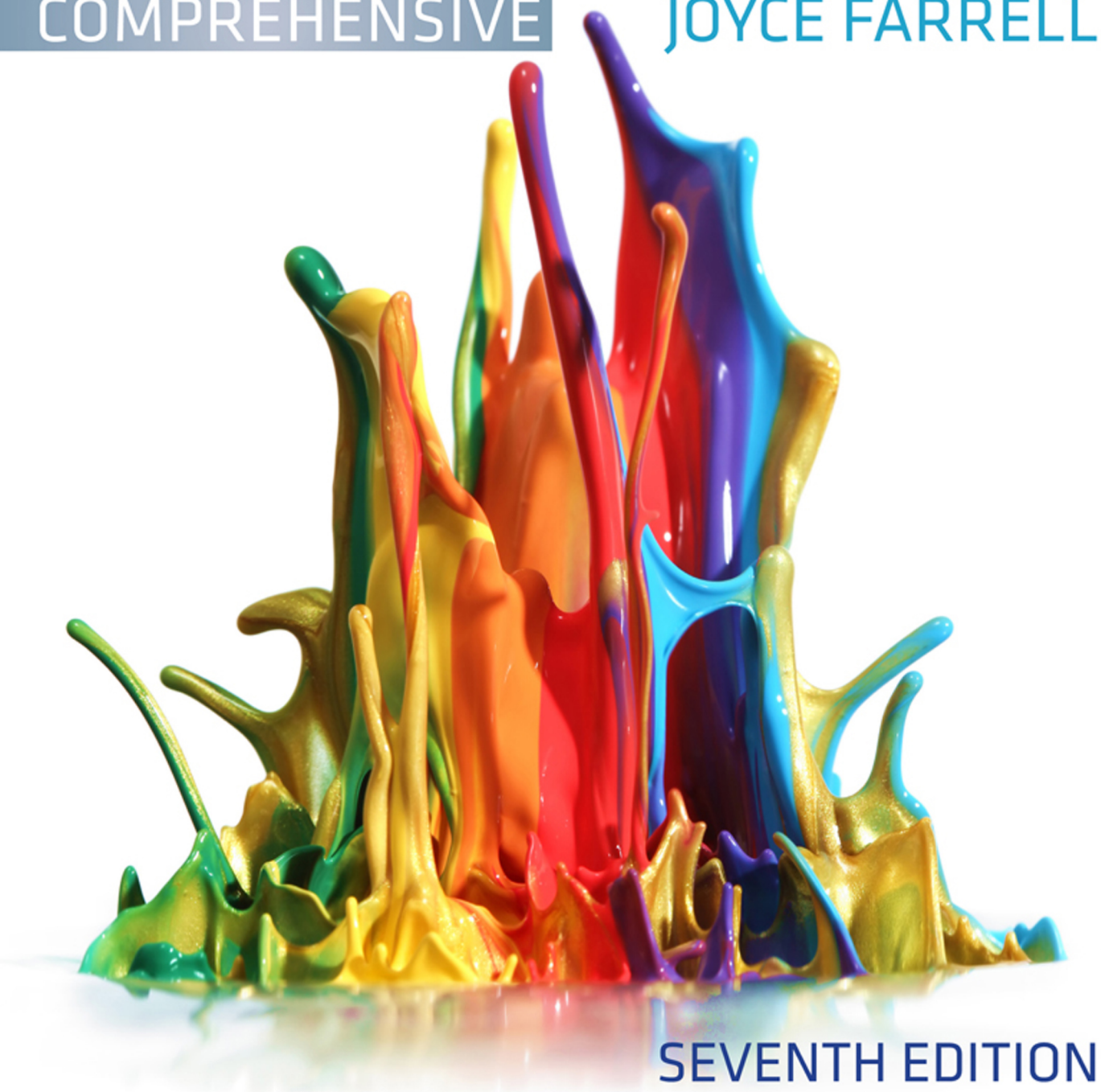
PROGRAMMING LOGIC AND DESIGN

COMPREHENSIVE

JOYCE FARRELL



SEVENTH EDITION

Programming Logic and Design, Comprehensive version, Seventh Edition
Joyce Farrell

Executive Editor: Marie Lee

Acquisitions Editor: Brandi Shailer

Senior Product Manager: Alyssa Pratt

Developmental Editor: Dan Seiter

Senior Content Project Manager: Catherine DiMassa

Associate Product Manager: Stephanie Lorenz

Associate Marketing Manager: Shanna Shelton

Art Director: Faith Brosnan

Text Designer: Shawn Girsberger

Cover Designer: Lisa Kuhn/Curio Press, LLC, www.curiopress.com

Image Credit: © Leigh Prather/Veer

Senior Print Buyer: Julio Esperas

Copy Editor: Michael Beckett

Proofreader: Kim Kosmatka

Indexer: Alexandra Nickerson

Compositor: Integra

Library of Congress Control Number: 2012930593

ISBN-13: 978-1-111-96975-2

Course Technology
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at:
www.cengage.com/global

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

To learn more about Course Technology, visit
www.cengage.com/coursetechnology.

