7   14  21  28   35   42
```

which is fine, except that I probably want the table to be square (same number of rows and columns), which means I have to add another parameter to `WriteMultiples`, to specify how many columns the table should have.

I also call this parameter `high`, demonstrating that different methods can have parameters with the same name (just like local variables):

```
public static void WriteMultiples(int n, int high) {
    int i = 1;
    while ( i <= high ) {
        Console.Write(n * i + "   ");
        i = i + 1;
    }
    Console.WriteLine();
}
```

```
    public static void WriteMultTable(int high) {
       int i = 1;
       while ( i <= high ) {
          WriteMultiples(i, high);
          i = i + 1;
       }
    }
```

Notice that when I added a new parameter, I had to change the first line, and I also had to change the place where the method is invoked in `WriteMultTable`. As expected, this program generates a square 7x7 table:

```
1    2    3    4    5    6    7
2    4    6    8    10   12   14
3    6    9    12   15   18   21
4    8    12   16   20   24   28
5    10   15   20   25   30   35
6    12   18   24   30   36   42
7    14   21   28   35   42   49
```

When you generalize a method appropriately, you often find that it has capabilities you did not plan. For example, you might notice that the multiplication table is symmetric, because $ab = ba$, so all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of `WriteMultTable`. Change

```
    WriteMultiples(i, high);
```

to

```
    WriteMultiples(i, i);
```

and you get

```
1
2    4
3    6    9
4    8    12   16
5    10   15   20   25
6    12   18   24   30   36
7    14   21   28   35   42   49
```

I'll leave it up to you to figure out how it works.

6.9   Glossary

loop: A statement that executes repeatedly while some condition is satisfied.

infinite loop: A loop whose condition is always true.

body: The statements inside the loop.

iteration: One pass through (execution of) the body of the loop, including the evaluation of the condition.

encapsulate: To divide a large complex program into components (like methods) and isolate the components from each other (for example, by using local variables).

local variable: A variable that is declared inside a method and that exists only within that method. Local variables cannot be accessed from outside their home method, and do not interfere with any other methods.

generalize: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

program development: A process for writing programs. So far we have seen "incremental development" and "encapsulation and generalization".

6.10 Exercises

Exercise 6.1. Consider the following code:

```
public static void Main(string[] args) {
   Loop(10);
}

public static void Loop(int n) {
   int i = n;
   while ( i > 0 ) {
      Console.WriteLine(i);
      if ( i % 2 == 0 ) {
         i = i / 2;
      } else {
         i = i + 1;
      }
   }
}
```

1. Draw a table that shows the value of the variables `i` and `n` during the execution of `Loop`. The table should contain one column for each variable and one line for each iteration.

2. What is the output of this program?

Exercise 6.2. Consider the following code:

```
static void Main(string[] args) {
    double count = 0.0;
    while ( count != 1.0 ) {
        count = count + 0.1;
        Console.Write("{0:R}", count);
        Console.ReadLine();
    }
}
```

NOTE: You will have to press the ENTER or RETURN key for each iteration of the loop. You will have to use CTRL-C or close the window to stop the program.

1. Why does it not stop when `count` reaches 1.0?

2. How can you alter it to do so?

Exercise 6.3. Let's say you are given a number, a, and you want to find its square root. One way to do that is to start with a very rough guess about the answer, x_0, and then improve the guess using the following formula:

$$x_1 = (x_0 + a/x_0)/2$$

For example, if we want to find the square root of 9, and we start with $x_0 = 6$, then $x_1 = (6 + 9/6)/2 = 15/4 = 3.75$, which is closer.

We can repeat the procedure, using x_1 to calculate x_2, and so on. In this case, $x_2 = 3.075$ and $x_3 = 3.00091$. So that is converging very quickly on the right answer(which is 3).

Write a method called `SquareRoot` that takes a `double` as a parameter and that returns an approximation of the square root of the parameter, using this technique. You may not use `Math.Sqrt`.

As your initial guess, you should use $a/2$. Your method should iterate until it gets two consecutive estimates that differ by less than 0.0001; in other words, until the absolute value of $x_n - x_{n-1}$ is less than 0.0001. You can use `Math.Abs` to calculate the absolute value.

Exercise 6.4. In Exercise 5.8 we wrote a recursive version of `Power`, which takes a double `x` and an integer `n` and returns x^n. Now write an iterative method to perform the same calculation.

Exercise 6.5. Section 5.8 presents a recursive method that computes the factorial function. Write an iterative version of `Factorial`.

Chapter 7

Strings and things

7.1 Characters

In object-oriented languages, **objects** are collections of related data that come with a set of methods. These methods operate on the objects, performing computations and sometimes modifying the object's data.

strings are objects, so you might ask "What is the data contained in a string object?" and "What are the methods we can invoke on string objects?" The components of a string object are letters or, more generally, characters. Not all characters are letters; some are numbers, symbols, and other things. For simplicity I call them all letters. There are many methods, but I use only a few in this book. The rest are documented on the MSDN website[1].

As a string is just a collection of characters a string is defined using the string variable type. The char variable type is used to store a single character, as opposed to a string of characters).

chars work just like the other types we have seen:

```
char letter = 'c';
if ( letter == 'c' ) {
   Console.WriteLine(letter);
}
```

Character values appear in single quotes, like 'c'. Unlike string values (which appear in double quotes), character values can contain only a single letter or symbol.

I can access an individual character in a string using the [] operator:

```
string fruit = "banana";
char letter = fruit[1];
Console.WriteLine(letter);
```

---

[1] http://msdn.microsoft.com/en-gb/library/362314fe.aspx.

`fruit[1]` means that I want to know the first letter of the string in the `fruit` variable. The result is a character, which is stored in a `char` named `letter`. When I print the value of `letter`, I get a surprise:

`a`

`a` is not the first letter of `"banana"`. Unless you are a computer scientist. For technical reasons, computer scientists start counting from zero. The 0th letter ("zeroeth") of `"banana"` is `b`. The 1th letter ("oneth") is `a` and the 2th ("twoth") letter is `n`.

If you want the zeroeth letter of a string, you have to put the 0 in the `[]` operator:

```
char letter = fruit[0];
```

7.2 Length

The next `string` thing we'll look at is `Length`, which is a attribute of a string, in this case the number of characters in the string. For example:

```
int Length = fruit.Length;
```

`Length` returns an integer, in this case 6. Notice that it is legal to have a variable with the same name as a method (although it can be confusing for human readers).

This is the reason we use "camel caps" for variable names. This is just a cute name for the first letter being in lower case and upper case letters are used to separate words, `jammingWordsTogetherLikeThis`).

To find the last letter of a string, you might be tempted to try something like

```
int length = fruit.Length;
char last = fruit[length];        // WRONG!!
```

That won't work. The reason is that there is no 6th letter in `"banana"`. Since we started counting at 0, the 6 letters are numbered from 0 to 5. To get the last character, you have to subtract 1 from `length`.

```
int length = fruit.Length;
char last = fruit[length-1];
```

7.3 Traversal

A common thing to do with a string is start at the beginning, select each character in turn, do some computation with it, and continue until the end. This pattern of processing is called a **traversal**. A natural way to encode a traversal is with a `while` statement:

```
int index = 0;
while ( index < fruit.Length ) {
   char letter = fruit[index];
   Console.WriteLine(letter);
   index = index + 1;
}
```

This loop traverses the string and prints each letter on a line by itself. Notice that the condition is `index < fruit.Length`, which means that when `index` is equal to the length of the string, the condition is false and the body of the loop is not executed. The last character we access is the one with the index `fruit.Length - 1`.

The name of the loop variable is `index`. An **index** is a variable or value used to specify a member of an ordered set, in this case the string of characters. The index indicates (hence the name) which one you want.

7.4 Run-time errors

Way back in Section 1.3.2 I talked about run-time errors, which are errors that don't appear until a program has started running. Run-time errors are called **exceptions**.

You probably haven't seen many run-time errors, because we haven't been doing many things that can cause one. Well, now we are. If you provide an index that is negative or greater than `length - 1` in the square brackets, it **throws** an exception. You can think of "throwing" an exception like throwing a tantrum.

When that happens, an error message with the type of exception and a **stack trace**, which shows the methods that were running when the exception occurred. Here is an example:

```csharp
using System;

namespace ThinkSharp {
  public class BadString {
    public static void Main(string[] args) {
      ProcessWord("banana");
    }

    public static void ProcessWord(string s) {
      char c = GetLastLetter(s);
      Console.WriteLine(c);
    }

    public static char GetLastLetter(string s) {
      int index = s.Length;        // WRONG!
      char c = s[index];
      return c;
    }
  }
}
```

Notice the error in `GetLastLetter`: the index of the last character should be `s.Length()` `- 1`. Here's what you get:

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of
the array.
  at System.String.get_Chars(Int32 index)
  at ThinkSharp.BadString.GetLastLetter(String s) in c:\ThinkSharp\BadString.cs:line 16
  at ThinkSharp.BadString.ProcessWord(String s) in c:\ThinkSharp\BadString.cs:line 10
  at ThinkSharp.BadString.Main(String[] args) in c:\ThinkSharp\BadString.cs:line 6
```

Then the program ends. The stack trace can be hard to read, but it contains a lot of information.

In Visual Studio, the line will be highlighted and an exception message box will be displayed with the title "IndexOutOfRangeException was unhandled." The stack trace is given in the "Call Stack" window.

In Xamarin Studio, the line is also highlighted and an exception box is displayed with the text "System.IndexOutOfRangeException has been thrown." The stack trace is presented in the "Call Stack" window.

7.5 The `IndexOf` method

`IndexOf` is the inverse of the `[]` operator: this operator takes an index and returns the character at that index; `IndexOf` takes a character and finds the index where that character appears.

The `[]` operator fails if the index is out of range, and throws an exception. `IndexOf` fails if the character does not appear in the string, and returns the value `-1`.

```
string fruit = "banana";
int index = fruit.IndexOf('a');
```

`fruit.IndexOf()` means that I am invoking the `IndexOf` method on the object named `fruit`. I am passing the argument `'a'` to this method, which means that I want to know index of the letter `'a'` in the string. In this case, the letter appears three times, so it is not obvious what `IndexOf` should do. According to the documentation, it returns the index of the *first* appearance.

To find subsequent appearances, there is another version of the `IndexOf` method. It takes a second argument that indicates where in the string to start looking. For an explanation of this kind of overloading, see Section 5.4.

If we invoke:

```
int index = fruit.IndexOf('a', 2);
```

it starts at the twoeth letter (the first `n`) and finds the second `a`, which is at index 3. If the letter happens to appear at the starting index, the starting index is the answer. So

```
int index = fruit.indexOf('a', 5);
```

returns 5.

7.6   Reading documentation

The Microsoft Developer Network[2] (MSDN) provides documentation for most, if not all, of the standard functions. Go to the MSDN and type "IndexOf" into the search box. This will show a list of over 1,900,000 places, where the MSDN metions the `IndexOf` function. Select the "**String.IndexOf** Method (System) - MSDN – the Microsoft ..." entry.

> If you place the cursor on the text `IndexOf` and press the F1 key, Visual Studio will bring up the MSDN documentation for the `IndexOf` method.

If the method is overloaded, as in the case, a list of the different methods available will be shown. Selecting the second item in the list, "String.IndexOf(Char,Int32)", will take you to the page that describes this version of the `IndexOf` method[3]. You get the documentation shown in figure 7.1 (or something like it).

The first line is the method's **prototype**, which specifies the name of the method, the type of the parameters, and the return type. The next line describes what the method does.

The Syntax section provides a more detailed description of the parameters and the return value. The Remarks sections provides further discussion, including a list of any exceptions the method might throw. The documentation is supposed to fit a standard format, it might take some time to get comfortable with this kind of documentation, but it is worth the effort.

7.7   Looping and counting

The following program counts the number of times the letter 'a' appears in a string:

```
string fruit = "banana";
int length = fruit.Length;
int count = 0;

int index = 0;
while ( index < length ) {
```

---

[2]http://msdn.microsoft.com/en-gb/library
[3]http://msdn.microsoft.com/en-gb/library/cc506244.aspx
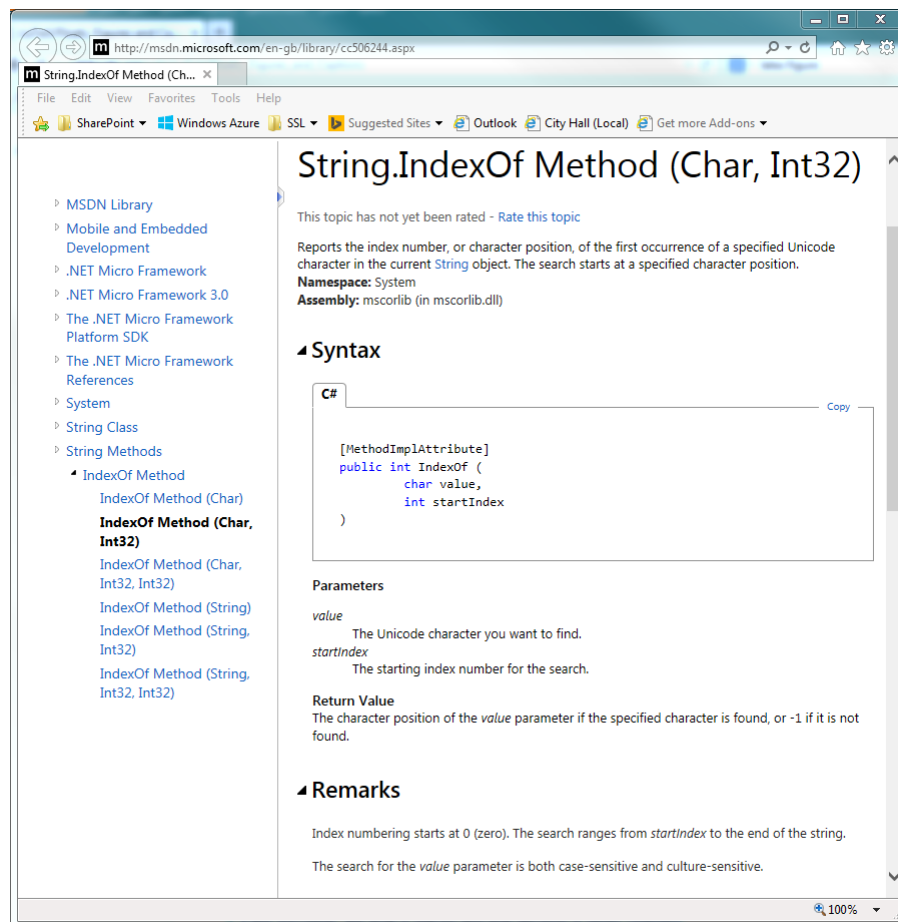
Figure 7.1: MSDN Documentation of `IndexOf`

```
    if ( fruit[index] == 'a' ) {
        count = count + 1;
    }
    index = index + 1;
}
Console.WriteLine(count);
```

This program demonstrates a common idiom, called a **counter**. The variable `count` is initialized to zero and then incremented each time we find an `'a'`. To **increment** is to increase by one; it is the opposite of **decrement**. When we exit the loop, `count` contains the result: the total number of a's.

7.8   Increment and decrement operators

Incrementing and decrementing are such common operations that special operators are provided for them. The `++` operator adds one to the current value of an `int`, `char` or `double`. `--` subtracts one. Neither operator works on `bool`s or `string`s.

Technically, it is legal to increment a variable and use it in an expression at the same time. For example, you might see something like:

```
Console.WriteLine(i++);
```

Looking at this, it is not clear whether the increment will take effect before or after the value is printed. Because expressions like this tend to be confusing, I discourage you from using them. In fact, to discourage you even more, I'm not going to tell you what the result is. If you really want to know, you can try it.

Using the increment operators, we can rewrite the letter-counter:

```
int index = 0;
while ( index < length ) {
    if ( fruit[index] == 'a' ) {
        count++;
    }
    index++;
}
```

It is a common error to write something like

```
index = index++;            // WRONG!!
```

Unfortunately, this is syntactically legal, so the compiler will not warn you. The effect of this statement is to leave the value of `index` unchanged. This is often a difficult bug to track down.

Remember, you can write `index = index + 1`, or you can write `index++`, but you shouldn't mix them.

7.9   Strings are immutable

As you read the documentation of the `string` methods, you might notice `ToUpper` and `ToLower`. These methods are often a source of confusion, because it sounds like they have the effect of changing (or mutating) an existing string. Actually, neither these methods nor any others can change a string, because strings are **immutable**.

When you invoke `ToUpper` on a `string`, you get a *new* `string` as a return value. For example:

```
string name = "Alan Turing";
string upperName = name.ToUpper();
```

After the second line is executed, `upperName` contains the value `"ALAN TURING"`, but `name` still contains `"Alan Turing"`.

7.10   Strings are incomparable

It is often necessary to compare strings to see if they are the same. We can use the equality operator (==) to compare if two string are the same. However, we can not use the < or > operators to see which comes first in alphabetical order.

To compare strings, we have to use the CompareTo method. For example:

```
string name1 = "Alan Turing";
string name2 = "Ada Lovelace";

int flag = name1.CompareTo(name2);
if ( flag == 0 ) {
   Console.WriteLine("The names are the same.");
} else if ( flag < 0 ) {
   Console.WriteLine("name1 comes before name2.");
} else if ( flag > 0 ) {
   Console.WriteLine("name2 comes before name1.");
}
```

The syntax here is a little weird. To compare two strings, you have to invoke a method on one of them and pass the other as an argument.

The return value from is a weird, too. If the strings are equal, it is 0. If the first string (the one on which the method is invoked) comes first in the alphabet, the difference is negative. Otherwise, it is positive. The intention is for the < and > operators to have their normal meaning when applied to the return value. I.e.,

value	meaning
-1	name1 < name2
0	name1 = name2
1	name1 > name2

7.11   Glossary

object: A collection of related data that comes with a set of methods that operate on it. The objects we have used so far are strings, Math and Console.

index: A variable or value used to select one of the members of an ordered set, like a character from a string.

exception: A run-time error.

throw: Cause an exception.

stack trace: A report that shows the state of a program when an exception occurs.

prototype: The first line of a method, which specifies the name, parameters and return
type.

traverse: To iterate through all the elements of a set performing a similar operation on
each.

counter: A variable used to count something, usually initialized to zero and then incre-
mented.

increment: Increase the value of a variable by one (the `++` operator).

decrement: Decrease the value of a variable by one (the `--` operator).

7.12   Exercises

Exercise 7.1. Write a method that takes a `string` as an argument and that prints the
letters backwards all on one line.

Exercise 7.2. Read the stack trace in Section 7.4 and answer these questions:

- What kind of Exception occurred, and what name space is it defined in?

- What method threw the exception, and where is that method defined?

- What method invoked the `[]` operator?

- In `BadString.cs`, what is the line number where the `[]` operator was invoked?

Exercise 7.3. Encapsulate the code in Section 7.7 in a method named `CountLetters`,
and generalize it so that it accepts the string and the letter as arguments.

Then rewrite the method so that it uses `IndexOf` to locate the a's, rather than checking
the characters one by one.

Exercise 7.4. The purpose of this exercise is to review encapsulation and generalization.

1. Encapsulate the following code fragment, transforming it into a method that takes a
   string as an argument and that returns the final value of `count`.

2. In a sentence or two, describe what the resulting method does (without getting into
   the details of how).

3. Now that you have generalized the code so that it works on any string, what could
   you do to generalize it more?

```csharp
string s = "((3 + 7) * 2)";
int len = s.Length;

int i = 0;
int count = 0;

while ( i < len ) {
    char c = s[i];

    if ( c == '(' ) {
        count = count + 1;
    } else if ( c == ')' ) {
        count = count - 1;
    }
    i = i + 1;
}

Console.WriteLine(count);
```

Exercise 7.5. The point of this exercise is to explore types and fill in some of the details that aren't covered in the chapter.

1. Create a new program named `Test.cs` and write a `Main` method that contains expressions that combine various types using the `+` operator. For example, what happens when you "add" a `string` and a `char`? Does it perform addition or concatenation? What is the type of the result? (How can you determine the type of the result?)

2. Make a bigger copy of the following table and fill it in. At the intersection of each pair of types, you should indicate whether it is legal to use the `+` operator with these types, what operation is performed (addition or concatenation), and what the type of the result is.

	bool	char	int	string
bool				
char				
int				
string				

3. Think about some of the choices the designers of C# made when they filled in this table. How many of the entries seem unavoidable, as if there were no other choice? How many seem like arbitrary choices from several equally reasonable possibilities? How many seem problematic?