Purchase any of our products at your local college store or at our preferred online store: **www.cengagebrain.com**

Printed in the United States of America
1 2 3 4 5 6 7 16 15 14 13 12

CHAPTER 1

An Overview of Computers and Programming

In this chapter, you will learn about:

- Computer systems
- Simple program logic
- The steps involved in the program development cycle
- Pseudocode statements and flowchart symbols
- Using a sentinel value to end a program
- Programming and user environments
- The evolution of programming models

Understanding Computer Systems

A **computer system** is a combination of all the components required to process and store data using a computer. Every computer system is composed of multiple pieces of hardware and software.

- **Hardware** is the equipment, or the physical devices, associated with a computer. For example, keyboards, mice, speakers, and printers are all hardware. The devices are manufactured differently for large mainframe computers, laptops, and even smaller computers that are embedded into products such as cars and thermostats, but the types of operations performed by different-sized computers are very similar. When you think of a computer, you often think of its physical components first, but for a computer to be useful, it needs more than devices; a computer needs to be given instructions. Just as your stereo equipment does not do much until you provide music, computer hardware needs instructions that control how and when data items are input, how they are processed, and the form in which they are output or stored.
- **Software** is computer instructions that tell the hardware what to do. Software is **programs**, which are instruction sets written by programmers. You can buy prewritten programs that are stored on a disk or that you download from the Web. For example, businesses use word-processing and accounting programs, and casual computer users enjoy programs that play music and games. Alternatively, you can write your own programs. When you write software instructions, you are **programming**. This book focuses on the programming process.

Software can be classified into two broad types:

- **Application software** comprises all the programs you apply to a task, such as word-processing programs, spreadsheets, payroll and inventory programs, and even games.
- **System software** comprises the programs that you use to manage your computer, including operating systems such as Windows, Linux, or UNIX.

This book focuses on the logic used to write application software programs, although many of the concepts apply to both types of software.

Together, computer hardware and software accomplish three major operations in most programs:

- **Input**—Data items enter the computer system and are placed in memory, where they can be processed. Hardware devices that perform input operations include keyboards and mice. **Data items** include all the text, numbers, and other raw material that are entered into and processed by a computer. In business, many of the data items used are facts and figures about such entities as products, customers, and personnel. However, data can also include items such as images, sounds, and a user's mouse movements.
- **Processing**—Processing data items may involve organizing or sorting them, checking them for accuracy, or performing calculations with them. The hardware component that performs these types of tasks is the **central processing unit**, or **CPU**.

- **Output**—After data items have been processed, the resulting information usually is sent to a printer, monitor, or some other output device so people can view, interpret, and use the results. Programming professionals often use the term *data* for input items, but use the term **information** for data that has been processed and output. Sometimes you place output on **storage devices**, such as disks or flash media. People cannot read data directly from these storage devices, but the devices hold information for later retrieval. When you send output to a storage device, sometimes it is used later as input for another program.

You write computer instructions in a computer **programming language** such as Visual Basic, C#, C++, or Java. Just as some people speak English and others speak Japanese, programmers write programs in different languages. Some programmers work exclusively in one language, whereas others know several and use the one that is best suited to the task at hand.

The instructions you write using a programming language are called **program code**; when you write instructions, you are **coding the program**.

Every programming language has rules governing its word usage and punctuation. These rules are called the language's **syntax**. Mistakes in a language's usage are **syntax errors**. If you ask, "How the geet too store do I?" in English, most people can figure out what you probably mean, even though you have not used proper English syntax—you have mixed up the word order, misspelled a word, and used an incorrect word. However, computers are not nearly as smart as most people; in this case, you might as well have asked the computer, "Xpu mxv ort dod nmcad bf B?" Unless the syntax is perfect, the computer cannot interpret the programming language instruction at all.

When you write a program, you usually type its instructions using a keyboard. When you type program instructions, they are stored in **computer memory**, which is a computer's temporary, internal storage. **Random access memory**, or **RAM**, is a form of internal, volatile memory. Programs that are currently running and data items that are currently being used are stored in RAM for quick access. Internal storage is **volatile**—its contents are lost when the computer is turned off or loses power. Usually, you want to be able to retrieve and perhaps modify the stored instructions later, so you also store them on a permanent storage device, such as a disk. Permanent storage devices are **nonvolatile**—that is, their contents are persistent and are retained even when power is lost. If you have had a power loss while working on a computer, but were able to recover your work when power was restored, it's not because the work was still in RAM. Your system has been configured to automatically save your work at regular intervals on a nonvolatile storage device.