4. Here's a puzzler: normally, the statement `x++` is exactly equivalent to `x = x + 1`. But if `x` is a `char`, it's not! In that case, `x++` is legal, but `x = x + 1` causes an error. Try it out and see what the error message is, then see if you can figure out what is going on.

Exercise 7.6. What is the output of this program? Describe in a sentence what `Mystery` does (not how it works).

```
using System;

namespace ThinkSharp {
   public class MysteryClass {

      public static string Mystery(string s) {
         int i = s.Length - 1;
         string total = "";

         while ( i >= 0 ) {
            char ch = s[i]);
            Console.WriteLine(i + "      " + ch);

            total = total + ch;
            i--;
         }
         return total;
      }

      public static void Main(string[] args) {
         Console.WriteLine(Mystery("Allen"));
      }
   }
}
```

Exercise 7.7. A friend of yours shows you the following method and explains that if `number` is any two-digit number, the program will output the number backwards. He claims that if `number` is 17, the method will output 71.

Is he right? If not, explain what the program actually does and modify it so that it does the right thing.

```
int number = 17;
int lastDigit = number % 10;
int firstDigit = number / 10;
Console.WriteLine(lastDigit + firstDigit);
```

Exercise 7.8. What is the output of the following program?

```
using System;

namespace ThinkSharp {
  public class EnigmaClass {

    public static void Enigma(int x) {
      if ( x == 0 ) {
        return;
      } else {
        Enigma(x / 2);
      }

      Console.WriteLine(x % 2);
    }

    public static void Main(string[] args) {
      Enigma(5);
      Console.WriteLine();
    }
  }
}
```

Explain in 4-5 words what the method `Enigma` really does.

Exercise 7.9.

1. Create a new program named `Palindrome.cs`.

2. Write a method named `First` that takes a string and returns the first letter, and one named `Last` that returns the last letter.

3. Write a method named `Middle` that takes a string and returns a substring that contains everything *except* the first and last characters.

   HINT: read the documentation of the `Substring` method in the `string` class. Run a few tests to make sure you understand how `Substring` works before you try to write `Middle`.

   What happens if you invoke `Middle` on a string that has only two letters? One letter? No letters?

4. The usual definition of a palindrome is a word that reads the same both forward and backward, like "otto" and "palindromeemordnilap." An alternative way to define a property like this is to specify a way of testing for the property. For example, we might say, "a single letter is a palindrome, and a two-letter word is a palindrome if

the letters are the same, and any other word is a palindrome if the first letter is the same as the last and the middle is a palindrome."

Write a recursive method named `IsPalindrome` that takes a string and that returns a boolean indicating whether the word is a palindrome or not.

5. Once you have a working palindrome checker, look for ways to simplify it by reducing the number of conditions you check. HINT: it might be useful to adopt the definition that the empty string is a palindrome.

6. On a piece of paper, figure out a strategy for checking palindromes iteratively. There are several possible approaches, so make sure you have a solid plan before you start writing code.

7. Implement your strategy in a method called `IsPalindromeIter`.

8. Optional: Appendix B provides code for reading a list of words from a file. Read a list of words and print the palindromes.

Exercise 7.10. A word is said to be "abecedarian" if the letters in the word appear in alphabetical order. For example, the following are all 6-letter English abecedarian words.

abdest, acknow, acorsy, adempt, adipsy, agnosy, befist, behint, beknow, bijoux, biopsy, cestuy, chintz, deflux, dehors, dehort, deinos, diluvy, dimpsy

1. Describe a process for checking whether a given word (string) is abecedarian, assuming that the word contains only lower-case letters. Your process can be iterative or recursive.

2. Implement your process in a method called `IsAbecedarian`.

Exercise 7.11. A dupledrome is a word that contains only double letters, like "llaammaa" or "ssaabb". I conjecture that there are no dupledromes in common English use. To test that conjecture, I would like a program that reads words from the dictionary one at a time and checks them for dupledromity.

Write a method called `IsDupledrome` that takes a string and returns a boolean indicating whether the word is a dupledrome.

Exercise 7.12.

1. The Captain Crunch decoder ring works by taking each letter in a string and adding 13 to it. For example, 'a' becomes 'n' and 'b' becomes 'o'. The letters "wrap around" at the end, so 'z' becomes 'm'.

Write a method that takes a string and that returns a new string containing the encoded version. You should assume that the string contains upper and lower case letters, and spaces, but no other punctuation. Lower case letters should be transformed into other lower case letters; upper into upper. You should not encode the spaces.

2. Generalize the Captain Crunch method so that instead of adding 13 to the letters, it adds any given amount. Now you should be able to encode things by adding 13 and decode them by adding -13. Try it.

Chapter 8

Mutable objects

Strings are objects, but they are atypical objects because

- They are immutable.

- You don't have to use `new` to create one.

In this chapter, we use two objects from the graphics library, `Point` and `Rectangle`. But first, I want to make it clear that these points and rectangles are not graphical objects that appear on the screen. They are values that contain data, just like `int`s and `double`s. Like other values, they are used internally to perform computations.

8.1   Name spaces

The libraries are divided into **name spaces**, including `System`, which contains most of the classes we have used so far, and the `System.Drawing` name space, which contains classes for windows, buttons, graphics, etc.

To use a class defined in another name space, you have to declare that you are **using** it. `Point` and `Rectangle` are in the `System.Drawing` name space, so to use them like this:

`using System.Drawing;`

All `using` statements appear at the beginning of the program, outside the class definition.

8.2   Assemblies

The libraries are spread across a number of different **assemblies**. All of the classes we have been using so far (`Math`, `String` and `Console`) are included in the standard `System.Core`

assembly. The `Point` and `Rectangle` classes are provided by the `System.Drawing` assembly. This assembly is not included in the project by default, so we will need to include it.

⊠ Find the "Solution Explorer" and open your project. Find the References entry in the project. Right-click on this entry and select the Add Reference. . . item. This will bring up the "Reference Manager" dialogue for the project.

In the reference manager, open the Assemblies option and select the Framework item. This will update the main panel showing all of the assemblies available to the .Net framework. Scan down this list until you find the System.Drawing assembly. Tick the check box to the left of the assembly name. Click on the OK button.

⊗ Find the project in the "Solution" window. Right-click on the References entry, select the Edit References. . . item. This will bring up the "Edit References" dialogue. Select All and scan down the list for the System.Drawing assembly. Click on the tick box next to the assembly name and click on the OK button.

You will see the `System.Drawing` assembly appear in the list of libraries under the References entry in the project.

8.3   `Point` objects

A point is two numbers (coordinates) that we treat collectively as a single object. In mathematical notation, points are often written in parentheses, with a comma separating the coordinates. For example, $(0, 0)$ indicates the origin, and (x, y) indicates the point x units to the right and y units up from the origin.

A point is represented by a `Point` object. To create a new point, you have to use `new`:

```
Point blank;
blank = new Point(3, 4);
```

The first line is a conventional variable declaration: `blank` has type `Point`. The second line invokes `new`, specifies the type of the new object, and provides arguments. The arguments are the coordinates of the new point, $(3, 4)$.

The result of `new` is a **reference** to the new point, so `blank` contains a reference to the newly-created object. There is a standard way to diagram this assignment, shown in the figure.
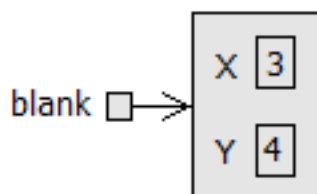
As usual, the name of the variable `blank` appears outside the box and its value appears inside the box. In this case, that value is a reference, which is shown graphically with an arrow. The arrow points to the object we're referring to.

The big box shows the newly-created object with the two values in it. The names `X` and `Y` are the names of the **fields**.

Taken together, all the variables, values, and objects in a program are called the **state**. Diagrams like this that show the state of the program are called **state diagrams**. As the program runs, the state changes, so you should think of a state diagram as a snapshot of a particular point in the execution.

8.4   Fields

The pieces of data that make up an object are called **fields**. Although they are commonly called **instance variables** because each object, which is an **instance** of its type, has its own copy of the instance variables.

It's like the glove compartment of a car. Each car is an instance of the type "car," and each car has its own glove compartment. If you ask me to get something from the glove compartment of your car, you have to tell me which car is yours.

Similarly, if you want to read a value from a field, you have to specify the object you want to get it from. This is done using "dot notation."

```
int x = blank.X;
```

The expression `blank.x` means "go to the object `blank` refers to, and get the value of `X`." In this case we assign that value to a local variable named `x`. There is no conflict between the local variable named `x` and any field named `x`. The purpose of dot notation is to identify *which* variable you are referring to unambiguously.

You can use dot notation as part of any expression, so the following are legal.

```
Console.WriteLine(blank.X + ", " + blank.Y);
int distance = blank.X * blank.X + blank.Y * blank.Y;
```

The first line prints `3, 4`; the second line calculates the value 25.

8.5   Objects as parameters

You can pass objects as parameters in the usual way. For example:

```
public static void WritePoint(Point p) {
    Console.WriteLine("(" + p.X + ", " + p.Y + ")");
}
```

This method takes a point as an argument and prints it in the standard format. If you invoke `WritePoint(blank)`, it prints `(3, 4)`. Actually, the system already has a method for printing `Point`s. If you invoke `Console.WriteLine(blank)`, you get

`(X=3,Y=4)`

As a second example, we can rewrite the `Distance` method from Section 5.2 so that it takes two `Point`s as parameters instead of four `double`s.

```
public static double Distance(Point p1, Point p2) {
    double dx = (double)(p2.X - p1.X);
    double dy = (double)(p2.Y - p1.Y);
    return Math.Sqrt(dx * dx + dy * dy);
}
```

The typecasts are not really necessary; I added them as a reminder that the fields in a `Point` are integers.

8.6 Rectangles

`Rectangle`s are similar to points, except that they have four fields: `X`, `Y`, `Width` and `Height`. Other than that, everything is pretty much the same.

This example creates a `Rectangle` object and makes `box` refer to it.

```
Rectangle box = new Rectangle(0, 0, 100, 200);
```

This figure shows the effect of this assignment.
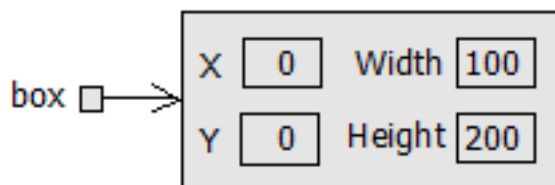
If you print `box`, you get

`(X=0,Y=0,Width=100,Height=200)`

This is the result of a method that knows how to write `Rectangle` objects.

8.7 Objects as return types

You can write methods that return objects. For example, `FindCentre` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the centre of the `Rectangle`:

```
public static Point FindCentre(Rectangle box) {
    int x = box.X + box.Width / 2;
    int y = box.Y + box.Height / 2;
    return new Point(x, y);
}
```

Notice that you can use `new` to create a new object, and then immediately use the result as the return value.

8.8   Objects are mutable

You can change the contents of an object by making an assignment to one of its fields. For example, to "move" a rectangle without changing its size, you can modify the X and Y values:

```
box.X = box.X + 50;
box.Y = box.Y + 100;
```

The result is shown in the figure:
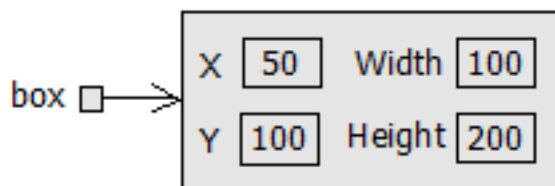
We can encapsulate this code in a method and generalize it to move the rectangle by any amount:

```
public static void MoveRect(Rectangle box, int dx, int dy) {
    box.X = box.X + dx;
    box.Y = box.Y + dy;
}
```

The variables `dx` and `dy` indicate how far to move the rectangle in each direction. Invoking this method has the effect of modifying the `Rectangle` that is passed as an argument.

```
Rectangle box = new Rectangle(0, 0, 100, 200);
MoveRect(box, 50, 100);
Console.WriteLine(box);
```

outputs (X=50,Y=100,Width=100,Height=200).

Modifying objects by passing them as arguments to methods can be useful, but it can also make debugging more difficult because it is not always clear which method invocations do or do not modify their arguments. Later, I discuss some pros and cons of this programming style.

C# provides methods that operate on `Points` and `Rectangle`s.

Place the cursor on the class name and press the F1 key to access the documentation for the class.

Place the cursor on the variable name and press the F1 key to access the documentation for the variables class. Select the **Members** option.

For example, `Offset` has the same effect as `MoveRect`, but instead of passing the Rectangle as an argument, you use dot notation:

```
box.Offset(50, 100);
```

8.9 Aliasing

Remember that when you assign an object to a variable, you are assigning a *reference* to an object. It is possible to have multiple variables that refer to the same object. For example, this code:

```
Rectangle box1 = new Rectangle(0, 0, 100, 200);
Rectangle box2 = box1;
```

generates a state diagram that looks like this:
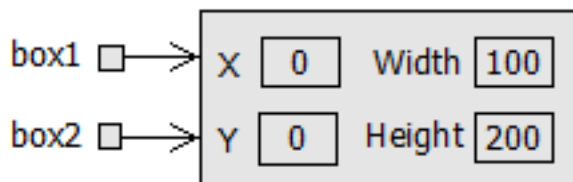
`box1` and `box2` refer to the same object. In other words, this object has two names, `box1` and `box2`. When a person uses two names, it's called **aliasing**. Same thing with objects.

When two variables are aliased, any changes that affect one variable also affect the other. For example:

```
Console.WriteLine(box2.Width);
box1.Inflate(50, 50);
Console.WriteLine(box2.Width);
```

The first line prints 100, which is the width of the `Rectangle` referred to by `box2`. The second line invokes the `Inflate` method on `box1`, which expands the `Rectangle` by 50 pixels in every direction (see the documentation for more details). The effect is shown in the figure:
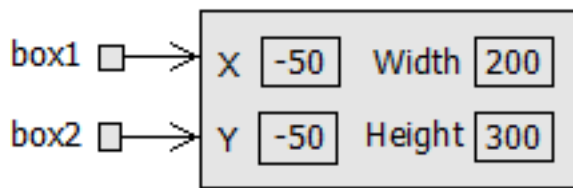
Whatever changes are made to `box1` also apply to `box2`. Thus, the value printed by the third line is `200`, the width of the expanded rectangle. (As an aside, it is perfectly legal for the coordinates of a `Rectangle` to be negative.)

As you can tell even from this simple example, code that involves aliasing can get confusing fast, and can be difficult to debug. In general, aliasing should be avoided or used with care.

8.10 `null`

When you create an object variable, remember that you are creating a *reference* to an object. Until you make the variable point to an object, the value of the variable is `null`. `null` is a special value (and a keyword) that means "no object."

The declaration `Point blank;` is equivalent to this initialization

```
Point blank = null;
```

and is shown in the following state diagram:

The value `null` is represented by a small square with no arrow.

If you try to use a null object, either by accessing a field or invoking a method, the system throws a `NullReferenceException`, prints an error message and terminates the program.

```
Point blank = null;
int x = blank.X;       // NullReferenceException
```

On the other hand, it is legal to pass a `null` object as an argument or receive one as a return value. In fact, it is common to do so, for example to represent an empty set or indicate an error condition.

8.11 Garbage collection

In Section 8.9 we talked about what happens when more than one variable refers to the same object. What happens when *no* variable refers to an object? For example:

```
Point blank = new Point(3, 4);
blank = null;
```

The first line creates a new `Point` object and makes `blank` refer to it. The second line changes `blank` so that instead of referring to the object, it refers to nothing (the `null` object).
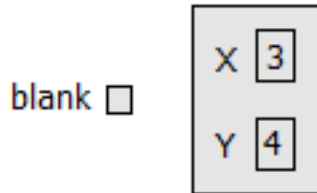
If no one refers to an object, then no one can read or write any of its values, or invoke a method on it. In effect, it ceases to exist. We could keep the object in memory, but it would only waste space, so periodically as your program runs, the system looks for stranded objects and reclaims them, in a process called **garbage collection**. Later, the memory space occupied by the object will be available to be used as part of a new object.

You don't have to do anything to make garbage collection happen, and in general you will not be aware of it. But you should know that it periodically runs in the background.

8.12   Objects and primitives

There are two kinds of types, primitive types and object types. Primitives, like `int` and `bool` begin with lower-case letters; object types begin with upper-case letters. This distinction is useful because it reminds us of some of the differences between them:

- When you declare a primitive variable, you get storage space for a primitive value. When you declare an object variable, you get a space for a reference to an object. To get space for the object itself, you have to use `new`.

- If you don't initialize a primitive type, it is given a default value that depends on the type. For example, `0` for `int`s and `false` for `bool`s. The default value for object types is `null`, which indicates no object.

- Primitive variables are well isolated in the sense that there is nothing you can do in one method that will affect a variable in another method. Object variables can be tricky to work with because they are not as well isolated. If you pass a reference to an object as an argument, the method you invoke might modify the object, in which case you will see the effect. Of course, that can be a good thing, but you have to be aware of it.

There is one other difference between primitives and object types. You cannot add new primitives (unless you get yourself on the standards committee), but you can create new object types! We'll see how in the next chapter.

8.13 Glossary

name space: A collection of classes. Classes are organized into name spaces.

assembly: A library of name spaces, Assemblies are libraries of code that contain name spaces.

field: One of the named data items that make up an object. Each object (instance) has its own copy of the fields for its class.

instance: An example from a category. My cat is an instance of the category "feline things." Every object is an instance of some class.

instance variable: Another name for a field.

reference: A value that indicates an object. In a state diagram, a reference appears as an arrow. Older programmers refer to these as **pointer**s.

aliasing: The condition when two or more variables refer to the same object.

garbage collection: The process of finding objects that have no references and reclaiming their storage space.

state: A complete description of all the variables and objects and their values, at a given point during the execution of a program.

state diagram: A snapshot of the state of a program, shown graphically.

8.14 Exercises

Exercise 8.1.

1. For the following program, draw a stack diagram showing the local variables and parameters of `Main` and `Riddle`, and show any objects those variables refer to.

2. What is the output of this program?

```
public static void Main(string[] args) {
    int x = 5;
    Point blank = new Point(1, 2);

    Console.WriteLine(Riddle(x, blank));
    Console.WriteLine(x);
    Console.WriteLine(blank.X);
    Console.WriteLine(blank.Y);
```

```
    }

    public static int Riddle(int x, Point p) {
        x = x + 7;
        return x + p.X + p.Y;
    }
```

The point of this exercise is to make sure you understand the mechanism for passing Objects as parameters.

Exercise 8.2.

1. For the following program, draw a stack diagram showing the state of the program just before `Distance` returns. Include all variables and parameters and the objects those variables refer to.

2. What is the output of this program?

```
    public static double Distance(Point p1, Point p2) {
        int dx = p1.X - p2.X;
        int dy = p1.Y - p2.Y;
        return Math.Sqrt(dx * dx + dy * dy);
    }

    public static Point FindCentre(Rectangle box) {
        int x = box.X + box.Width / 2;
        int y = box.Y + box.Height / 2;
        return new Point(x, y);
    }

    public static void Main(string[] args) {
        Point blank = new Point(5, 8);

        Rectangle rect = new Rectangle(0, 2, 4, 4);
        Point centre = FindCentre(rect);

        double dist = Distance(centre, blank);

        Console.WriteLine(dist);
    }
```

Exercise 8.3. The method `Inflate` is part of the `Rectangle` class. Use the `F1` key to read the MSDN documentation.

1. What is the output of the following program?

2. Draw a state diagram that shows the state of the program just before the end of `Main`. Include all local variables and the objects they refer to.

3. At the end of `Main`, are `p1` and `p2` aliased? Why or why not?

```
public static void WritePoint(Point p) {
    Console.WriteLine("(" + p.X + ", " + p.Y + ")");
}

public static Point FindCentre(Rectangle box) {
    int x = box.X + box.Width / 2;
    int y = box.Y + box.Height / 2;
    return new Point(x, y);
}

public static void Main(string[] args) {
    Rectangle box1 = new Rectangle(2, 4, 7, 9);
    Point p1 = FindCentre(box1);
    WritePoint(p1);

    box1.Inflate(1, 1);
    Point p2 = FindCentre(box1);
    WritePoint(p2);
}
```

Exercise 8.4. You might be sick of the factorial method by now, but we're going to do one more version.

1. Create a new program called `Big.cs` and write an iterative version of `Factorial`.

2. Print a table of the integers from 0 to 30 along with their factorials. At some point around 15, you will probably see that the answers are not right any more. Why not?

3. `BigInteger`s are objects that can represent arbitrarily big integers. There is no upper bound except the limitations of memory size and processing speed. Read the documentation of BigIntegers[1].

4. To use BigIntegers, you have to add `using System.Numerics;` to the beginning of your program and the `System.Numerics` assembly to the project references.