After a computer program is typed using programming language statements and stored in memory, it must be translated to **machine language** that represents the millions of on/off circuits within the computer. Your programming language statements are called **source code**, and the translated machine language statements are **object code**.

Each programming language uses a piece of software, called a **compiler** or an **interpreter**, to translate your source code into machine language. Machine language is also called **binary language**, and is represented as a series of 0s and 1s. The compiler or interpreter that translates your code tells you if any programming language component has been used incorrectly. Syntax errors are relatively easy to locate and correct because your compiler or interpreter highlights them. If you write a computer program using a language such as C++

but spell one of its words incorrectly or reverse the proper order of two words, the software lets you know that it found a mistake by displaying an error message as soon as you try to translate the program.

Although there are differences in how compilers and interpreters work, their basic function is the same—to translate your programming statements into code the computer can use. When you use a compiler, an entire program is translated before it can execute; when you use an interpreter, each instruction is translated just prior to execution. Usually, you do not choose which type of translation to use—it depends on the programming language. However, there are some languages for which both compilers and interpreters are available.

After a program's source code is successfully translated to machine language, the computer can carry out the program instructions. When instructions are carried out, a program **runs**, or **executes**. In a typical program, some input will be accepted, some processing will occur, and results will be output.

Besides the popular, comprehensive programming languages such as Java and C++, many programmers use **scripting languages** (also called **scripting programming languages** or **script languages**) such as Python, Lua, Perl, and PHP. Scripts written in these languages usually can be typed directly from a keyboard and are stored as text rather than as binary executable files. Scripting language programs are interpreted line by line each time the program executes, instead of being stored in a compiled (binary) form. Still, with all programming languages, each instruction must be translated to machine language before it can execute.

TWO TRUTHS & A LIE

Understanding Computer Systems

In each Two Truths and a Lie section, two of the numbered statements are true, and one is false. Identify the false statement and explain why it is false.

1. Hardware is the equipment, or the devices, associated with a computer. Software is computer instructions.
2. The grammar rules of a computer programming language are its syntax.
3. You write programs using machine language, and translation software converts the statements to a programming language.

The false statement is #3. You write programs using a programming language such as Visual Basic or Java, and a translation program (called a compiler or interpreter) converts the statements to machine language, which is 0s and 1s.

Understanding Simple Program Logic

A program with syntax errors cannot be fully translated and cannot execute. A program with no syntax errors is translatable and can execute, but it still might contain **logical errors** and produce incorrect output as a result. For a program to work properly, you must develop correct **logic**; that is, you must write program instructions in a specific sequence, you must not leave any instructions out, and you must not add extraneous instructions.

Suppose you instruct someone to make a cake as follows:

```
Get a bowl
Stir
Add two eggs
Add a gallon of gasoline
Bake at 350 degrees for 45 minutes
Add three cups of flour
```

Don't Do It
Don't bake a cake like this!

The dangerous cake-baking instructions are shown with a Don't Do It icon. You will see this icon when the book contains an unrecommended programming practice that is used as an example of what *not* to do.

Even though the cake-baking instructions use English language syntax correctly, the instructions are out of sequence, some are missing, and some instructions belong to procedures other than baking a cake. If you follow these instructions, you will not make an edible cake, and you may end up with a disaster. Many logical errors are more difficult to locate than syntax errors—it is easier for you to determine whether *eggs* is spelled incorrectly in a recipe than it is for you to tell if there are too many eggs or if they are added too soon.

Just as baking directions can be provided in Mandarin, Urdu, or Spanish, program logic can be expressed correctly in any number of programming languages. Because this book is not concerned with a specific language, the programming examples could have been written in Visual Basic, C++, or Java. For convenience, this book uses instructions written in English!

After you learn French, you automatically know, or can easily figure out, many Spanish words. Similarly, after you learn one programming language, it is much easier to understand several other languages.

Most simple computer programs include steps that perform input, processing, and output. Suppose you want to write a computer program to double any number you provide. You can write the program in a programming language such as Visual Basic or Java, but if you were to write it using English-like statements, it would look like this:

```
input myNumber
set myAnswer = myNumber * 2
output myAnswer
```

The number-doubling process includes three instructions:

- The instruction to `input myNumber` is an example of an input operation. When the computer interprets this instruction, it knows to look to an input device to obtain a number. When you work in a specific programming language, you write instructions that tell the computer which device to access for input. For example, when a user enters a number as data for a program, the user might click on the number with a mouse, type it from a keyboard, or speak it into a microphone. Logically, however, it doesn't matter which hardware device is used, as long as the computer knows to accept a number. When the number is retrieved from an input device, it is placed in the computer's memory in a variable named `myNumber`. A **variable** is a named memory location whose value can vary—for example, the value of `myNumber` might be 3 when the program is used for the first time and 45 when it is used the next time. In this book, variable names will not contain embedded spaces; for example, the book will use `myNumber` instead of `my Number`.