5. To create a new BigInteger you must pass a value to the `new BigInteger()` method. The following code converts an integer to a `BigInteger`:

   ```
   int x = 17;
   BigInteger big = new.BigInteger(x);
   ```

   Type in this code and try it out. Try printing a BigInteger.

---

[1] http://msdn.microsoft.com/en-gb/library/vstudio/system.numerics.aspx

6. Because BigIntegers are not primitive types, the usual maths operators don't work. Instead we have to use methods like `Add`. To add two BigIntegers, invoke `Add` on the BigInteger class passing both values as the arguments. For example:

```
BigInteger small = new BigInteger(17);
BigInteger big   = new BigInteger(1700000000);
BigInteger total = BigInteger.Add(small, big);
```

Try out some of the other methods, like `Multiply` and `Pow`.

7. Convert `Factorial` so that it performs its calculation using BigIntegers and returns a BigInteger as a result. You can leave the parameter alone—it will still be an integer.

8. Try printing the table again with your modified factorial method. Is it correct up to 30? How high can you make it go? I calculated the factorial of all the numbers from 0 to 999, but my machine is pretty slow, so it took a while. The last number, 999!, has 2565 digits.

Exercise 8.5. Many encryption techniques depend on the ability to raise large integers to an integer power. Here is a method that implements a (reasonably) fast technique for integer exponentiation:

```
public static int Pow(int x, int n) {
    if ( n == 0 ) return 1;

    // find x to the n/2 recursively
    int t = Pow(x, n / 2);

    // if n is even, the result is t squared
    // if n is odd, the result is t squared times x

    if ( n % 2 == 0 ) {
        return t * t;
    } else {
        return t * t * x;
    }
}
```

The problem with this method is that it only works if the result is smaller than 2 billion. Rewrite it so that the result is a `BigInteger`. The parameters should still be integers, though.

You can use the BigInteger methods `Add` and `Multiply`, but don't use `Pow`, which would spoil the fun.

Chapter 9

Create your own objects

9.1 Class definitions and object types

Way back in Section 1.5 when we defined the class `Hello`, we also created an object type named `Hello`. We didn't create any variables with type `Hello`, and we didn't use `new` to create any `Hello` objects, but we could have!

That example doesn't make much sense, since there is no reason to create a `Hello` object, and it wouldn't do much if we did. In this chapter, we will look at class definitions that create *useful* object types.

Here are the most important ideas in this chapter:

- Defining a new class also creates a new object type with the same name.

- A class definition is like a template for objects: it determines what fields the objects have and what methods can operate on them.

- Every object belongs to some object type; that is, it is an instance of some class.

- When you invoke `new` to create an object, a special method called a **constructor** is invoked to initialize the fields. You provide one or more constructors as part of the class definition.

- The methods that operate on a type are defined in the class definition for that type.

Here are some syntax issues about class definitions:

- Class names (and hence object types) should begin with a capital letter, which helps distinguish them from primitive types and variable names.

- You usually put one class definition in each file, and the name of the file must be the same as the name of the class, with the suffix `.cs`. For example, the `Time` class is defined in the file named `Time.cs`.

- In any program or project, one class is designated as the **startup class**. The startup class must contain a method named `Main`, which is where the execution of the program begins. Other classes *must not* have a method named `Main`.

- In any solution, one project is designated as the **startup project**. The startup project must contain the startup class.

With those issues out of the way, let's look at an example of a user-defined class, `Time`.

9.2   Time

A common motivation for creating an object type is to encapsulate related data in an object that can be treated as a single unit. We have already seen two types like this, `Point` and `Rectangle`.

Another example, which we will implement ourselves, is `Time`, which represents the time of day. The data encapsulated in a Time object are an hour, a minute, and a number of seconds. Because every `Time` object contains these data, we need fields to hold them.

The first step is to decide what type each variable should be. It seems clear that `Hour` and `Minute` should be integers. Just to keep things interesting, let's make `Second` a `double`.

Fields are declared at the beginning of the class definition, outside of any method definition, like this:

```
public class Time {
    public int Hour, Minute;
    public double Second;
}
```

By itself, this code fragment is a legal class definition. The state diagram for a `Time` object looks like this:
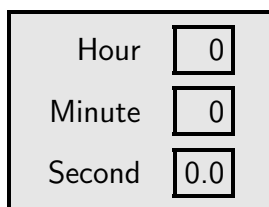
After declaring the fields, the next step is to define a constructor for the new class.

9.3 Constructors

Constructors initialize fields. The syntax for constructors is similar to that of other methods, with three exceptions:

- The name of the constructor is the same as the name of the class.

- Constructors have no return type and no return value.

- The keyword `static` is omitted.

Here is an example for the `Time` class:

```java
public Time() {
    this.Hour   = 0;
    this.Minute = 0;
    this.Second = 0.0;
}
```

Where you would expect to see a return type, between `public` and `Time`, there is nothing. That's how we (and the compiler) can tell that this is a constructor.

This is the **default constructor** as it does not take any arguments. Each line of the constructor initializes an fields to a default value (in this case, midnight). The name `this` is a special keyword that refers to the object we are creating. You can use `this` the same way you use the name of any other object. For example, you can read and write the fields of `this`, and you can pass `this` as an argument to other methods.

But you do not declare `this` and you can't make an assignment to it. `this` is created by the system; all you have to do is initialize its fields.

A common error when writing constructors is to put a `return` statement at the end. Resist the temptation.

9.4 More constructors

Constructors can be overloaded, just like other methods, which means that you can provide multiple constructors with different parameters. The system knows which constructor to invoke by matching the arguments of `new` with the parameters of the constructors.

It is common to have one constructor that takes no arguments (shown above), and one constructor that takes a parameter list identical to the list of fields. For example:

```java
public Time(int hour, int minute, double second) {
    this.Hour   = hour;
    this.Minute = minute;
    this.Second = second;
}
```

The names and types of the parameters are the same as the names and types of the fields. All the constructor does is copy the information from the parameters to the fields. This is known as the **parametric constructor**.

If you look at the documentation for `Point`s and `Rectangle`s, you will see that both classes provide constructors like this. Overloading constructors provides the flexibility to create an object first and then fill in the blanks, or to collect all the information before creating the object.

This might not seem very interesting, and in fact it is not. Writing constructors is a boring, mechanical process. Once you have written two, you will find that you can write them quickly just by looking at the list of fields.

9.5   Creating a new object

Although constructors look like methods, you never invoke them directly. Instead, when you invoke `new`, the system allocates space for the new object and then invokes your constructor.

The following program demonstrates two ways to create and initialize `Time` objects:

```csharp
using System;

namespace ThinkSharp {
   public class Time {
      public int Hour, Minute;
      public double Second;

      // Default constructor
      public Time() {
         this.Hour   = 0;
         this.Minute = 0;
         this.Second = 0.0;
      }

      // Parametric constructor
      public Time(int hour, int minute, double second) {
         this.Hour   = hour;
         this.Minute = minute;
         this.Second = second;
      }

      public static void Main(string[] args) {
         // one way to create and initialize a Time object
         Time t1 = new Time();
         t1.Hour   = 11;
```

```
        t1.Minute = 8;
        t1.Second = 3.14159;
        Console.WriteLine(t1);

        // another way to do the same thing
        Time t2 = new Time(11, 8, 3.14159);
        Console.WriteLine(t2);
    }
  }
}
```

In `Main`, the first time we invoke `new`, we provide no arguments, so the default constructor is invoked. The next few lines assign values to the fields.

The second time we invoke `new`, we provide arguments that match the parameters of the parametric constructor. This way of initializing the fields is more concise and slightly more efficient, but it can be harder to read, since it is not as clear which values are assigned to which fields.

9.6   Writing objects

The output of this program is:

```
ThinkSharp.Time
ThinkSharp.Time
```

When the system outputs the value of a user-defined object type, it prints the name of the type.

To output objects in a way that is more meaningful to users (as opposed to programmers), you can write a method called something like `WriteTime`:

```
  public static void WriteTime(Time t) {
      Console.WriteLine(t.Hour + ":" + t.Minute + ":" + t.Second);
  }
```

Compare this method to the version of `WriteTime` in Section 3.10.

The output of this method, if we pass either `t1` or `t2` as an argument, is `11:8:3.14159`. Although this is recognizable as a time, it is not quite in the standard format. For example, if the number of minutes or seconds is less than 10, we expect a leading `0` as a place-keeper. Also, we might want to drop the decimal part of the seconds. In other words, we want something like `11:08:03`.

In most languages, there are simple ways to control the output format for numbers. Microsoft provide the `String.Format` method for just such an occasion. `WriteTime` could be rewritten:

```
public static void WriteTime(Time t) {
    Console.WriteLine(String.Format("{0:D2}:{1:D2}:{2:D2}",
        t.Hour, t.Minute, (int) t.Second));
}
```

The `String.Format` method is a powerful tool for formatting output like this. Unfortunately is is not very easy to use, so I am going to leave it out of this book. If you are interested, you can look up the documentation on the MSDN[1].

9.7   Operations on objects

In the next few sections, I demonstrate three kinds of methods that operate on objects:

pure function: Takes objects as arguments but does not modify them. The return value is either a primitive or a new object created inside the method.

modifier: Takes objects as arguments and modifies some or all of them. Often returns void.

fill-in method: One of the arguments is an "empty" object that gets filled in by the method. Technically, this is a type of modifier.

Often it is possible to write a given method as a pure function, a modifier, or a fill-in method. I will discuss the pros and cons of each.

9.8   Pure functions

A method is considered a pure function if the result depends only on the arguments, and it has no side effects like modifying an argument or printing something. The only result of invoking a pure function is the return value.

One example is `IsAfter`, which compares two `Time`s and returns a `bool` that indicates whether the first operand comes after the second:

```
public static bool IsAfter(Time time1, Time time2) {
    if ( time1.Hour > time2.Hour )      return true;
    if ( time1.Hour < time2.Hour )      return false;

    if ( time1.Minute > time2.Minute ) return true;
    if ( time1.Minute < time2.Minute ) return false;
```

---

[1]http://msdn.microsoft.com/en-gb/library/system.string.format.aspx

```
            if ( time1.Second > time2.Second ) return true;
            return false;
    }
```

What is the result of this method if the two times are equal? Does that seem like the appropriate result for this method? If you were writing the documentation for this method, would you mention that case specifically?

A second example is `AddTime`, which calculates the sum of two times. For example, if it is 9:14:30, and your breadmaker takes 3 hours and 35 minutes, you could use `AddTime` to figure out when the bread will be done.

Here is a rough draft of this method that is not quite right:

```
    public static Time AddTime(Time t1, Time t2) {
        Time sum = new Time();
        sum.Hour   = t1.Hour   + t2.Hour;
        sum.Minute = t1.Minute + t2.Minute;
        sum.Second = t1.Second + t2.Second;
        return sum;
    }
```

Although this method returns a `Time` object, it is not a constructor. You should go back and compare the syntax of a method like this with the syntax of a constructor, because it is easy to get confused.

Here is an example of how to use this method. If `currentTime` contains the current time and `breadTime` contains the amount of time it takes for your breadmaker to make bread, then you could use `AddTime` to figure out when the bread will be done.

```
    Time currentTime = new Time(9, 14, 30.0);
    Time breadTime   = new Time(3, 35,  0.0);
    Time doneTime    = AddTime(currentTime, breadTime);
    WriteTime(doneTime);
```

The output of this program is 12:49:30, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?

The problem is that this method does not deal with cases where the number of seconds or minutes adds up to more than 60. In that case, we have to "carry" the extra seconds into the minutes column, or extra minutes into the hours column.

Here's a corrected version of the method.

```
    public static Time AddTime(Time t1, Time t2) {
        Time sum = new Time();
        sum.Hour   = t1.Hour   + t2.Hour;
        sum.Minute = t1.Minute + t2.Minute;
        sum.Second = t1.Second + t2.Second;
```

```
        if ( sum.Second >= 60.0 ) {
            sum.Second -= 60.0;
            sum.Minute += 1;
        }
        if ( sum.Minute >= 60 ) {
            sum.Minute -= 60;
            sum.Hour += 1;
        }
        return sum;
    }
```

Although it's correct, it's starting to get big. Later I suggest much shorter alternative.

This code demonstrates two operators we have not seen before, `+=` and `-=`. These operators provide a concise way to increment and decrement variables. They are similar to `++` and `--`, except the amount of the increment does not have to be 1. The statement `sum.Second -= 60.0;` is equivalent to `sum.Second = sum.Second - 60;`

9.9   Modifiers

As an example of a modifier, consider `Increment`, which adds a given number of seconds to a `Time` object. Again, a rough draft of this method looks like:

```
    public static void Increment(Time time, double secs) {
        time.Second += secs;

        if ( time.Second >= 60.0 ) {
            time.Second -= 60.0;
            time.Minute += 1;
        }
        if ( time.Minute >= 60 ) {
            time.minute -= 60;
            time.Hour += 1;
        }
    }
```

The first line performs the basic operation; the remainder deals with the same cases we saw before.

Is this method correct? What happens if the argument `secs` is much greater than 60? In that case, it is not enough to subtract 60 once; we have to keep doing it until `Second` is below 60. We can do that by replacing the `if` statements with `while` statements:

```
    public static void Increment(Time time, double secs) {
        time.Second += secs;
```

```
    while ( time.Second >= 60.0 ) {
        time.Second -= 60.0;
        time.Minute += 1;
    }
    while ( time.Minute >= 60 ) {
        time.Minute -= 60;
        time.Hour += 1;
    }
}
```

This solution is correct, but not very efficient. Can you think of a solution that does not require iteration?

9.10 Fill-in methods

Instead of creating a new object every time `AddTime` is invoked, we could require the caller to provide an object where `AddTime` stores the result. Compare this to the previous version:

```
public static void AddTimeFill(Time t1, Time t2, Time sum) {
    sum.Hour   = t1.Hour   + t2.Hour;
    sum.Minute = t1.Minute + t2.Minute;
    sum.Second = t1.Second + t2.Second;

    if ( sum.Second >= 60.0 ) {
        sum.Second -= 60.0;
        sum.Minute += 1;
    }
    if ( sum.Minute >= 60 ) {
        sum.Minute -= 60;
        sum.Hour += 1;
    }
}
```

The result is stored in `sum`, so the return type is `void`.

Modifiers and fill-in methods are efficient because they don't have to create new objects. But they make it more difficult to isolate parts of a program; in large projects they can cause errors that are hard to find.

Pure functions help manage the complexity of large projects, in part by making certain kinds of errors impossible. Also, they lend themselves to certain kids of composition and nesting. And because the result of a pure function depends only on the parameters, it is possible to speed them up by storing previously-computed results.

I recommend that you write pure functions whenever it is reasonable, and resort to modifiers only if there is a compelling advantage.

9.11   Incremental development and planning

In this chapter I demonstrated a program development process called **rapid prototyping**[2]. For each method, I wrote a rough draft that performed the basic calculation, then tested it on a few cases, correcting flaws as I found them.

This approach can be effective, but it can lead to code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to convince yourself that you have found *all* the errors.

An alternative is to look for insight into the problem that can make the programming easier. In this case the insight is that a `Time` is really a three-digit number in base 60! The `Second` is the "ones column," the `Minute` is the "60's column", and the `Hour` is the "3600's column."

When we wrote `AddTime` and `Increment`, we were effectively doing addition in base 60, which is why we had to "carry" from one column to the next.

Another approach to the whole problem is to convert `Time`s into `double`s and take advantage of the fact that the computer already knows how to do arithmetic with `double`s. Here is a method that converts a `Time` into a `double`:

```
public static double ConvertToSeconds(Time t) {
    int minutes = t.Hour * 60 + t.Minute;
    double seconds = minutes * 60 + t.Second;
    return seconds;
}
```

Now all we need is a way to convert from a `double` to a `Time` object. We could write a method to do it, but it might make more sense to write it as a third constructor:

---

[2]What I am calling rapid prototyping is similar to test-driven development (TDD); the difference is that TDD is usually based on automated testing. See http://en.wikipedia.org/wiki/Test-driven_development.

```
public Time(double secs) {
    this.Hour =(int)(secs / 3600.0);
    secs -= this.Hour * 3600.0;
    this.Minute =(int)(secs / 60.0);
    secs -= this.Minute * 60;
    this.Second = secs;
}
```

This constructor is a little different from the others; it involves some calculation along with assignments to the fields.

You might have to think to convince yourself that the technique I am using to convert from one base to another is correct. But once you're convinced, we can use these methods to rewrite AddTime:

```
public static Time AddTime(Time t1, Time t2) {
    double seconds = ConvertToSeconds(t1) + ConvertToSeconds(t2);
    return new Time(seconds);
}
```

This is shorter than the original version, and it is much easier to demonstrate that it is correct (assuming, as usual, that the methods it invokes are correct). As an exercise, rewrite Increment the same way.

9.12 Generalization

In some ways converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.

But if we have the insight to treat times as base 60 numbers, and make the investment of writing the conversion methods (ConvertToSeconds and the third constructor), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. Imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtraction complete with "borrowing." Using the conversion methods would be much easier.

Ironically, sometimes making a problem harder (more general) makes it easier (fewer special cases, fewer opportunities for error).

9.13 Algorithms

When you write a general solution for a class of problems, as opposed to a specific solution to a single problem, you have written an **algorithm**. This word is not easy to define, so I will try a couple of approaches.

First, consider some things that are not algorithms. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions, so that knowledge is not really algorithmic.

But if you were "lazy," you probably learned a few tricks. For example, to find the product of n and 9, you can write $n - 1$ as the first digit and $10 - n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In my opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence. On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the most difficult to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

Soon you will have the opportunity to design simple algorithms for a variety of problems.

9.14   Glossary

class: Previously, I defined a class as a collection of related methods. In this chapter we learned that a class definition is also a template for a new type of object.

instance: A member of a class. Every object is an instance of some class.

constructor: A special method that initialises the fields of a newly-constructed instance of a class.

default constructor: A constructor that initialises the fields of a newly-constructed instance of a class to the default values.

parametric constructor: A constructor that takes parameters for each of the fields in a class. Initialising the fields of a newly-constructed instance of the class with the given parameters.

startup class: The class that contains the `Main` method where execution of the program begins.

startup project: The project that contains the startup class in the solution.

pure function: A method whose result depends only on its parameters, and that has no side-effects other than returning a value.

modifier: A method that changes one or more of the objects it receives as parameters, and usually returns `void`.

fill-in method: A type of method that takes an "empty" object as a parameter and fills in its fields instead of generating a return value.

algorithm: A set of instructions for solving a class of problems by a mechanical process.

9.15 Exercises

Exercise 9.1. In the board game Scrabble[3], each tile contains a letter, which is used to spell words, and a score, which is used to determine the value of words.

1. Write a definition for a class named `Tile` that represents Scrabble tiles. The fields should be a character named `Letter` and an integer named `Value`.

2. Write a constructor that takes parameters named `letter` and `value` and initializes the fields.

3. Write a method named `WriteTile` that takes a `Tile` object as a parameter and prints the fields in a reader-friendly format.

4. Write a method named `TestTile` that creates a Tile object with the letter `Z` and the value 10, and then uses `WriteTile` to print the state of the object.

The point of this exercise is to practice the mechanical part of creating a new class definition and code that tests it.

Exercise 9.2. Write a class definition for `Date`, an object type that contains three integers, `Year`, `Month` and `Day`. This class should provide two constructors. The first should take no parameters. The second should take parameters named `year`, `month` and `day`, and use them to initialize the fields.

Write a `Main` method that creates a new `Date` object named `birthday`. The new object should contain your birth date. You can use either constructor.

Exercise 9.3. A rational number is a number that can be represented as the ratio of two integers. For example, 2/3 is a rational number, 355/133 is a good approximation of π and you can think of 7 as a rational number with an implicit 1 in the denominator (7/1). For this assignment, you are going to write a class definition for rational numbers.

---

[3]Scrabble is a registered trademark owned by J. W. Spear & Sons Limited of Maidenhead, Berkshire, England, a subsidiary of Mattel Inc., except for the U.S.A and Canada where it is owned by Hasbro Inc.

1. Create a new program called `Rational.cs` that defines a class named `Rational`. A `Rational` object should have two integer fields to store the numerator and denominator.

2. Write a default constructor, one that takes no arguments, and that sets the numerator to 0 and denominator to 1.

3. Write a method called `WriteRational` that takes a Rational object as an argument and outputs it in some reasonable format.

4. Write a `Main` method that creates a new object with type Rational, sets its fields to some values, and writes the object.

5. At this stage, you have a minimal testable program. Test it and, if necessary, debug it.

6. Write a parametric constructor for your class that takes two arguments and that uses them to initialize the fields.

7. Write a method called `Negate` that reverses the sign of a rational number. This method should be a modifier, so it should return `void`. Add lines to `Main` to test the new method.

8. Write a method called `Invert` that inverts the number by swapping the numerator and denominator. Add lines to `Main` to test the new method.

9. Write a method called `ToDouble` that converts the rational number to a double (floating-point number) and returns the result. This method is a pure function; it does not modify the object. As always, test the new method.

10. Write a modifier named `Reduce` that reduces a rational number to its lowest terms by finding the greatest common divisor (GCD) of the numerator and denominator and dividing through. This method should be a pure function; it should not modify the fields of the object on which it is invoked. See Exercise 5.9 for the algorithm to find the GCD.

11. Write a method called `Add` that takes two Rational numbers as arguments and returns a new Rational object. The return object should contain the sum of the arguments.

    There are several ways to add fractions. You can use any one you want, but you should make sure that the result of the operation is reduced so that the numerator and denominator have no common divisor (other than 1).

The purpose of this exercise is to write a class definition that includes a variety of methods, including constructors, modifiers and pure functions.

Chapter 10

Arrays

An **array** is a set of values where each value is identified by an index. You can make an array of `int`s, `double`s, or any other type, but all the values in an array have to have the same type.

Syntactically, array types look like other types except they are followed by `[]`. For example, `int[]` is the type "array of integers" and `double[]` is the type "array of doubles."

You can declare variables with these types in the usual ways:

```
int[] count;
double[] values;
```

Until you initialize these variables, they are set to `null`. To create the array itself, use `new`.

```
count = new int[4];
values = new double[size];
```

The first assignment makes `count` refer to an array of 4 integers; the second makes `values` refer to an array of `double`s. The number of elements in `values` depends on `size`. You can use any integer expression as an array size.

The following figure shows how arrays are represented in state diagrams:
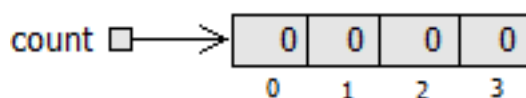
The large numbers inside the boxes are the **elements** of the array. The small numbers outside the boxes are the indices used to identify each box. When you allocate an array if `int`s, the elements are initialized to zero.

10.1   Accessing elements

To store values in the array, use the `[]` operator. For example `count[0]` refers to the "zeroeth" element of the array, and `count[1]` refers to the "oneth" element. You can use the `[]` operator anywhere in an expression:

```
count[0] = 7;
count[1] = count[0] * 2;
count[2]++;
count[3] -= 60;
```

These are all legal assignment statements. Here is the result of this code fragment:
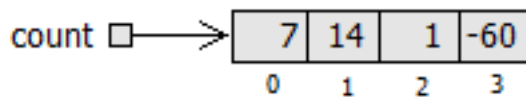
The elements of the array are numbered from 0 to 3, which means that there is no element with the index 4. This should sound familiar, since we saw the same thing with `string` indices. Nevertheless, it is a common error to go beyond the bounds of an array, which throws an `IndexOutOfRangeException`.

You can use any expression as an index, as long as it has type `int`. One of the most common ways to index an array is with a loop variable. For example:

```
int i = 0;
while ( i < 4 ) {
    Console.WriteLine(count[i]);
    i++;
}
```

This is a standard `while` loop that counts from 0 up to 4, and when the loop variable `i` is 4, the condition fails and the loop terminates. Thus, the body of the loop is only executed when `i` is 0, 1, 2 and 3.

Each time through the loop we use `i` as an index into the array, printing the ith element. This type of array traversal is very common.

10.2   Copying arrays

When you copy an array variable, remember that you are copying a reference to the array. For example:

```
double[] a = new double [3];
double[] b = a;
```

This code creates one array of three `double`s, and sets two different variables to refer to it. This situation is a form of aliasing.
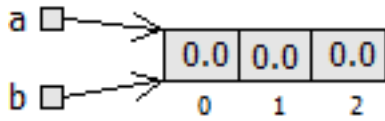
Any changes in either array will be reflected in the other. This is not usually the behaviour you want; more often you want to allocate a new array and copy elements from one to the other.

```java
double[] b = new double[3];

int i = 0;
while ( i < 4 ) {
    b[i] = a[i];
    i++;
}
```

10.3   Arrays and objects

In many ways, arrays behave like objects:

- When you declare an array variable, you get a reference to an array.

- You have to use `new` to create the array itself.

- When you pass an array as an argument, you pass a reference, which means that the invoked method can change the contents of the array.

Some of the objects we looked at, like `Rectangle`s, are similar to arrays in the sense that they are collections of values. This raises the question, "How is an array of 4 integers different from a Rectangle object?"

If you go back to the definition of "array" at the beginning of the chapter, you see one difference: the elements of an array are identified by indices, and the elements of an object have names.

Another difference is that the elements of an array have to be the same type. Objects can have fields with different types.

10.4   `for` loops

The loops we have written have a number of elements in common. All of them start by initializing a variable; they have a test, or condition, that depends on that variable; and inside the loop they do something to that variable, like increment it.

This type of loop is so common that there is another loop statement, called `for`, that expresses it more concisely. The general syntax looks like this:

```
for (⟨initializer⟩; ⟨condition⟩; ⟨incrementor⟩) {
    ⟨body⟩
}
```

This statement is equivalent to

```
⟨initializer⟩;
while ( ⟨condition⟩ ) {
    ⟨body⟩
    ⟨imcrementor⟩
}
```

except that it is more concise and, since it puts all the loop-related statements in one place, it is easier to read. For example:

```
int i = 0;
while ( i < 4 ) {
    Console.WriteLine(count[i]);
    i++;
}
```

can be written

```
for (int i = 0; i < 4; i++) {
    Console.WriteLine(count[i]);
}
```

10.5   `foreach` loops

Indeed traversing a collection of items in this way is so common, there is yet another loop statement, called `foreach`, that does away with the index count. The general syntax looks like this:

```
foreach ( ⟨type⟩ ⟨variable⟩ in ⟨collection⟩ ) {
    ⟨body⟩
}
```

so the loop could be written:

```
foreach ( int item in count ) {
    Console.WriteLine(item);
}
```

This will step though the `count` array one `item` at a time. The only problem is that the `foreach`-loop does not not guarantee the items will be any particular order. Or even in the same order each time. But, when order does not matter, this is a very quick and simple way to traversing a list of items.

10.6   Array length

All arrays have one field named: `Length`. Not surprisingly, it contains the length of the array (number of elements). It is a good idea to use this value as the upper bound of a loop, rather than a constant value. That way, if the size of the array changes, you won't have to go through the program changing all the loops; they will work correctly for any size array.

```
for ( int i = 0; i < a.Length; i++ ) {
    b[i] = a[i];
}
```

The last time the body of the loop gets executed, `i` is `a.Length - 1`, which is the index of the last element. When `i` is equal to `a.Length`, the condition fails and the body is not executed, which is a good thing, since it would throw an exception. This code assumes that the array `b` contains at least as many elements as `a`.

10.7   Random numbers

Most computer programs do the same thing every time they are executed, so they are said to be **deterministic**. Usually, determinism is a good thing, since we expect the same calculation to yield the same result. But for some applications we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly **nondeterministic** turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to generate random numbers and use them to determine the outcome of the program. A library provides a method that generates **pseudorandom** numbers, which may not be truly random, but for our purposes, they will do.

Check out the documentation of the `Random` class. The `NextDouble` method returns a `double` between 0.0 and 1.0. To be precise, it is greater than or equal to 0.0 and strictly less than 1.0. Each time you invoke `NextDouble` you get the next number in a pseudorandom sequence. To see a sample, run this loop:

```
Random random = new Random();
for ( int i = 0; i < 10; i++ ) {
    double x = random.NextDouble();
    Console.WriteLine(x);
}
```

To generate a random number between 0.0 and an upper bound like `high`, you can multiply `x` by `high`.

10.8   Array of random numbers

How would you generate a random integer between `low` and `high`? If your implementation of `RandomInt` is correct, then every value in the range from `low` to `high-1` should have the same probability. If you generate a long series of numbers, every value should appear, at least approximately, the same number of times.

One way to test your method is to generate a large number of random values, store them in an array, and count the number of times each value occurs.

The following method takes a single argument, the size of the array. It allocates a new array of integers, fills it with random values, and returns a reference to the new array.

```
public static int[] RandomArray(int size) {
    int[] array = new int[size];
    for ( int i = 0; i < array.Length; i++ ) {
        array[i] = RandomInt(0, 100);
    }
    return array;
}
```

The return type is `int[]`, which means that this method returns an array of integers. To test this method, it is convenient to have a method that prints the contents of an array.

```
public static void WriteArray(int[] array) {
    for ( int i = 0; i < array.Length; i++ ) {
        Console.WriteLine(array[i]);
    }
}
```

The following code generates an array and writes it:

```
int numValues = 8;
int[] array = RandomArray(numValues);
WriteArray(array);
```

On my machine the output is

```
27
6
54
62
54
2
44
81
```

which is pretty random-looking. Your results should differ.

If these were exam scores (and they would be pretty bad exam scores) the teacher might present the results to the class in the form of a **histogram**, which is a set of counters that keeps track of the number of times each value appears.

For exam scores, we might have ten counters to keep track of how many students scored in the 90s, the 80s, and so on. The next few sections develop code to generate a histogram.

10.9 Counting

A good approach to problems like this is to think of simple methods that are easy to write, then combine them into a solution. This process is called **bottom-up development**[1].

It is not always obvious where to start, but a good approach is to look for subproblems that fit a pattern you have seen before.

In Section 7.7 we saw a loop that traversed a string and counted the number of times a given letter appeared. You can think of this program as an example of a pattern called "traverse and count." The elements of this pattern are:

- A set or container that can be traversed, like an array or a string.

- A test that you can apply to each element in the container.

- A counter that keeps track of how many elements pass the test.

In this case, the container is an array of integers. The test is whether or not a given score falls in a given range of values.

Here is a method called `InRange` that counts the number of elements in an array that fall in a given range. The parameters are the array and two integers that specify the lower and upper bounds of the range.

```
public static int InRange(int[] array, int low, int high) {
    int count = 0;
    foreach ( int item in array ) {
        if ( item >= low && item < high ) {
            count++;
        }
    }
    return count;
}
```

---

[1] http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design

I wasn't specific about whether something equal to `low` or `high` falls in the range, but you can see from the code that `low` is in and `high` is out. That keeps us from counting any elements twice.

Now we can count the number of scores in the ranges we are interested in:

```
int[] scores = RandomArray(30);
int a = InRange(scores, 90, 100);
int b = InRange(scores, 80,  90);
int c = InRange(scores, 70,  80);
int d = InRange(scores, 60,  70);
int f = InRange(scores,  0,  60);
```

10.10   The histogram

This code is repetitious, but it is acceptable as long as the number of ranges is small. But imagine that we want to keep track of the number of times each score appears, all 100 possible values. Would you want to write this?

```
int count0   = InRange(scores,  0,   1);
int count1   = InRange(scores,  1,   2);
int count2   = InRange(scores,  2,   3);
               ⋮
int count100 = InRange(scores, 99, 100);
```

I don't think so. What we really want is a way to store 100 integers, preferably so we can use an index to access each one. HINT: array.

The counting pattern is the same whether we use a single counter or an array of counters. In this case, we initialize the array outside the loop; then, inside the loop, we invoke `InRange` and store the result:

```
int[] counts = new int[100];

for ( int i = 0; i < counts.Length; i++ ) {
   counts[i] = InRange(scores, i, i+1);
}
```

The only tricky thing here is that we are using the loop variable in two roles: as in index into the array, and as the parameter to `InRange`.

10.11   A single-pass solution

This code works, but it is not as efficient as it could be. Every time it invokes `InRange`, it traverses the entire array. As the number of ranges increases, that gets to be a lot of traversals.

It would be better to make a single pass through the array, and for each value, compute which range it falls in. Then we could increment the appropriate counter. In this example, that computation is trivial, because we can use the value itself as an index into the array of counters.

Here is code that traverses an array of scores once and generates a histogram.

```csharp
int[] counts = new int[100];

for ( int i = 0; i < scores.Length; i++ ) {
    int index = scores[i];
    counts[index]++;
}
```

Using a `foreach`-loop would make this even simpler:

```csharp
foreach (int score in scores) {
    counts[score]++;
}
```

10.12   Glossary

array: A collection of values, where all the values have the same type, and each value is identified by an index.

element: One of the values in an array. The `[]` operator selects elements.

index: An integer variable or value used to indicate an element of an array.

deterministic: A program that does the same thing every time it is invoked.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

histogram: An array of integers where each integer counts the number of values that fall into a certain range.

10.13   Exercises

Exercise 10.1. Write a method called `CloneArray` that takes an array of integers as a parameter, creates a new array that is the same size, copies the elements from the first array into the new one, and then returns a reference to the new array.

Exercise 10.2. Write a method called `RandomDouble` that takes two doubles, `low` and `high`, and that returns a random double x so that $low \leq x < high$.

Exercise 10.3. Write a method called `RandomInt` that takes two arguments, `low` and `high`, and that returns a random integer between `low` and `high`, not including `high`.

Exercise 10.4. Encapsulate the code in Section 10.11 in a method called `MakeHist` that takes an array of scores and returns a histogram of the values in the array.

Exercise 10.5. Write a method named `AreFactors` that takes an integer `n` and an array of integers, and that returns `true` if the numbers in the array are all factors of `n` (which is to say that `n` is divisible by all of them). HINT: See Exercise 5.1.

Exercise 10.6. Write a method that takes an array of integers and an integer named `target` as arguments, and that returns the first index where `target` appears in the array, if it does, and -1 otherwise.

Exercise 10.7. Some programmers disagree with the general rule that variables and methods should be given meaningful names. Instead, they think variables and methods should be named after fruit.

For each of the following methods, write one sentence that describes abstractly what the method does. For each variable, identify the role it plays.

```
public static int Banana(int[] a) {
   int grape = 0;
   int i = 0;
   while ( i < a.Length ) {
      grape = grape + a[i];
      i++;
   }
   return grape;
}

public static int Apple(int[] a, int p) {
   int i = 0;
   int pear = 0;
   while ( i < a.Length ) {
      if ( a[i] == p ) pear++;
      i++;
   }
   return pear;
}

public static int Grapefruit(int[] a, int p) {
   for ( int i = 0; i < a.Length; i++) {
      if ( a[i] == p ) return i;
   }
   return -1;
}
```

The purpose of this exercise is to practice reading code and recognizing the computation patterns we have seen.

Exercise 10.8.

1. What is the output of the following program?

2. Draw a stack diagram that shows the state of the program just before `Mus` returns.

3. Describe in a few words what `Mus` does.

```
public static int[] Make(int n) {
    int[] a = new int[n];

    for ( int i = 0; i < n; i++ ) {
        a[i] = i+1;
    }
    return a;
}

public static void Dub(int[] jub) {
    for ( int i = 0; i < jub.Length; i++) {
        jub[i] *= 2;
    }
}

public static int Mus(int[] zoo) {
    int fus = 0;
    foreach (int z in zoo) {
        fus += z;
    }
    return fus;
}

public static void Main(string[] args) {
    int[] bob = Make(5);
    Dub(bob);

    Console.WriteLine(Mus(bob));
}
```

Exercise 10.9. Many of the patterns we have seen for traversing arrays can also be written recursively. It is not common to do so, but it is a useful exercise.

1. Write a method called `MaxInRange` that takes an array of integers and a range of indices (`lowIndex` and `highIndex`), and that finds the maximum value in the array,

considering only the elements between `lowIndex` and `highIndex`, including both ends.

This method should be recursive. If the length of the range is 1, that is, if `lowIndex == highIndex`, we know immediately that the sole element in the range must be the maximum. So that's the base case.

If there is more than one element in the range, we can break the array into two pieces, find the maximum in each of the pieces, and then find the maximum of the maxima.

2. Methods like `MaxInRange` can be awkward to use. To find the largest element in an array, we have to provide a range that includes the entire array.

   ```
   double max = MaxInRange(array, 0, a.Length-1);
   ```

   Write a method called `Max` that takes an array as a parameter and that uses `MaxInRange` to find and return the largest value. Methods like `Max` are sometimes called **wrapper methods** because they provide a layer of abstraction around an awkward method and make it easier to use. The method that actually performs the computation is called the **helper method**.

3. Write a recursive version of `Find` using the wrapper-helper pattern. `Find` should take an array of integers and a target integer. It should return the index of the first location where the target integer appears in the array, or -1 if it does not appear.

Exercise 10.10. One not-very-efficient way to sort the elements of an array is to find the largest element and swap it with the first element, then find the second-largest element and swap it with the second, and so on. This method is called a **selection sort**[2].

1. Write a method called `IndexOfMaxInRange` that takes an array of integers, finds the largest element in the given range, and returns its *index*. You can modify your recursive version of `MaxInRange` or you can write an iterative version from scratch.

2. Write a method called `SwapElement` that takes an array of integers and two indices, and that swaps the elements at the given indices.

3. Write a method called `SelectionSort` that takes an array of integers and that uses `IndexOfMaxInRange` and `SwapElement` to sort the array from largest to smallest.

Exercise 10.11. Write a method called `LetterHist` that takes a string as a parameter and that returns a histogram of the letters in the string. The zeroeth element of the histogram should contain the number of a's in the string (upper and lower case); the 25th element should contain the number of z's. Your solution should only traverse the string once.

---

[2]http://en.wikipedia.org/wiki/Selection_sort

Exercise 10.12. A word is said to be a "doubloon" if every letter that appears in the word appears exactly twice. For example, the following are all the doubloons I found in my dictionary.

> Abba, Anna, appall, appearer, appeases, arraigning, beriberi, bilabial, boob, Caucasus, coco, Dada, deed, Emmett, Hannah, horseshoer, intestines, Isis, mama, Mimi, murmur, noon, Otto, papa, peep, reappear, redder, sees, Shanghaiings, Toto

Write a method called `IsDoubloon` that returns `true` if the given word is a doubloon and `false` otherwise.

Exercise 10.13. Two words are anagrams if they contain the same letters (and the same number of each letter). For example, "stop" is an anagram of "pots" and "allen downey" is an anagram of "well annoyed."

Write a method that takes two strings and returns `true` if the strings are anagrams of each other.

Optional challenge: read the letters of the strings only once.

Exercise 10.14. In Scrabble each player has a set of tiles with letters on them, and the object of the game is to use those letters to spell words. The scoring system is complicated, but longer words are usually worth more than shorter words.

Imagine you are given your set of tiles as a string, like `"quijibo"` and you are given another String to test, like `"jib"`. Write a method called `CanSpell` that takes two strings and returns `true` if the set of tiles can be used to spell the word. You might have more than one tile with the same letter, but you can only use each tile once.

Optional challenge: read the letters of the Strings only once.

Exercise 10.15. In real Scrabble, there are some blank tiles that can be used as wild cards; that is, a blank tile can be used to represent any letter.

Think of an algorithm for `CanSpell` that deals with wild cards. Don't get bogged down in details of implementation like how to represent wild cards. Just describe the algorithm, using English, pseudocode, or code.

Chapter 11

Arrays of Objects

11.1 The Road Ahead

In the next three chapters we will develop programs to work with playing cards and decks of cards. Before we dive in, here is an outline of the steps:

1. In this chapter we'll define a `Card` class and write methods that work with `Card`s and arrays of `Card`s.

2. In Chapter 12 we will create a `Deck` class and write methods that operate on `Deck`s.

3. In Chapter 13 I will present object-oriented programming (OOP) and we will transform the `Card` and `Deck` classes into a more OOP style.

I think that way of proceeding makes the road smoother; the drawback is that we will see several versions of the same code, which can be confusing. If it helps, you can download the code for each chapter as you go along. The code for this chapter is here[1].

11.2 `Card` objects

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense.

There are 52 cards in a deck; each belongs to one of four suits and one of 13 ranks. The suits are Spades, Hearts, Diamonds and Clubs (in descending order in Bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen and King. Depending on what game you are playing, the Ace may be considered higher than King or lower than 2.

---

[1]http://www.rigwit.co.uk/think/sharp/code/Card1.cs

If we want to define a new object to represent a playing card, it is pretty obvious what the fields should be: `Rank` and `Suit`. It is not as obvious what type the fields should be. One possibility is `string`s, containing things like `"Spade"` for suits and `"Queen"` for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. By "encode" I do not mean what some people think, which is to encrypt or translate into a secret code. What a computer scientist means by "encode" is something like "define a mapping between a sequence of numbers and the things I want to represent." For example,

$$
\begin{aligned}
\text{Spades} &\mapsto 3 \\
\text{Hearts} &\mapsto 2 \\
\text{Diamonds} &\mapsto 1 \\
\text{Clubs} &\mapsto 0
\end{aligned}
$$

The obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

$$
\begin{aligned}
\text{Jack} &\mapsto 11 \\
\text{Queen} &\mapsto 12 \\
\text{King} &\mapsto 13
\end{aligned}
$$

The reason I am using mathematical notation for these mappings is that they are not part of the program. They are part of the program design, but they never appear explicitly in the code. The class definition for the `Card` type looks like this:

```java
public class Card {
    public int Suit, Rank;

    public Card() {
        this.Suit = 0;
        this.Rank = 0;
    }

    public Card(int suit, int rank) {
        this.Suit = suit;
        this.Rank = rank;
    }
}
```

As usual, I provide the default and parametric constructors: one takes no parameters; the other takes a parameter for each fields.

To create an object that represents the 3 of Clubs, we invoke `new`:

```java
Card threeOfClubs = new Card(0, 3);
```

The first argument, `0` represents the suit Clubs.

11.3   The `WriteCard` method

When you create a new class, the first step is to declare the fields and write constructors. The second step is to write the standard methods that every object should have, including one that writes the object, and one or two that compare objects. Let's start with `WriteCard`.

To write `Card` objects in a way that humans can read easily, we want to map the integer codes onto words. A natural way to do that is with an array of `string`s. You can create an array of `string`s the same way you create an array of primitive types:

```
string[] suits = new string[4];
```

Then we can set the values of the elements of the array.

```
suits[0] = "Clubs";
suits[1] = "Diamonds";
suits[2] = "Hearts";
suits[3] = "Spades";
```

Creating an array and initializing the elements is such a common operation a special syntax has been provided for it:

```
string[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
```

This statement is equivalent to the separate declaration, allocation, and assignment. The state diagram of this array looks like:
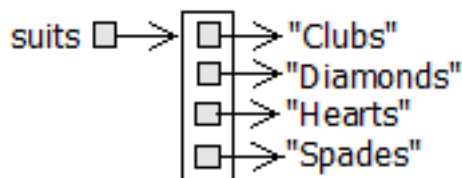
The elements of the array are *references* to the `string`s, rather than `string`s themselves.

Now we need another array of `string`s to decode the ranks:

```
string[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
                   "7", "8", "9", "10", "Jack", "Queen", "King" };
```

The reason for the `"narf"` is to act as a place-keeper for the zeroeth element of the array, which is never used (or shouldn't be). The only valid ranks are 1–13. To avoid this wasted element, we could have started at 0, but the mapping is more natural if we encode 2 as 2, and 3 as 3, etc.

Using these arrays, we can select the appropriate `string`s by using the `Suit` and `Rank` as indices. In the method `WriteCard`,

```
public static void WriteCard(Card c) {
    string[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
    string[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
                       "7", "8", "9", "10", "Jack", "Queen", "King" };

    Console.WriteLine(ranks[c.Rank] + " of " + suits[c.Suit]);
}
```

the expression `suits[c.suit]` means "use the field `suit` from the object `c` as an index into the array named `suits`, and select the appropriate string." The output of this code

```
Card card = new Card(1, 11);
WriteCard(card);
```

is `Jack of Diamonds`.

11.4   The `SameCard` method

The word "same" is one of those things that occur in natural language that seem perfectly clear until you give it some thought, and then you realize there is more to it than you expected.

For example, if I say "Chris and I have the same car," I mean that his car and mine are the same make and model, but they are two different cars. If I say "Chris and I have the same mother," I mean that his mother and mine are one person. So the idea of "sameness" is different depending on the context.

When you talk about objects, there is a similar ambiguity. For example, if two `Card`s are the same, does that mean they contain the same data (rank and suit), or they are actually the same `Card` object?

To see if two references refer to the same object, we use the `==` operator. For example:

```
Card card1 = new Card(1, 11);
Card card2 = card1;

if ( card1 == card2 ) {
    Console.WriteLine("card1 and card2 are identical.");
}
```

References to the same object are **identical**. References to objects with same data are **equivalent**.

To check equivalence, it is common to write a method with a name like `SameCard`.

```
public static bool SameCard(Card c1, Card c2) {
    return (c1.Suit == c2.Suit && c1.Rank == c2.Rank);
}
```

Here is an example that creates two objects with the same data, and uses `SameCard` to see if they are equivalent:

```
Card card1 = new Card(1, 11);
Card card2 = new Card(1, 11);

if ( SameCard(card1, card2) ) {
    Console.WriteLine("card1 and card2 are equivalent.");
}
```

If references are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical. For example, the cards would be equivalent if they are the same card (Jack of Diamonds) from different packs. The cards are equivalent, but not identical. The state diagram looks like this:
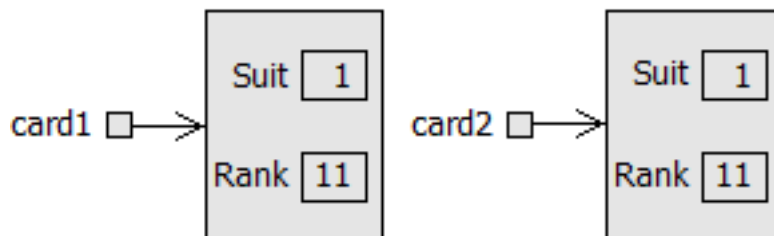
What does it look like when `card1` and `card2` are identical?

11.5   The CompareCard method

For primitive types, the conditional operators compare values and determine when one is greater or less than another. These operators (< and > and the others) don't work for object types. For `string`s there is the `CompareTo` method. For `Card`s we have to write our own, which we will call `CompareCard`. Later, we will use this method to sort a deck of cards.

Some sets are completely ordered, which means that you can compare any two elements and tell which is bigger. Integers and floating-point numbers are totally ordered. Some sets are unordered, which means that there is no meaningful way to say that one element is bigger than another. Fruits are unordered, which is why we cannot compare apples and oranges. The `bool` type is unordered; we cannot say that `true` is greater than `false`.

The set of playing cards is partially ordered, which means that sometimes we can compare cards and sometimes not. For example, I know that the 3 of Clubs is higher than the 2 of Clubs, and the 2 of Diamonds is higher than the 3 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

To make cards comparable, we have to decide which is more important, rank or suit. The choice is arbitrary, but when you buy a new deck of cards, it comes sorted with all the

Clubs together, followed by all the Diamonds, and so on. So let's say that suit is more important.

With that decided, we can write `CompareCard`. It takes two `Card`s as parameters and returns 1 if the first card wins, -1 if the second card wins, and 0 if they are equivalent.

First we compare suits:

```
if ( c1.Suit > c2.Suit ) return 1;
if ( c1.Suit < c2.Suit ) return -1;
```

If neither statement is true, the suits must be equal, and we have to compare ranks:

```
if ( c1.Rank > c2.Rank ) return 1;
if ( c1.Rank < c2.Rank ) return -1;
```

If neither of these is true, the ranks must be equal, so we return `0`.

11.6   Arrays of cards

By now we have seen several examples of composition (the ability to combine language features in a variety of arrangements). One of the first examples we saw was using a method invocation as part of an expression. Another example is the nested structure of statements: you can put an `if` statement within a `while` loop, or within another `if` statement, and so on.

Having seen this pattern, and having learned about arrays and objects, you should not be surprised to learn that you can make arrays of objects. And you can define objects with arrays as fields; you can make arrays that contain arrays; you can define objects that contain objects, and so on. In the next two chapters we will see examples of these combinations using `Card` objects.

This example creates an array of 52 cards:

```
Card[] cards = new Card[52];
```

Here is the state diagram for this object:

The array contains *references* to objects; it does not contain the `Card` objects themselves. The elements are initialized to `null`. You can access the elements of the array in the usual way:

```
if ( cards[0] == null ) {
    Console.WriteLine("No cards yet!");
}
```

But if you try to access the fields of the non-existent `Card`s, you get a `NullReferenceException`.

```
cards[0].Rank;              // NullReferenceException
```

But that is the correct syntax for accessing the `Rank` of the "zeroeth" card in the deck. This is another example of composition, combining the syntax for accessing an element of an array and a field of an object.

The easiest way to populate the deck with `Card` objects is to write nested for loops (i.e., one loop inside the body of another):

```
int index = 0;
for ( int suit = 0; suit <= 3; suit++ ) {
    for ( int rank = 1; rank <= 13; rank++ ) {
        cards[index] = new Card(suit, rank);
        index++;
    }
}
```

The outer loop enumerates the suits from 0 to 3. For each suit, the inner loop enumerates the ranks from 1 to 13. Since the outer loop runs 4 times, and the inner loop runs 13 times, the body is executed is 52 times.

I used `index` to keep track of where in the deck the next card should go. The following state diagram shows what the deck looks like after the first two cards have been allocated:
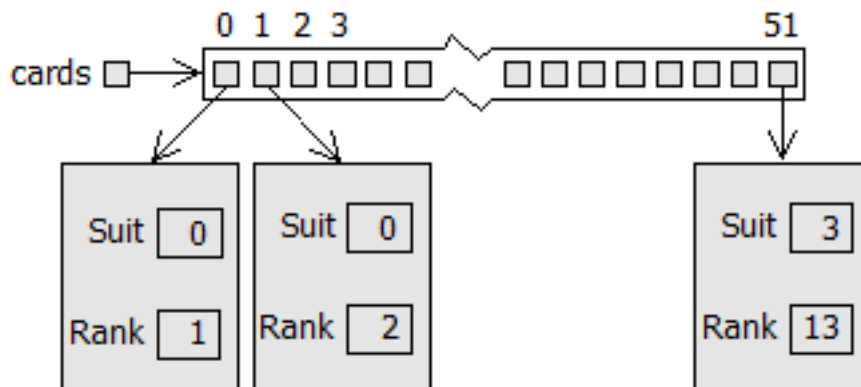
11.7 The `WriteDeck` method

When you work with arrays, it is convenient to have a method that prints the contents. We have seen the pattern for traversing an array several times, so the following method should be familiar:

```
public static void WriteDeck(Card[] cards) {
    foreach ( Card card in cards ) {
```

```
        WriteCard(card);
    }
}
```

Since `cards` has type `Card[]`, an element of `cards` has type `Card`.

11.8   Searching

The next method I'll write is `FindCard`, which searches an array of `Card`s to see whether it contains a certain card. This method gives me a chance to demonstrate two algorithms: **linear search** and **bisection search**.

Linear search is pretty obvious; we traverse the deck and compare each card to the one we are looking for. If we find it we return the index where the card appears. If it is not in the deck, we return -1.

```
public static int FindCard(Card[] cards, Card card) {
    for ( int i = 0; i < cards.Length; i++ ) {
        if ( SameCard(cards[i], card) ) {
            return i;
        }
    }
    return -1;
}
```

The arguments of `FindCard` are `card` and `cards`. It might seem odd to have a variable with the same name as a type (the `card` variable has type `Card`). We can tell the difference because the variable begins with a lower-case letter.

The method returns as soon as it discovers the card, which means that we do not have to traverse the entire deck if we find the card we are looking for. If we get to the end of the loop, we know the card is not in the deck.

If the cards in the deck are not in order, there is no way to search faster than this. We have to look at every card because otherwise we can't be certain the card we want is not there.

But when you look for a word in a dictionary, you don't search linearly through every word, because the words are in alphabetical order. As a result, you probably use an algorithm similar to a bisection search[2]:

1. Start in the middle somewhere.

2. Choose a word on the page and compare it to the word you are looking for.

---

[2]Also known as the Binary Search or Binary Chop

3. If you find the word you are looking for, stop.

4. If the word you are looking for comes after the word on the page, flip to somewhere later in the dictionary and go to step 2.

5. If the word you are looking for comes before the word on the page, flip to somewhere earlier in the dictionary and go to step 2.

If you ever get to the point where there are two adjacent words on the page and your word comes between them, you can conclude that your word is not in the dictionary.

Getting back to the deck of cards, if we know the cards are in order, we can write a faster version of `FindCard`. The best way to write a bisection search is with a recursive method, because bisection is naturally recursive.

The trick is to write a method called `FindBisect` that takes two indices as parameters, `low` and `high`, indicating the segment of the array that should be searched (including both `low` and `high`).

1. To search the array, choose an index between `low` and `high` (call it `mid`) and compare it to the card you are looking for.

2. If you found it, stop.

3. If the card at `mid` is higher than your card, search the range from `low` to `mid - 1`.

4. If the card at `mid` is lower than your card, search the range from `mid + 1` to `high`.

Steps 3 and 4 look suspiciously like recursive invocations. Here's what this looks like translated into program code:

```java
public static int FindBisect(Card[] cards, Card card, int low, int high) {
    // TODO: need a base case
    int mid = (high + low) / 2;
    int comp = CompareCard(cards[mid], card);

    if ( comp == 0 ) {
        return mid;
    } else if ( comp > 0 ) {
        return FindBisect(cards, card, low, mid - 1);
    } else {
        return FindBisect(cards, card, mid + 1, high);
    }
}
```

This code contains the kernel of a bisection search, but it is still missing an important piece, which is why I added a TODO comment. As written, the method recurses forever if the card is not in the deck. We need a base case to handle this condition.

If `high` is less than `low`, there are no cards between them, see we conclude that the card is not in the deck. If we handle that case, the method works correctly:

```
public static int FindBisect(Card[] cards, Card card, int low, int high) {
    Console.WriteLine(low + ", " + high);

    if ( high < low ) return -1;

    int mid = (high + low) / 2;
    int comp = CompareCard(cards[mid], card);

    if ( comp == 0 ) {
        return mid;
    } else if ( comp > 0 ) {
        return FindBisect(cards, card, low, mid-1);
    } else {
        return FindBisect(cards, card, mid+1, high);
    }
}
```

I added a write statement so I can follow the sequence of recursive invocations. I tried out the following code:

```
    Card card1 = new Card(1, 11);
    Console.WriteLine(FindBisect(cards, card1, 0, 51));
```

And got the following output:

```
0, 51
0, 24
13, 24
19, 24
22, 24
23
```

Then I made up a card that is not in the deck (the 15 of Diamonds), and tried to find it. I got the following:

```
0, 51
0, 24
13, 24
13, 17
13, 14
13, 12
-1
```

These tests don't prove that this program is correct. In fact, no amount of testing can prove that a program is correct. On the other hand, by looking at a few cases and examining the code, you might be able to convince yourself.

The number of recursive invocations is typically 6 or 7, so we only invoke `CompareCard` 6 or 7 times, compared to up to 52 times if we did a linear search. In general, bisection is much faster than a linear search, and even more so for large arrays. The bisection search is guaranteed to find a value within $\lceil \log_2 n \rceil$ iterations, as opposed to n for a linear search, where n is the size of the array.

Two common errors in recursive programs are forgetting to include a base case and writing the recursive call so that the base case is never reached. Either error causes infinite recursion, which throws a `StackOverflowException`. (Think of a stack diagram for a recursive method that never ends.)

11.9   Decks and subdecks

Here is the prototype (see Section 7.6) of `FindBisect`:

```
public static int FindBisect(Card[] deck, Card card, int low, int high)
```

We can think of `cards`, `low`, and `high` as a single parameter that specifies a **subdeck**. This way of thinking is common, and is sometimes referred to as an **abstract parameter**. What I mean by "abstract" is something that is not literally part of the program text, but which describes the function of the program at a higher level.

For example, when you invoke a method and pass an array and the bounds `low` and `high`, there is nothing that prevents the invoked method from accessing parts of the array that are out of bounds. So you are not literally sending a subset of the deck; you are really sending the whole deck. But as long as the recipient plays by the rules, it makes sense to think of it abstractly as a subdeck.

This kind of thinking, in which a program takes on meaning beyond what is literally encoded, is an important part of thinking like a computer scientist. The word "abstract" gets used so often and in so many contexts that it comes to lose its meaning. Nevertheless, **abstraction** is a central idea in computer science (and many other fields).

A more general definition of "abstraction" is "The process of modelling a complex system with a simplified description to suppress unnecessary details while capturing relevant behaviour."

11.10   Glossary

encode: To represent one set of values using another set of values, by constructing a mapping between them.

identity: Equality of references. Two references that refer (point) to the same object in memory.

equivalence: Equality of values. Two references that point to objects that contain the same data.

abstract parameter: A set of parameters that act together as a single parameter.

abstraction: The process of interpreting a program (or anything else) at a higher level than what is literally represented by the code.

11.11 Exercises

Exercise 11.1. Encapsulate the code in Section 11.5 in a method. Then modify it so that aces are ranked higher than Kings.

Exercise 11.2. Encapsulate the deck-building code of Section 11.6 in a method called `MakeDeck` that takes no parameters and returns a fully-populated array of `Card`s.

Exercise 11.3. In Blackjack the object of the game is to get a collection of cards with a score of 21. The score for a hand is the sum of scores for all cards. The score for an aces is 1, for all face cards is ten, and for all other cards the score is the same as the rank. Example: the hand (Ace, 10, Jack, 3) has a total score of $1 + 10 + 10 + 3 = 24$.

Write a method called `HandScore` that takes an array of cards as an argument and that returns the total score.

Exercise 11.4. In Poker a "flush" is a hand that contains five or more cards of the same suit. A hand can contain any number of cards.

1. Write a method called `SuitHist` that takes an array of Cards as a parameter and that returns a histogram of the suits in the hand. Your solution should only traverse the array once.

2. Write a method called `HasFlush` that takes an array of Cards as a parameter and that returns `true` if the hand contains a flush, and `false` otherwise.

Chapter 12

Objects of Arrays

WARNING: In this chapter, we take another step toward object-oriented programming, but we are not there yet. So many of the examples are non-idiomatic; that is, while they are valid code, it is not good code. This transitional form will help you learn (I hope), but don't write code like this.

You can download the code in this chapter from here[1].

12.1   The `Deck` class

In the previous chapter, we worked with an array of objects, but I also mentioned that it is possible to have an object that contains an array as a field. In this chapter we create a `Deck` object that contains an array of `Card`s.

The class definition looks like this:

```
public class Deck {
   public Card[] Cards;

   public Deck(int n) {
      this.Cards = new Card[n];
   }
}
```

The constructor initializes the field with an array of cards, but it doesn't create any cards. Here is a state diagram showing what a `Deck` looks like with no cards:

---

[1]http://www.rigwit.co.uk/think/sharp/code/Card2.cs
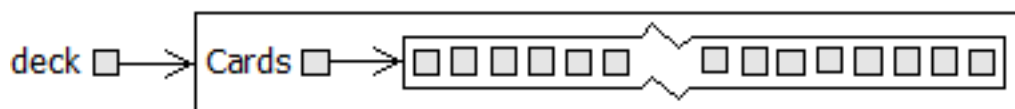
Here is the default constructor that makes a 52-card deck and populates it with `Card`s:

```
public Deck() {
   this.cards = new Card[52];
   int index = 0;
   for ( int suit = 0; suit <= 3; suit++ ) {
      for ( int rank = 1; rank <= 13; rank++ ) {
         cards[index] = new Card(suit, rank);
         index++;
      }
   }
}
```

This method is similar to `MakeDeck` (from Section 11.6); we just changed the syntax to make it a constructor. To invoke it, we use `new`:

```
Deck deck = new Deck();
```

Now it makes sense to put the methods that pertain to `Deck`s in the `Deck` class definition. Looking at the methods we have written so far, one obvious candidate is `WriteDeck` (Section 11.7). Here's how it looks, rewritten to work with a `Deck`:

```
public static void WriteDeck(Deck deck) {
   foreach ( Card card in deck.Cards ) {
      Card.WriteCard(card);
   }
}
```

One change is the type of the parameter, from `Card[]` to `Deck`. Renaming the parameter from `cards` to, the hopefully more meaningful, `deck` in the process.

The second change is that we can no longer use `cards` in the `foreach`, because `deck` is a `Deck` object now, not an array. It contains an array, but it is not an array. So we have to write `deck.Cards` to extract the array from the `Deck` object.

The last change is that the invocation of `WriteCard` has to say explicitly that `WriteCard` is defined in the `Card` class.

12.2   Shuffling

For most card games you need to be able to shuffle the deck; that is, put the cards in a random order. In Section 10.7 we saw how to generate random numbers, but it is not obvious how to use them to shuffle a deck.

One possibility is to model the way humans shuffle, which is usually by dividing the deck in two and then choosing alternately from each deck. Since humans usually don't shuffle perfectly, after about 7 iterations the order of the deck is pretty well randomized. But a

computer program would have the annoying property of doing a perfect shuffle every time, which is not really very random. In fact, after 8 perfect shuffles, you would find the deck back in the order you started in[2].

A better shuffling algorithm is to traverse the deck one card at a time, and at each iteration choose two cards and swap them.

Here is an outline of how this algorithm works. To sketch the program, I am using a combination of program statements and English words that is sometimes called **pseudocode**:

```
for ( int i = 0; i < deck.Cards.Length; i++) {
    // choose a number between i and deck.cards.length-1
    // swap the ith card and the randomly-chosen card
}
```

The nice thing about pseudocode is that it often makes it clear what methods you are going to need. In this case, we need something like `RandomInt`, which chooses a random integer between `low` and `high`, and `SwapCards` which takes two indices and switches the cards at the indicated positions.

This process—writing pseudocode first and then writing methods to make it work—is called **top-down development**[3].

12.3 Sorting

Now that we have messed up the deck, we need a way to put it back in order. There is an algorithm for sorting that is ironically similar to the algorithm for shuffling. It's called **selection sort** because it works by traversing the array repeatedly and selecting the lowest remaining card each time.

During the first iteration we find the lowest card and swap it with the card in the 0th position. During the ith, we find the lowest card to the right of i and swap it with the ith card.

Here is pseudocode for selection sort:

```
for ( int i = 0; i < deck.Cards.Length; i++ ) {
    // find the lowest card at or to the right of i
    // swap the ith card and the lowest card
}
```

Again, the pseudocode helps with the design of the **helper methods**. In this case we can use `SwapCards` again, so we only need one new one, called `IndexLowestCard`, that takes an array of cards and an index where it should start looking.

---

[2]For more information, see http://en.wikipedia.org/wiki/Faro_shuffle
[3]http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design

12.4   Subdecks

How should we represent a hand or some other subset of a full deck? One possibility is to create a new class called `Hand`, which might extend `Deck`. Another possibility, the one I will demonstrate, is to represent a hand with a `Deck` object with fewer than 52 cards.

We might want a method, `Subdeck`, that takes a Deck and a range of indices, and that returns a new Deck that contains the specified subset of the cards:

```
public static Deck Subdeck(Deck deck, int low, int high) {
   Deck sub = new Deck(high - low + 1);

   for ( int i = 0; i < sub.Cards.Length; i++ ) {
      sub.Cards[i] = deck.Cards[low + i];
   }
   return sub;
}
```

The length of the subdeck is `high-low+1` because both the low card and high card are included. This sort of computation can be confusing, and lead to "off-by-one" errors. Drawing a picture is usually the best way to avoid them.

Because we provide an argument with `new`, the parametric constructor gets invoked, which only allocates the array and doesn't allocate any cards. Inside the `for` loop, the subdeck gets populated with copies of the references from the deck.

The following is a state diagram of a subdeck being created with the parameters `low=3` and `high=7`. The result is a hand with 5 cards that are shared with the original deck; i.e. they are aliased.
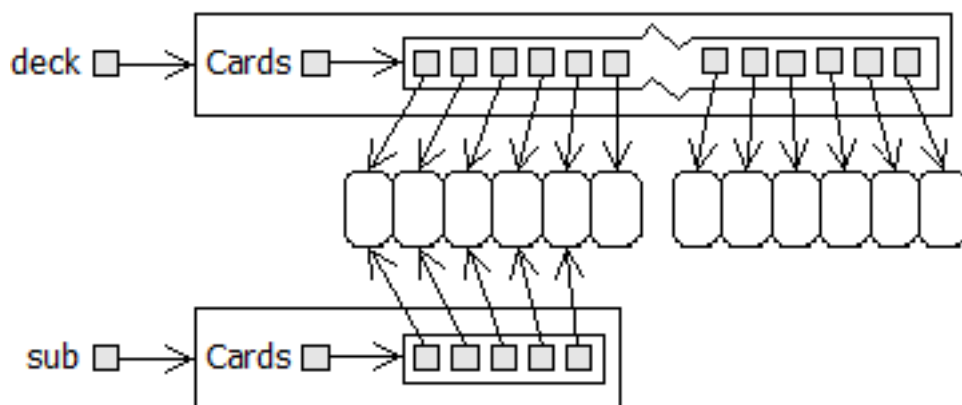
Aliasing is usually not generally a good idea, because changes in one subdeck are reflected in others, which is not the behaviour you would expect from real cards and decks. But if the cards are immutable, aliasing is less dangerous. In this case, there is probably no reason ever to change the rank or suit of a card. Instead we can create each card once and then treat it as an immutable object. So for `Cards` aliasing is a reasonable choice.

12.5   Shuffling and dealing

In Section 12.2 I wrote pseudocode for a shuffling algorithm. Assuming that we have a method called `ShuffleDeck` that takes a deck as an argument and shuffles it, we can use it to deal hands:

```
Deck deck = new Deck();
ShuffleDeck(deck);

Deck hand1 = Subdeck(deck,  0,  4);
Deck hand2 = Subdeck(deck,  5,  9);
Deck pack  = Subdeck(deck, 10, 51);
```

This code puts the first 5 cards in one hand, the next 5 cards in the other, and the rest into the pack.

When you thought about dealing, did you think we should give one card to each player in the round-robin style that is common in real card games? I thought about it, but then realized that it is unnecessary for a computer program. The round-robin convention is intended to mitigate imperfect shuffling and make it more difficult for the dealer to cheat. Neither of these is an issue for a computer.

This example is a useful reminder of one of the dangers of engineering metaphors: sometimes we impose restrictions on computers that are unnecessary, or expect capabilities that are lacking, because we unthinkingly extend a metaphor past its breaking point.

12.6   Merge sort

In Section 12.3, we saw a simple sorting algorithm that turns out not to be very efficient. To sort n items, it has to traverse the array n times, and each traversal takes an amount of time that is proportional to n. The total time, therefore, is proportional to n^2.

In this section I sketch a more efficient algorithm called **merge sort**. To sort n items, a merge sort takes time proportional to $n \log n$. That may not seem impressive, but as n gets big, the difference between n^2 and $n \log n$ can be enormous. Try out a few values of n and see.

The basic idea behind merge sort is this: if you have two subdecks, each of which has been sorted, it is easy (and fast) to merge them into a single, sorted deck. Try this out with a deck of cards:

1. Form two subdecks with about 10 cards each and sort them so that when they are face up the lowest cards are on top. Place both decks face up in front of you.

2. Compare the top card from each deck and choose the lower one. Flip it over and add it to the merged deck.

3. Repeat step two until one of the decks is empty. Then take the remaining cards and add them to the merged deck.

The result should be a single sorted deck. Here's what this looks like in pseudocode:

```
public static Deck Merge(Deck d1, Deck d2) {
    // create a new deck big enough for all the cards
    Deck result = new Deck(d1.Cards.Length + d2.Cards.Length);

    int i = 0;  // used to keep track of where we are in the first deck
    int j = 0;  // used to keep track of where we are in the second deck

    // the index k traverses the result deck
    for ( int k = 0; k < result.Cards.Length; k++ ) {
        // if d1 is empty, d2 wins; if d2 is empty, d1 wins;
        // otherwise, compare the two cards
        // add the winner to the new deck
    }
    return result;
}
```

The best way to test `Merge` is to build and shuffle a deck, use subdeck to form two (small) hands, and then use the selection sort routine from earlier to sort the two halves. Then you can pass the two halves to `Merge` to see if it works.

If you can get that working, try a simple implementation of `MergeSort`:

```
public static Deck MergeSort(Deck deck) {
    // find the midpoint of the deck
    // divide the deck into two subdecks
    // sort the subdecks using sortDeck
    // merge the two halves and return the result
}
```

Then, if you get that working, the real fun begins! The magical thing about merge sort is that it is recursive. At the point where you sort the subdecks, why should you invoke the old, slow version of `Sort`? Why not invoke the spiffy new `MergeSort` you are in the process of writing?

Not only is that a good idea, it is *necessary* to achieve the performance advantage I promised. But to make it work you have to have a base case; otherwise it recurses forever. A simple base case is a subdeck with 0 or 1 cards. If `MergeSort` receives such a small subdeck, it can return it unmodified, since it is already sorted.

The recursive version of `MergeSort` should look something like this:

```
public static Deck MergeSort(Deck deck) {
    // if the deck is 0 or 1 cards, return it

    // find the midpoint of the deck
    // divide the deck into two subdecks
    // sort the subdecks using mergesort
    // merge the two halves and return the result
}
```

As usual, there are two ways to think about recursive programs: you can think through the entire flow of execution, or you can make the "leap of faith" (see Section 5.9). I have constructed this example to encourage you to make the leap of faith.

When you use `SortDeck` to sort the subdecks, you don't feel compelled to follow the flow of execution, right? You just assume it works because you already debugged it. Well, all you did to make `MergeSort` recursive was replace one sorting algorithm with another. There is no reason to read the program differently.

Actually, you have to give some thought to getting the base case right and making sure that you reach it eventually, but other than that, writing the recursive version should be no problem. Good luck!

12.7 Static fields

So far we have seen local variables, which are declared inside a method, and fields, which are declared in a class definition, usually before the method definitions.

Local variables are created when a method is invoked and destroyed when the method ends. Fields are created when you create an object and destroyed when the object is garbage collected.

Now it's time to learn about **static fields**. Like fields, static fields are defined in a class definition before the method definitions, but they are identified by the keyword `static`. They are created when the program starts and survive until the program ends.

You can refer to a static field from anywhere inside the class definition. Static fields are often used to store constant values that are needed in several places.

As an example, here is a version of `Card` where `suits` and `ranks` are static fields:

```csharp
public class Card {
   public int Suit, Rank;

   private static string[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
   private static string[] ranks = {
         "narf", "Ace", "2", "3", "4", "5", "6",
         "7", "8", "9", "10", "Jack", "Queen", "King" };

   public static void WriteCard(Card card) {
      Console.WriteLine(ranks[card.Rank] + " of " + suits[card.Suit]);
   }
}
```

Inside `WriteCard` we can refer to `suits` and `ranks` as if they were local variables.

12.8   Glossary

pseudocode: A way of designing programs by writing rough drafts in a combination of English and program code.

helper method: Often a small method that does not do anything enormously useful by itself, but which helps another, more useful method.

static field: A field declared within a class as `static`; there is always exactly one copy of this field in existence, no matter how many instances of the class there are.

12.9   Exercises

Exercise 12.1. The goal of this exercise is to implement the shuffling and sorting algorithms from this chapter.

1. Download the code from this chapter[4] and import it into your development environment. I have provided outlines for the methods you will write, so the program should compile. But when it runs it prints messages indicating that the empty methods are not working. When you fill them in correctly, the messages should go away.

2. If you did Exercise 10.3, you already wrote `RandomInt`. Otherwise, write it now and add code to test it.

3. Write a method called `SwapCards` that takes a deck (array of cards) and two indices, and that switches the cards at those two locations.

   HINT: it should switch references, not the contents of the objects. This is faster; also, it correctly handles the case where cards are aliased.

4. Write a method called `ShuffleDeck` that uses the algorithm in Section 12.2. You might want to use the `RandomInt` method from Exercise 10.3.

5. Write a method called `IndexLowestCard` that uses the `CompareCard` method to find the lowest card in a given range of the deck (from `low` to `high`, including both).

6. Write a method called `SortDeck` that arranges a deck of cards from lowest to highest.

7. Using the pseudocode in Section 12.6, write the method called `Merge`. Be sure to test it before trying to use it as part of a `MergeSort`.

8. Write the simple version of `MergeSort`, the one that divides the deck in half, uses `SortDeck` to sort the two halves, and uses `Merge` to create a new, fully-sorted deck.

9. Write the fully recursive version of `MergeSort`. Remember that `SortDeck` is a modifier and `MergeSort` is a function, which means that they get invoked differently:

   ```
   SortDeck(deck);                 // modifies existing deck
   deck = MergeSort(deck);         // replaces old deck with new
   ```

---

[4]http://www.rigwit.co.uk/think/sharp/code/Card2.cs

Chapter 13

Object-oriented programming

13.1 Programming languages and styles

There are many programming languages and almost as many programming styles (sometimes called paradigms). The programs we have written so far are **procedural**, because the emphasis has been on specifying computational procedures.

Most C# programs are **object-oriented**, which means that the focus is on objects and their interactions. Here are some of the characteristics of object-oriented programming:

- Objects often represent entities in the real world. In the previous chapter, creating the `Deck` class was a step toward object-oriented programming.

- The majority of methods are instance methods (like the methods you invoke on `string`s) rather than static methods (like the `Math` methods). The methods we have written so far have been static methods. In this chapter we write some instance methods.

- Objects are isolated from each other by limiting the ways they interact, especially by preventing them from accessing instance variables without invoking methods.

- Classes are organized in family trees where new classes extend existing classes, adding new methods and replacing others.

In this chapter I translate the `Card` program from the previous chapter from procedural to object-oriented style. You can download the code from this chapter[1].

---

[1]http://www.rigwit.co.uk/think/sharp/code/Card3.cs

13.2   Instance methods and static methods

There are two types of methods in object-orientated languages, the **static method** (sometimes called **class methods**) and the **instance method** (sometimes called **object methods**). Static methods are identified by the keyword `static` in the first line. Any method that does *not* have the keyword `static` is an instance method.

Although we have not written instance methods, we have invoked some. Whenever you invoke a method "on" an object, it's an instance method. For example, `IndexOf` and the other methods we invoked on `string` objects are all instance methods.

The name "object" refers to a specific instance of a class of objects. Most people use the words "object" and "instance" to mean the same thing. A cause of confusion when learning about objects.

Anything that can be written as a static method can also be written as an instance method, and vice versa. But sometimes it is more natural to use one or the other.

For example, here is `WriteCard` as a static method:

```csharp
public static void WriteCard(Card card) {
    Console.WriteLine(ranks[card.Rank] + " of " + suits[card.Suit]);
}
```

Here it is re-written as an instance method:

```csharp
public void Write() {
    Console.WriteLine(ranks[Rank] + " of " + suits[Suit]);
}
```

Here are the changes:

1. I removed `static`.

2. I changed the name of the method to be more idiomatic.

3. I removed the parameter.

4. Inside an instance method you can refer to fields as if they were local variables, so I changed `card.Rank` to `Rank`, and likewise for `Suit`.

Here's how this method is invoked:

```csharp
Card card = new Card(1, 1);
card.Write();
```

When you invoke a method on an object, we are invoking the method on a particular instance of the class of objects. The specific instance becomes the **current object**, also known as `this`. Inside `Write`, the keyword `this` refers to the card the method was invoked on.

13.3 The `ToString` method

Every object type has a method called `ToString` that returns a string representation of the object. When you print an object using `Write` or `WriteLine`, the object's `ToString` method is invoked.

The default version of `ToString` returns a string that contains the type of the object. When you define a new object type, you can **override** the default behaviour by providing a new method with the behaviour you want.

For example, here is a `ToString` method for `Card`:

```
public override string ToString() {
    return ranks[Rank] + " of " + suits[Suit];
}
```

The `override` tells the compiler that you know you are overriding the default method. The return type is `string`, naturally, and it takes no parameters. You can invoke `ToString` in the usual way:

```
    Card card = new Card(1, 1);
    String s = card.ToString();
```

or you can invoke it indirectly through `WriteLine`:

```
    Console.WriteLine(card);
```

13.4 The `Equals` method

In Section 11.4 we talked about two notions of equality: identity, which means that two variables refer to the same object, and equivalence, which means that they have the same value.

The `==` operator tests identity, but there is no operator that tests equivalence, because what "equivalence" means depends on the type of the objects. Instead, objects provide a method named `Equals` that defines equivalence.

System classes provide `Equals` methods that do the right thing. But for user defined types the default behaviour is the same as identity, which is usually not what you want.

For `Card`s we already have a method that checks equivalence:

```
    public static bool SameCard(Card c1, Card c2) {
        return (c1.Suit == c2.Suit && c1.Rank == c2.Rank);
    }
```

So all we have to do is rewrite is as an instance method:

```
    public override bool Equals(Card c2) {
        return (Suit == c2.Suit && Rank == c2.Rank);
     }
```

Again, I removed `static` and the first parameter, `c1`.

Unfortunately this is not quite enough. The default method can take a parameter of any type, so our replacement must do so as well. We could possibly define another class that that we would like to compare to a `Card`, such as `FaceCard` or `Joker`. So here is the final version:

```
1    public override bool Equals(object obj) {
2        if ( obj is Card ) {
3            Card card = (Card) obj;
4            return (Suit == card.Suite && Rank == card.Rank);
5        }
6        return false;
7    }
```

Here is how it works:

1 The method now takes a parameter of any type (the `object` type).

2 The `is` operator is true if the parameter object (`obj`) is an instance of the `Card` class.

3 A typecase is used to cast the parameter into a `Card` variable.

4 Compares the current (`this`) card with the parameter (`card`).

6 Otherwise the parameter was not a `Card`, so return `false`.

The type cast (line 3) is not strictly necessary, it saves casting the object `obj` into a `Card` every time we use it. Without it the `card.Suit` on line 4 would be `((Card) obj).Suit`. This would be a good source for errors, so I prefer to have the extra line and save myself all that trouble.

Here's how it's invoked:

```
Card card1 = new Card(1, 1);
Card card2 = new Card(1, 1);
bool flag = card1.Equals(card2);
Console.WriteLine(flag);
```

Inside `Equals`, `card1` is the current object and `card2` is the parameter, `obj`.

13.5   The `GetHashCode` method

If you implement your own equivalence method, you are also required to override the `GetHashCode` method. This method is intended to return a numeric value that is used to identify an object. It is normally just the sum of the hash codes of the instance variables.

```csharp
public override int GetHashCode() {
    return Suit.GetHashCode() + Rank.GetHashCode();
}
```

As both `Suit` sand `Rank` are both `int`s, we don't need to call `GetHashCode` as it simply returns the value. In this case just summing the instance variables could lead to some confusion as many cards could have the same hash code. Consider the following:

```csharp
Card card1 = new Card(1, 1);
Card card2 = new Card(0, 2);
Console.WriteLine(card1.GetHashCode() + ": " + card1);
Console.WriteLine(card2.GetHashCode() + ": " + card2);
```

When I run this, I get the following output:

```
2: Ace of Diamonds
2: 2 of Clubs
```

$1 + 1 = 2$ and $0 + 2 = 2$, or in other words, both cards have the same hash code. As the hash code is intended to be unique for each object, that is not a good idea. If we multiply the suit encoding by the number of cards in a suit (13) and then add the cards rank, we should get a unique number between 1 and 52:

```csharp
public override int GetHashCode() {
    return Suit * 13 + Rank;
}
```

Now, when I run the test code, I get the output:

```
14: Ace of Diamonds
2: 2 of Clubs
```

This means we can now consider the cards to be completely ordered. We could use this to simply the `Equals` and `CompareCards` methods.

13.6   Oddities and errors

If you have instance methods and static methods in the same class, it is easy to get confused. A common way to organize a class definition is to put all the constructors at the beginning, followed by all the instance methods and then all the static methods.

You can have an instance method and a static method with the same name, as long as they do not have the same number and types of parameters. As with other kinds of overloading, the compiler decides which version to invoke by looking at the arguments you provide.

Now that we know what the keyword `static` means, you have probably figured out that `Main` is a static method, which means that there is no "current object" when it is invoked.

Since there is no current object in a class method, it is an error to use the keyword `this`. If you try, you get an error message like: "Keyword 'this' is not valid in a static property, static method, or static field initializer."

Both static and instance methods can access fields as if they where local variables, without using the dot notation. Static methods can only access static fields, unless you provide an instance, an object name. If you try, you will get the message "An object reference is required for the non-static field, method, or property..."

Instance methods can access both static and non-static fields. The current object is used for the non-static fields. The same as using `this`.

Methods in other classes can access the static fields but must use the name of the object's type (class) rather than the name of the object (instance).

13.7   Inheritance

The language feature most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of an existing class. Extending the metaphor, the existing class is sometimes called the **parent** class and the new class is called the **child**.

The primary advantage of this feature is that you can add methods and fields without modifying the parent. This is particularly useful for library classes, since you can't modify them even if you want to.

If you remember, when I started down the slippery slope to objects, way back in Chapter 9, I presented a `Time` class. Don't worry if you don't remember, you can find it on pages 153–154, but in this version I have converted the static methods into instance methods. Take a moment to read through the code and make sure you understand it. Don't worry about the `public` and `protected` just yet, I am about to explain them, see I told you to be patient.

Notice how `adjustForOverflow` starts with a lower case letter. This is because it is not a `public` method. There is a convention that all `public` definitions start with an upper case letter and all other definitions start with a lower case letter.

Declaring a definition (class, method or field) `private` means that only methods in the same class are allowed to access the definition.   Declaring a definition `public`, as we have done up until now, means that any method in any other class can use the definition.

In addition to `public` and `private` there is a third level of definition, the `protected` definition. This means that only other methods, in the same class, or children of the class, can use it. The method is not available to classes outside of the family. It is half-way between `public`, child classes can access it, and `private`, no other classes can access it.

```
public class Time {
   public int Hour, Minute;
   public double Second;

   public Time () {
      this.Hour   = 0;
      this.Minute = 0;
      this.Second = 0.0;
   }

   public Time (int hour, int minute, double second) {
      this.Hour   = hour;
      this.Minute = minute;
      this.Second = second;
   }

   public Time (double seconds) {
      Hour = (int) (seconds / 3600.0);
      seconds -= Hour * 3600.0;

      Minute = (int) (seconds / 60.0);
      Second = seconds - (Minute * 60.0);
   }

   public double ToSeconds() {
      int minutes = Hour * 60 + Minute;
      return minutes * 60 + Seconds;
   }

   public overload string ToString() {
      return String.Format("{0:D2}:{1:D2}:{2:D2}",
            Hour, Minute, (int) Second);
   }

   public bool IsAfter(Time time) {
      return ToSecionds() > time.ToSeconds();
   }

   public void AddTime(Time time) {
      Hour   += time.Hour;
      Minute += time.Minute;
      Second += time.Second;
      adjustForOverflow();
   }
```

```
   public void Increment(double seconds) {
      Seconds += seconds;
      adjustForOverflow();
   }

   private void adjustForOverflow() {
      while ( Second >= 60.0 ) {
         Second -= 60.0;
         Minute += 1;
      }

      while ( Minute > 59 ) {
         Minute -= 60;
         Hour    += 1;
      }
   }
}
```

Now, suppose that I want a class that records both the Date and Time. Rather than rewriting all that lovely code, which has been debugged and, more importantly, tested. I can create a new class that inherits from the `Time` class. Normally, I would call this new class `DateTime`, but there is already a library class with that name, so I am going to call my new class `DateAndTime`:

```
   public class DateAndTime : Time {
   }
```

The colon tells the compiler that the `DateAndTime` class inherits from the `Time` class. The `DateAndTime` class is the **child** or sub-class, while `Time` is the **parent** or super-class. In C# the parent is called the **base** class.

The child class inherits all of the `public` and `protected` definitions from its parent, both static and instance definitions. It does not inherit any of the `private` definitions.

I can add new attributes to the class that are specific to my new class, such as day, month and year:

```
      public int Day, Month, Year;
```

I will also need a couple of constructors.

```
   public DateAndTime() : base () {
      Day   = 0;
      Month = 0;
      Year  = 0;
   }
```

```
public DateAndTime(int day, int month, int year,
                                 int hour, int minute, double second)
    : base(hour, minute, second) {
      day   = day;
      Month = month;
      Year  = year;
}
```

In this case the colon is telling the compiler to invoke the constructor from the parent, or base, class before it preforms the code in the constructor for the child class. When invoking:

```
new DateAndTime();
```

The system will first invoke Time(), the default constructor from the parent class, before going on to execute the code in the DateAndTime() constructor. While

```
new DateAndTime(20, 4, 2014, 12, 0, 0);
```

will cause the system to invoke Time(12, 0, 0) before executing the code in the constructor. The parameters are taken from the parameters for the DateAndTime constructor.

I can now write a version of the Increment method for my new class:

```
public void Increment(double seconds) {
    base.Increment(seconds);
    adjustForOverflow();
}
```

Here, I am using base to invoke the Increment method from the parent class (Time.Increment). I have just defined a new method with the same name for the child class (DateAndTime.Increment). If I don't include the base, the compiler will assume I meant the new method, a recursive definition. To overcome this, I have to tell the compiler to use the version in the base class.

Recall that the Time.adjustForOverflow method was declared as private. Meaning that no other class can access it, including sub-classes like this one. We have to write our own version. Thankfully, it only has to deal with overflow from hours into days, months and years. The Time.Increment method has already processed the overflow between seconds, minutes and hours:

```
private void adjustForOverflow() {
    while ( Hour > 23 ) {
        Hour -= 24;
        Day  += 1;
    }
    // TODO: Day overflows into Month
    // TODO: Month overflows into Year
}
```

Notice how I can access the `Hour` field inherited from the parent class as well as the `Day` field from the child class.

In addition to providing it's own versions of the parents methods (or not), a child class can provide it's own methods, unique to the child.

There are several advantages to using inheritance in this way:

1. I have built on the `Time` class to provide a new function, the `DateAndTime` class.

2. I have not changed anything in the `Time` class.

3. I could even change the way the `Time` class works. So long as I do not change the `public` interface, the names of the methods or fields, and `public` methods keep the same parameters. I only need to re-test the class works as expected. I don't need to re-test every program that uses the class.

13.8   The class hierarchy

All classes extend some other class. The most basic class is called `Object`. It contains no fields, but it provides the methods `Equals` and `ToString`, among others.

All classes extend `Object`, including all of the classes we have written and all of the library classes, like `Rectangle`. Any class that does not explicitly name a parent inherits from `Object` by default.

Some inheritance chains are much longer, though. For example, the `Button` class from the Windows Presentation Framework, `Windows.Controls.Button` extends `Windows.Controls.Primitives.ButtonBase`, which extends `Windows.Controls.ContentControl` and so on for another 7 classes, until it finally extends the `Object` class. No matter how long the chain, `Object` is the common ancestor of all classes.

The "family tree" of classes is called the class hierarchy. `Object` usually appears at the top, with all the "child" classes below. If you look at the documentation of `Windows.Controls.Button`, for example, you see the part of the hierarchy that makes up `Button`'s pedigree. You will also see that `Button` is extended by another 4 classes.

13.9   Object-oriented design

Inheritance is a powerful feature. Some programs that would be complicated without it can be written concisely and simply with it. Also, inheritance can facilitate code reuse, since you can customize the behaviour of existing classes without having to modify them.

On the other hand, inheritance can make programs hard to read. When you see a method invocation, it can be hard to figure out which method gets invoked.

Also, many of the things that can be done with inheritance can be done as well or better without it. A common alternative is **composition**, where new objects are composed of existing objects, adding new capability without inheritance.

Designing objects and the relationships among them is the topic of **object-oriented design**, which is beyond the scope of this book. But if you are interested, I recommend *Head First Design Patterns*, published by O'Reilly Media.

13.10 Glossary

instance method: A method that is invoked on an object, and that operates on that object. Instance methods do not have the keyword `static`.

static method: A method with the keyword `static`. Static methods are not invoked on objects and they do not have a current object (`this`).

current object: The object on which an instance method is invoked. Inside the method, the current object is referred to by `this`.

implicit: Anything that is left unsaid or implied. Within an instance method, you can refer to the fields implicitly (i.e., without naming the object).

explicit: Anything that is spelled out completely. Within a static method, all references to the non-static fields have to be explicit.

public definition: A method or field that can be accessed by a method in any class.

private definition: A definition (method or field) that can only be accessed by methods in the current class. No other classes (including sub-classes) can access the definition.

protected definition: A definition that can only be accessed by methods in the current class and in sub-classes, but not by methods in any other class.

13.11 Exercises

Exercise 13.1. Download the solutions to the exercises from the previous chapter[2]. It uses only static methods (except the constructors). There is a version of the code from this chapter, you should also download[3], that contains the same program, but most of the

---

[2]http://www.rigwit.co.uk/think/sharp/code/CardSoln2.cs
[3]http://www.rigwit.co.uk/think/sharp/code/CardSoln3.cs

method are instance methods. I left `Merge` unchanged because I think it is more readable as a static method.

Transform `Merge` into an instance method, and change `MergeSort` accordingly. Which version of `Merge` do you prefer?

Exercise 13.2. Transform the following static method into an instance method.

```
public static double Abs(Complex c) {
    return Math.Sqrt(c.Real * c.Real + c.Imaginary * c.Imaginary);
}
```

NOTE: The `Complex` class can be found in the `System.Numerics` name space, along with the `BigInteger` class (Exercise 8.4).

Exercise 13.3. Transform the following instance method into a static method.

```
public bool Compare(Complex b) {
    return (Real == b.Real && Imaginary == b.Imaginary);
}
```

Exercise 13.4. This exercise is a continuation of Exercise 9.3. The purpose is to practice the syntax of instance methods and get familiar with the relevant error messages.

1. Transform the methods in the `Rational` class from static methods to instance methods, and make the necessary changes in `Main`.

2. Make a few mistakes. Try invoking static methods as if they were instance methods and vice-versa. Try to get a sense for what is legal and what is not, and for the error messages that you get when you mess up.

3. Think about the pros and cons of static and instance methods. Which is more concise (usually)? Which is a more natural way to express computation (or, maybe more fairly, what kind of computations can be expressed most naturally using each style)?

Exercise 13.5. The goal of this exercise is to write a program that generates random poker hands[4] and classifies them, so that we can estimate the probability of the various poker hands.

1. Start with `CardSoln3.cs`[5] and make sure you can compile and run it.

2. Write a definition for a class named `PokerHand` that extends `Deck`.

3. Write a `Deck` method named `Deal` that creates a `PokerHand`, transfers cards from the deck to the hand, and returns the hand.

---

[4]http://en.wikipedia.org/wiki/List_of_poker_hands
[5]http://www.rigwit.co.uk/think/sharp/code/CardSoln3.cs

4. In `Main` use `Shuffle` and `Deal` to generate and output four `PokerHands` with five cards each. Did you get anything good?

5. Write a `PokerHand` method called `HasFlush` that returns a boolean indicating whether the hand contains a flush (five chards of the same suit).

6. Write a method called `HasThreeKind` that indicates whether the hand contains Three of a Kind.

7. Write a loop that generates a few thousand hands and checks whether they contain a flush or three of a kind. Estimate the probability of getting one of those hands. Compare your results to the mathematical probability of a flush (0.196%) or of a three of a kind (2.11%).

8. Write methods that test for the other poker hands. Some are easier than others. You might find it useful to write some general-purpose helper methods that can be used for more than one test.

9. In some poker games players get seven cards each and they form a hand with the best five of the seven. Modify your program to generate seven-card hands and recompute the probabilities.

Appendix A

Compiling Hello World

A.1 Visual Studio

Visual Studio is Microsoft's integrated development environment (IDE) for working with C#. They provide a free version called "Visual Studio Express" that you can download from the visual studio website[1]. I recommend the "Express for Windows Desktop" version for a basic introduction.

A.1.1 Solution

To compile a program (such as the `Hello` example on page 7), you will first need to create a new project. But first we are going to need a new solution.

Visual studio collects project into solutions. I suggest you only use one solution, "ThinkSharp," with each program being a separate project within the solution. To create a new solution we have to create a project. Go to the File menu and select the New item, this will open a sub-menu where you can select the Project... item.

This will open the "New Project" dialogue (see figure A.1). In the left hand panel, you should select the Installed item. Then open the Templates item, finally, selecting the Visual C# item. A list of available project templates is given in the centre panel. You should select the Console Application template from this list. Note, do not double-click, but only select the template (a single-click).

You are now ready to enter the name of the project and solution. At the bottom of the dialogue, there are three lines. The first line, Name, is the name of the project. I am going to call this project "`Hello`". Notice how it has automatically replaced the last line (Solution name) with "Hello." I am going to replace this with "`ThinkSharp`". The final dialogue should look like figure A.1.

---

[1]http://www.visualstudio.com/en-gb/products/visual-studio-express-vs.aspx
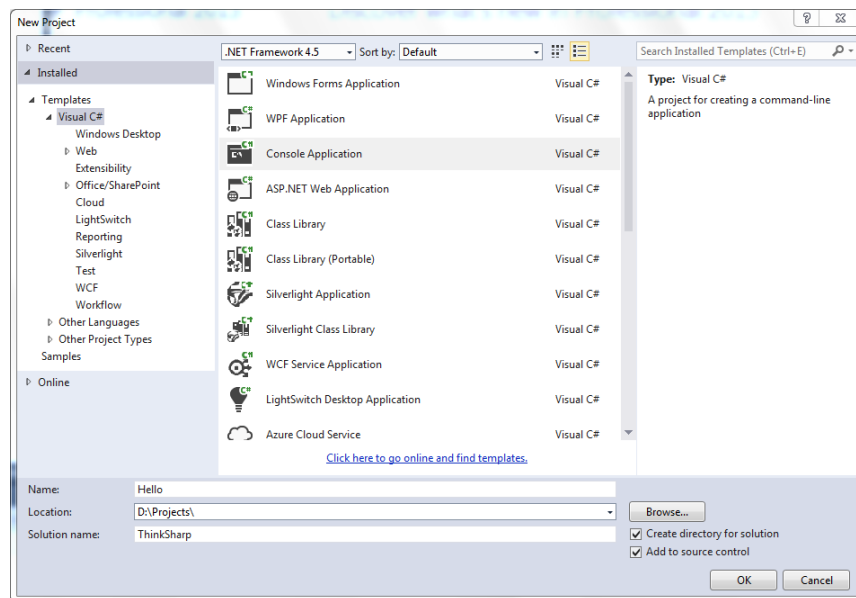
Figure A.1: Visual Studio's "New Project" dialogue

Click the OK button. Depending on the version and configuration of your environment you may be presented with a "Choose Source Control" dialogue. Source control is beyond this book, so if you are presented with the dialogue, simply click on the Cancel button.

We now have a project called Hello in the ThinkSharp solution. The screen will be different for each user depending on which version of visual studio you are using and your personal settings. The editor window is now open with a file called Program.cs. This is the default program for any project.

A.1.2   Projects

To add new projects to this solution, you need to find the solution in the "Solution Explorer" view. Right-click on the solution name and select the Add item from the pop-up menu, then the New Project... item from the sub-menu. This will bring up the "Add New Project" dialogue, which looks exactly the same as the "New Project" dialogue except the Solution name line is missing from the bottom.

A solution will have a main or start-up project which it runs when you run the solution. You need to select the new project as the start-up project for the solution. Find the new project in the "Solution Explorer" view. Right-click to get the pup-up menu and select the Set As StartUp Project item to set this project as the start-up project. Notice how the project name is now bold.

A.1.3 Compiling the `Hello` program

Replace the default `Program.cs` with the program on page 7. Once you have written the text, we can compile the project by selecting the Build Hello item from the Build menu. Any errors are displayed in the "Error List" view. Actually, Visual Studio attempts to compile the program, updating the error list, as you type. I find it useful to have the error list view open when I am writing a program. So I can see the errors before I compile it.

A.1.4 Running the program

There are two ways to run the program, both on the Debug menu. You can simply run the program using the Start Without Debugging item, or you can debug the program using the Start Debugging item. The debug mode allows you to single-step the program and look at the values in the program's variables. Simply pressing the F5 key will cause the program to be compiled before it is run.

A.2 MonoDevelop / Xamarin Studio

MonoDevelop is an open source integrated development environment (IDE) for C# developed by Xamarin to work with Linux, Mac (OS X) and Windows. It has been designed to provide an IDE that allows you to develop code that will run on Linux, Mac, Android, iPhone and Windows Mobile devices. You can download it from the MonoDevelop website[2]. Note that it downloads and installs as Xamarin Studio.

A.2.1 Solution

To compile a program (such as the `Hello` example on page 7), you will first need to create a new project. But first we are going to need a new solution.

Like Visual Studio, Xamarin Studio collects project into solutions. I suggest you only use one solution, "ThinkSharp," with each program being a separate project within the solution. Go to the File menu and select the New item, this will open a sub-menu where you can select the Solution... item.

This will open the "New Solution" dialogue (see figure A.2). In the left hand panel, you should select the C# templates. A list of available project templates is given in the centre panel. You should now select the Console Project template from this list.

You are now ready to enter the name of the project and solution. At the bottom of the dialogue, there are three lines. The first line, Name, is the name of the project. I am

---

[2]http://monodevelop.com

going to call this project "`Hello`". Notice how it has automatically replaced the last line (Solution name) with "Hello." I am going to replace this with "`ThinkSharp`". The final dialogue should look like figure A.2.
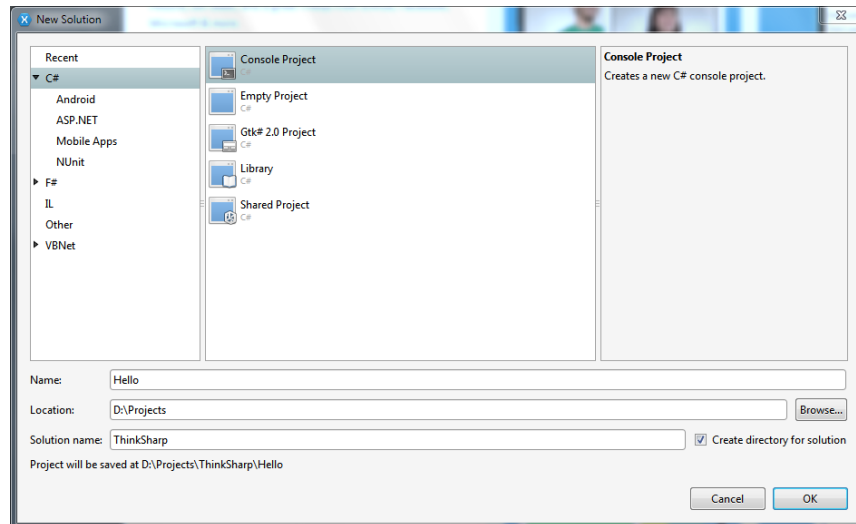
Figure A.2: Xamarin Studio's "New Solution" dialogue

Click the `OK` button. We now have a solution, `ThinkSharp`, with a project called `Hello`. The screen will be different for each user depending on which version of Xamarin studio you are using and your personal settings. The editor window is now open with a file called `Program.cs`. This is the default program for any project.

A.2.2 Projects

To add new projects to this solution, you need to find `ThinkSharp` in the "Solution" view. Right-click on the solution name and select the `Add` item from the pop-up menu, then the `Add New Project...` item from the sub-menu. This will bring up the "New Project" dialogue, which looks exactly the same as the "New Solution" dialogue except the `Solution name` line is missing from the bottom.

A solution will have a main or start-up project which it runs when you run the solution. You need to select the new project as the start-up project for the solution. Find the new project in the "Solution" view. Right-click to get the pup-up menu and select the `Set As Startup Project` item to set this project as the start-up project. Notice how the project name is now bold.

A.2.3 Compiling the `Hello` program

Once we have replaced the default program `Program.cs` with the on on page 7, we can compile the project by selecting the `Build Hello` item from the `Build` menu. Any errors are

displayed in bubbles next to the error.

A.2.4   Running the program

There are two ways to run the program, both on the Run menu. You can simply run the program using the Start Without Debugging item, or you can debug the program using the Start Debugging item.

The debug mode allows you to single-step the program and look at the values in the program's variables (the watch window). To enter debug mode, you must first place a breakpoint in your code, double click in the margin next to the line number where you want to start debugging.

Simply pressing the F5 key will cause the program to be compiled before it is run.

Appendix B

Input and Output

B.1   System objects

The `Console` class provides methods and objects that get input from the keyboard, output text on the screen. The `FileStream` class from the `System.IO` library does the same for file input and output (I/O). `Console` is the object that displays on the screen. When you invoke `Write` and `WriteLine`, you invoke them on `Console`.

For example, when running the following program

```csharp
using System;

namespace ThinkSharp {
   public class Test {
      public static Main() {
         Test test = new Test();
         Console.Write(test);
      }
   }
}
```

The result is:

```
ThinkSharp.Test
```

When the system writes an object, it prints the type of the object (`Test`) and the name space where the type is defined (`ThinkSharp`).

B.2   Keyboard input

`Console` also has methods to get input from the keyboard. There are three methods of interest:

`ReadLine()` This will read a line of text from the keyboard. It will only return when the user presses the Enter or Return keys. The text they have typed is returned in a `string`. Note that the string could be empty if they only pressed the Enter/Return key.

`ReadKey()` This will wait for the user to press a key, but will return as they have pressed it. Unfortunately it does does not return a `char`, but a `ConsoleKeyInfo` object. This object contains information such as whether the SHIFT, CONTROL or ALT keys where pressed at the same time. The `KeyChar` class variable holds the `char` of the key pressed.

Here is a simple version of the `ReadLine` method:

```
public static ReadLine() {
    string line = "";
    ConsoleKeyInfo cki;

    while ( true ) {
        cki = Console.ReadKey();
        if ( cki.Key == ConsoleKey.Enter ) {
            return str;
        }
        line += cki.KeyChar;
    }
}
```

Notice how the character typed is displayed.

`ReadKey(bool)` If you invoke `ReadKey()` with a `false` parameter it will behave just as if you did not provide a parameter. But if you invoke it with a `true`, it will not display the character, but simply return the `ConsoleKeyInfo` object.

B.3 File input

All of the file Input and Output (I/O) operations are provided by the `System.IO` name space. First you must tell the compiler to use the name space:

`using System.IO;`

All file I/O is done using data streams. The next thing you need to do is open a data stream reader as a text reader:

```
TextReader reader = new StreamReader("filename");
```

Finally you can invoke `ReadLine` on `reader`, to take input from the file and convert it to a `string` or `null` if you have reached the end of the file.

Here's a program that reads lines from a file and writes them to the console:

```
using System;
using System.IO;

namespace ThinkSharp {
   public class ListFile {
      public static void Main() {
         processFile("words.txt");
      }

      private static void processFile(string filename) {
         string line;
         int count = 1;

         TextReader reader = new StreamReader(filename);
         while ( true ) {
            line = reader.ReadLine();
            if ( line == null ) {
               break;
            }
            Console.WriteLine(count + ": " + line);
            count++;
         }

         reader.Close();
      }
   }
}
```

The default working directory is the same as the executable file. In the IDEs that is going to be the project's `bin/Debug` sub-directory.

In Visual Studio you can set the working directory in the Debug panel of the project's properties.

B.4   Catching exceptions

In the previous example, `processFile` can throw a `FileNotFoundException` or an `OutOfMemoryException`. And since `Main` calls `processFile`, we could **catch** the exceptions with a `try-catch` statement. Here's an example:

```
public static void Main() {
   try {
      processFile("words.txt");
   } catch (Exception ex) {
      Console.WriteLine("That didn't work.  Here's why:");
      Console.WriteLine(ex.Message);
      Console.WriteLine(ex.StackTrace);
   }
}
```

The structure is similar to an `if` statement. If the first "branch" runs without causing an Exception, the second branch is skipped.

If the first branch causes an Exception, the flow of execution jumps to the second branch, which tries to deal with the exceptional condition (by saying "error" in a polite way). In this case it prints an error message and the stack trace.

Now go do Exercises 7.9, 7.10, and 7.11.

Appendix C

Program development

C.1 Strategies

I present different program development strategies throughout the book, so I wanted to pull them together here. The foundation of all strategies is **incremental development**, which goes like this:

1. Start with a working program that does something visible, like writing something to the console.

2. Add a small number of lines of code at a time, and test the program after every change.

3. Repeat until the program does what it is supposed to do.

After every change, the program should produce some visible effect that tests the new code. This approach to programming can save a lot of time.

Because you only add a few lines of code at a time, it is easy to find syntax errors. And because each version of the program produces a visible result, you are constantly testing your mental model of how the program works. If your mental model is wrong, you are confronted with the conflict (and have a chance to correct it) before you write a lot of bad code.

The challenge of incremental development is that is it not easy to figure out a path from the starting place to a complete and correct program. To help with that, there are several strategies to choose from:

Encapsulation and generalization: If you don't know yet how to divide the computation into methods, start writing code in `main`, then look for coherent chunks to encapsulate in a method, and generalize them appropriately.

Rapid prototyping: If you know what method to write, but not how to write it, start with a rough draft that handles the simplest case, then test it with other cases, extending and correcting as you go.

Bottom-up: Start by writing simple methods, then assemble them into a solution.

Top-down: Use pseudocode to design the structure of the computation and identify the methods you'll need. Then write the methods and replace the pseudocode with real code.

Along the way, you might need some scaffolding. For example, each class should have a `ToString` method that lets you write the state of an object in human-readable form. This method is useful for debugging, but usually not part of a finished program.

C.2 Failure modes

If you are spending a lot of time debugging, it is probably because you are using an ineffective development strategy. Here are the failure modes I see most often (and occasionally fall into):

Non-incremental development: If you write more than a few lines of code without compiling and testing, you are asking for trouble. One time when I asked a student how the homework was coming along, he said, "Great! I have it all written. Now I just have to debug it."

Attachment to bad code: If you write more than a few lines of code without compiling and testing, you may not be able to debug it. Ever. Sometimes the only strategy is (gasp!) to delete the bad code and start over (using an incremental strategy). But beginners are often emotionally attached to their code, even if it doesn't work. The only way out of this trap is to be ruthless.

Random-walk programming: I sometimes work with students who seem to be programming at random. They make a change, run the program, get an error, make a change, run the program and so on. The problem is that there is no apparent connection between the outcome of the program and the change. If you get an error message, take the time to read it. More generally, take time to think.

Compiler submission: Error messages are useful, but they are not always right. For example, if the message says, "Semi-colon expected on line 13," that means there is a syntax error near line 13. But putting a semi-colon on line 13 is not always the solution. Don't submit to the will of the compiler.

The next chapter makes more suggestions for effective debugging.

Appendix D

Debugging

The best debugging strategy depends on what kind of error you have:

- Syntax errors are produced by the compiler and indicate that there is something wrong with the syntax of the program. Example: omitting the semi-colon at the end of a statement.

- Exceptions are produced if something goes wrong while the program is running. Example: an infinite recursion eventually causes a `StackOverflowException`.

- Logic errors cause the program to do the wrong thing. Example: an expression may not be evaluated in the order you expect, yielding an unexpected result.

The following sections are organized by error type; some techniques are useful for more than one type.

D.1   Syntax errors

The best kind of debugging is the kind you don't have to do because you avoid making errors in the first place. In the previous section, I suggested development strategies that minimize errors and makes it easy to find them when you do. The key is to start with a working program and add small amounts of code at a time. When there is an error, you will have a pretty good idea where it is.

Nevertheless, you might find yourself in one of the following situations. For each situation, I make some suggestions about how to proceed.

The compiler is spewing error messages.

If the compiler reports 100 error messages, that doesn't mean there are 100 errors in your program. When the compiler encounters an error, it often gets thrown off track for a while. It tries to recover and pick up again after the first error, but sometimes it reports spurious errors.

Only the first error message is truly reliable. I suggest that you only fix one error at a time, and then recompile the program. You may find that one semi-colon "fixes" 100 errors.

I'm getting a weird compiler message and it won't go away.

First of all, read the error message carefully. It is written in terse jargon, but often there is a carefully hidden kernel of information.

If nothing else, the message will tell you where in the program the problem occurred. Actually, it tells you where the compiler was when it noticed a problem, which is not necessarily where the error is. Use the information the compiler gives you as a guideline, but if you don't see an error where the compiler is pointing, broaden the search.

Generally the error will be prior to the location of the error message, but there are cases where it will be somewhere else entirely. For example, if you get an error message at a method invocation, the actual error may be in the method definition.

If you don't find the error quickly, take a breath and look more broadly at the entire program. Make sure the program is indented properly; that makes it easier to spot syntax errors.

Now, start looking for common errors:

1. Check that all parentheses and brackets are balanced and properly nested. All method definitions should be nested within a class definition. All program statements should be within a method definition.

2. Remember that upper case letters are not the same as lower case letters.

3. Check for semi-colons at the end of statements (and no semi-colons after squiggly-braces).

4. Make sure that any strings in the code have matching quotation marks. Make sure that you use double-quotes for strings and single quotes for characters.

5. For each assignment statement, make sure that the type on the left is the same as the type on the right. Make sure that the expression on the left is a variable name or something else that you can assign a value to (like an element of an array).

6. For each method invocation, make sure that the arguments you provide are in the right order, and have right type, and that the object you are invoking the method on is the right type.

7. If you are invoking a value method, make sure you are doing something with the result. If you are invoking a void method, make sure you are *not* trying to do something with the result.

8. If you are invoking an object method, make sure you are invoking it on an object with the right type. If you are invoking a class method from outside the class where it is defined, make sure you specify the class name.

9. Inside an object method you can refer to the instance variables without specifying an object. If you try that in a class method, you get a message like, "An object reference is required for the non-static field, method, or property '...'."

10. If you use the assignment operator (`=`) instead of the equality operator (`==`) in the condition of an `if`, `while`, or `for` statement, you should get a syntax error along the lines of "Cannot implicitly convert type '...' to 'bool'." Although this is not guaranteed.

If nothing works, move on to the next section...

I can't get my program to compile no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your development environment to make sure the program you are editing is the program the compiler is compiling. If you are not sure, try putting an obvious and deliberate syntax error right at the beginning of the program. Now compile again. If the compiler doesn't find the new error, there is probably something wrong with the way you set up the development environment.

If you have examined the code thoroughly, and you're sure the compiler is compiling the right code, it is time for desperate measures: **debugging by bisection**.

- Make a copy of the file you are working on. If you are working on `Bob.cs`, make a copy called `Bob.cs.old`.

- Delete about half the code from `Bob.cs`. Try compiling again.
  - If the program compiles now, you know the error is in the other half. Bring back about half of the code you deleted and repeat.
  - If the program still doesn't compile, the error must be in this half. Delete about half of the code and repeat.

- Once you have found and fixed the error, start bringing back the code you deleted, a little bit at a time.

This process is ugly, but it goes faster than you might think, and it is very reliable.

I did what the compiler told me to do, but it still doesn't work.

Some compiler messages come with titbits of advice, like " Golfer does not implemented interface member 'System.IComparable.CompareTo(object)'." It sounds like the compiler is telling you to declare Golfer as an abstract class, and if you are reading this book, you probably don't know what that is or how to do it.

Fortunately, the compiler is wrong. The solution in this case is to make sure `Golfer` has a method called `CompareTo` that takes an `object` as a parameter.

Don't let the compiler lead you by the nose. Error messages give you evidence that something is wrong, but the remedies they suggest are unreliable.

D.2   Run-time errors

My program hangs.

If a program stops and seems to be doing nothing, we say it is **hanging**. Often that means that it is caught in an infinite loop or an infinite recursion.

- If there is a particular loop that you suspect is the problem, add a print statement immediately before the loop that says "entering the loop" and another immediately after that says "exiting the loop."

  Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the section titled "Infinite loop."

- Most of the time an infinite recursion will cause the program to run for a while and then produce a StackOverflowException. If that happens, go to the section titled "Infinite recursion."

  If you are not getting a StackOverflowException, but you suspect there is a problem with a recursive method, you can still use the techniques in the infinite recursion section.

- If neither of those suggestions helps, you might not understand the flow of execution in your program. Go to the section titled "Flow of execution."

Infinite loop

If you think you have an infinite loop and you know which loop it is, add a print statement at the end of the loop that prints the values of the variables in the condition, and the value of the condition.

For example,

```
while (x > 0 && y < 0) {
    // do something to x
    // do something to y

    Console.WriteLine("x: " + x);
    Console.WriteLine("y: " + y);
    Console.WriteLine("condition: " + (x > 0 && y < 0));
}
```

Now when you run the program you see three lines of output for each time through the loop. The last time through the loop, the condition should be `false`. If the loop keeps going, you will see the values of `x` and `y` and you might figure out why they are not updated correctly.

Infinite recursion

Most of the time an infinite recursion will cause the program to throw a `StackOverflowException`. But if the program is slow it may take a long time to fill the stack.

If you know which method is causing an infinite recursion, check that there is a base case. There should be some condition that makes the method return without making a recursive invocation. If not, you need to rethink the algorithm and identify a base case.

If there is a base case, but the program doesn't seem to be reaching it, add a print statement at the beginning of the method that prints the parameters. Now when you run the program you see a few lines of output every time the method is invoked, and you see the values of the parameters. If the parameters are not moving toward the base case, you might see why not.

Flow of execution

If you are not sure how the flow of execution is moving through your program, add write statements to the beginning of each method with a message like "entering method foo," where `foo` is the name of the method.

Now when you run the program it writes a trace of each method as it is invoked.

You can also write the arguments each method receives. When you run the program, check whether the values are reasonable, and check for one of the most common errors—providing arguments in the wrong order.

When I run the program I get an Exception.

When an exception occurs, a message that includes the name of the exception, the line of the program where the problem occurred, and a stack trace are output. The stack trace includes the method that was running, the method that invoked it, the method that invoked *that*, and so on.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened.

NullReferenceException: You tried to access an instance variable or invoke a method on an object that is currently `null`. You should figure out which variable is `null` and then figure out how it got to be that way.

Remember that when you declare a variable with an object type, it is initially `null` until you assign a value to it. For example, this code causes a NullReferenceException:

```
Point blank;
Console.WriteLine(blank.X);
```

IndexOutOfRangeException: The index you are using to access an array is either negative or greater than `array.Length-1`. If you can find the site where the problem is, add a write statement immediately before it to write the value of the index and the length of the array. Is the array the right size? Is the index the right value?

Now work your way backwards through the program and see where the array and the index come from. Find the nearest assignment statement and see if it is doing the right thing.

If either one is a parameter, go to the place where the method is invoked and see where the values are coming from. Remember, computers start counting from 0, the "off-by-one" error is very common, even amongst seasoned programmers.

StackOverflowException: See "Infinite recursion."

FileNotFoundException: This means the system didn't find the file it was looking for. You might have to import the file into the project. Otherwise make sure the file exists and that the path is correct. This problem depends on your file system, so it can be hard to track down.

DivideByZeroException: Occurs when something goes wrong during an arithmetic operation and you are attempting to divide a number by 0. Work back though the program and see where the divisor is being set to zero.

I added so many write statements I get inundated with output.

One of the problems with using write statements for debugging is that you can end up buried in output. There are two ways to proceed: either simplify the output or simplify the program.

To simplify the output, you can remove or comment out write statements that aren't helping, or combine them, or format the output so it is easier to understand. As you develop a program, you should write code to generate concise, informative visualizations of what the program is doing.

To simplify the program, scale down the problem the program is working on. For example, if you are sorting an array, sort a *small* array. If the program takes input from the user, give it the simplest input that causes the error.

Also, clean up the code. Remove dead code and reorganize the program to make it easier to read. For example, if you suspect that the error is in a deeply-nested part of the program, rewrite that part with simpler structure. If you suspect a large method, split it into smaller methods and test them separately. As a rule, a method should only contain around seven keywords. This helps you keep methods short and manageable.

The process of finding the minimal test case often leads you to the bug. For example, if you find that a program works when the array has an even number of elements, but not when it has an odd number, that gives you a clue about what is going on.

Reorganizing the program can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

D.3   Logic errors

My program doesn't work.

Logic errors are hard to find because the compiler and the run-time system provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the code and the behaviour you get. You need a hypothesis about what the program is actually doing. Here are some questions to ask yourself:

- Is there something the program was supposed to do, but doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should. See "Flow of execution" above.

- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.

- Is a section of code producing an unexpected effect? Make sure you understand the code, especially if it invokes library methods. Read the documentation for those methods, and try them out with simple test cases. They might not do what you think they do.

To program, you need a mental model what your code does. If it doesn't do what you expect, the problem might not be the program; it might be in your head.

The best way to correct your mental model is to break the program into components (usually the classes and methods) and test them independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Here are some common logic errors to check for:

- Remember that integer division always rounds down. If you want fractions, use codedoubles.

- Floating-point numbers are only approximate, so don't rely on perfect accuracy.

- More generally, use integers for countable things and floating-point numbers for measurable things.

- Arithmetic overflows are not caught. This means that you can get some very unusual results for arithmetic expressions. For example, consider the program:

  ```
  int x = int.MaxValue;
  Console.WriteLine(x);
  x = x + 1;
  Console.WriteLine(x);
  ```

  The first `WriteLine` outputs `2147483647`. You would expect the second `WriteLine` to output `2147483648`, but it does in fact output `-2147483648`. What happened?

  Well, 2,147,483,647 is the largest value that can be stored as an `int`. Adding 1 to this value causes an arithmetic overflow. The system ignores the overflow and only returns the remainder. You can tell the system to check for the arithmetic overflow:

  ```
  x = checked(x + 1);
  ```

  This causes the system to throw an `OverflowException` if there is an arithmetic overflow. But, using `checked` is quite slow, so I suggest only using it if you suspect a calculation could be causing an overflow. Once you have found the cause of the overflow, you should consider removing the `checked`.

- If you use the assignment operator (`=`) instead of the equality operator (`==`) in an expression, you may get some very strange results. While it is syntactically legal it is very unlikely to be correct.

- When you apply the equality operator (==) to an object, it checks identity. If you meant to check equivalence, you should use the Equals method.

- For user defined types, Equals checks identity. If you want a different notion of equivalence, you have to override it.

- Inheritance can lead to subtle logic errors, because you can run inherited code without realizing it. See "Flow of Execution" above.

I've got a big hairy expression and it doesn't do what I expect.

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
diff = (band*rate[n]/9 - (last*rate[n]/9) / (last*rate[n]/9) * 10000;
```

Can be rewritten as

```
double rate = rates[n] / 9;
double target = band * rate;
double current = last * rate;
diff = target - current;
diff = diff / current * 10000;
```

The explicit version is easier to read, because the variable names provide additional documentation, and easier to debug, because you can check the types of the temporary variables and display their values. Using rate twice also makes it easier to read. Although using diff twice might not be so obvious.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, to evaluate $\frac{x}{2\pi}$, you might write

```
double y = x / 2 * Math.PI;
```

That is not correct, because multiplication and division have the same precedence, and they are evaluated from left to right. This expression computes $\frac{x}{2}\pi$.

If you are not sure of the order of operations, use parentheses to make it explicit.

```
double y = x / (2 * Math.PI);
```

This version is correct, and more readable for other people who haven't memorized the order of operations.

My method doesn't return what I expect.

If you have a return statement with a complex expression, you don't have a chance to wrote the value before returning. Again, you can use a temporary variable. For example, instead of

```
public Rectangle Intersection(Rectangle a, Rectangle b) {
    return new Rectangle(
        Math.Min(a.X, b.X),
        Math.Min(a.Y, b.Y),
        Math.Max(a.X + a.Width, b.X + b.Width) - Math.Min(a.X, b.X),
        Math.Max(a.Y + a.Height, b.Y + b.Height) - Math.Min(a.Y, b.Y) );
}
```

You could write

```
public Rectangle Intersection(Rectangle a, Rectangle b) {
    int minX = Math.Min(a.X, b.X);
    int minY = Math.Min(a.Y, b.Y);
    int maxX = Math.Max(a.X + a.Width, b.X + b.Width);
    int maxY = Math.Max(a.Y + a.Height, b.Y + b.Height);
    Rectangle rect = new Rectangle(minX, minY, maxX - minX, maxY - minY);
    return rect;
}
```

Now you have the opportunity to display any of the intermediate variables before returning. And by reusing `minX` and `minY`, you made the code smaller, too.

My write statement isn't doing anything

If you use the `WriteLine` method, the output is displayed immediately, but if you use `Write` (at least in some environments) the output gets stored without being displayed until the next newline. If the program terminates without printing a newline, you may never see the stored output.

If you suspect that this is happening to, change some or all of the `Write` statements to `WriteLine`.

I'm really, really stuck and I need help

First, get away from the computer for a few minutes. Computers emit waves that affect the brain, causing the following symptoms:

- Frustration and rage.

- Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backwards").

- Sour grapes ("this program is lame anyway").

If you suffer from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are possible causes of that behaviour? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. I often find bugs when I let my mind wander. Good places to find bugs are trains, showers, and bed.

No, I really need help.

It happens. Even the best programmers get stuck. Sometimes you need a fresh pair of eyes.

Before you bring someone else in, make sure you have tried the techniques described above. Reduce the problem to a minimum example of your problem. The program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have write statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, give them the information they need.

- What kind of bug is it? Syntax, run-time, or logic?

- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?

- If the bug occurs at compile-time or run-time, what is the error message, and what part of the program does it indicate?

- What have you tried, and what have you learned?

By the time you explain the problem to someone, you might see the answer. This phenomenon is so common that some people recommend a debugging technique called "rubber ducking[1]." Here's how it works:

1. Buy a standard-issue rubber duck.

2. When you are really stuck on a problem, put the rubber duck on the desk in front of you and say, "Rubber duck, I am stuck on a problem. Here's what's happening..."

---

[1]http://en.wikipedia.org/wiki/Rubber_duck_debugging

3. Explain the problem to the rubber duck.

4. See the solution.

5. Thank the rubber duck.

I am not kidding. I can't tell you how many times a student has asked for my help. When explaining the problem, they realise they have just described the cause of the problem. All I did was stand there and listen. Something, any inanimate object (a rubber duck for example) could do just as well.

I found the bug!

When you find the bug, it is usually obvious how to fix it. But not always. Sometimes what seems to be a bug is really an indication that you don't understand the program, or there is an error in your algorithm. In these cases, you might have to rethink the algorithm, or adjust your mental model. Take some time away from the computer to think, work through test cases by hand, or draw diagrams to represent the computation.

After you fix the bug, don't just start in making new errors. Take a minute to think about what kind of bug it was, why you made the error, how the error manifested itself, and what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Index