From a logical perspective, when you input, process, or output a value, the hardware device is irrelevant. The same is true in your daily life. If you follow the instruction "Get eggs for the cake," it does not really matter if you purchase them from a store or harvest them from your own chickens—you get the eggs either way. There might be different practical considerations to getting the eggs, just as there are for getting data from a large database as opposed to getting data from an inexperienced user working at home on a laptop computer. For now, this book is only concerned with the logic of operations, not the minor details.

- The instruction `set myAnswer = myNumber * 2` is an example of a processing operation. In most programming languages, an asterisk is used to indicate multiplication, so this instruction means "Change the value of the memory location `myAnswer` to equal the value at the memory location `myNumber` times two." Mathematical operations are not the only kind of processing operations, but they are very typical. As with input operations, the type of hardware used for processing is irrelevant—after you write a program, it can be used on computers of different brand names, sizes, and speeds.
- In the number-doubling program, the `output myAnswer` instruction is an example of an output operation. Within a particular program, this statement could cause the output to appear on the monitor (which might be a flat-panel plasma screen or a cathode-ray tube), or the output could go to a printer (which could be laser or ink-jet), or the output could be written to a disk or DVD. The logic of the output process is the same no matter what hardware device you use. When this instruction executes, the value stored in memory at the location named `myAnswer` is sent to an output device. (The output value also remains in computer memory until something else is stored at the same memory location or power is lost.)

Watch the video *A Simple Program*.

Computer memory consists of millions of numbered locations where data can be stored. The memory location of `myNumber` has a specific numeric address, but when you write programs, you seldom need to be concerned with the value of the memory address; instead, you use the easy-to-remember name you created. Computer programmers often refer to memory addresses using hexadecimal notation, or base 16. Using this system, they might use a value like 42FF01A to refer to a memory address. Despite the use of letters, such an address is still a hexadecimal number. Appendix A contains information on this numbering system.

TWO TRUTHS & A LIE

Understanding Simple Program Logic

1. A program with syntax errors can execute but might produce incorrect results.
2. Although the syntax of programming languages differs, the same program logic can be expressed in different languages.
3. Most simple computer programs include steps that perform input, processing, and output.

The false statement is #1. A program with syntax errors cannot execute; a program with no syntax errors can execute, but might produce incorrect results.

Understanding the Program Development Cycle

A programmer's job involves writing instructions (such as those in the doubling program in the preceding section), but a professional programmer usually does not just sit down at a computer keyboard and start typing. Figure 1-1 illustrates the **program development cycle**, which can be broken down into at least seven steps:

1. Understand the problem.
2. Plan the logic.
3. Code the program.
4. Use software (a compiler or interpreter) to translate the program into machine language.
5. Test the program.
6. Put the program into production.
7. Maintain the program.

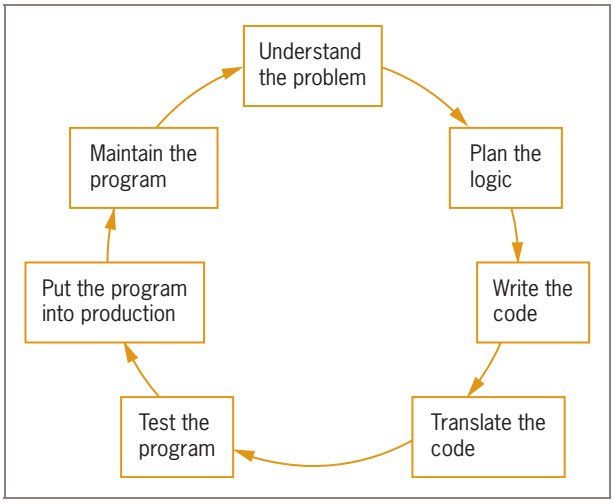


Figure 1-1 The program development cycle

Understanding the Problem

Professional computer programmers write programs to satisfy the needs of others, called **users** or **end users**. Examples of end users include a Human Resources department that needs a printed list of all employees, a Billing department that wants a list of clients who are 30 or more days overdue on their payments, and an Order department that needs a Web site to provide buyers with an online shopping cart. Because programmers are providing a service to these users, programmers must first understand what the users want. When a program runs, you usually think of the logic as a cycle of input-processing-output operations, but when you plan a program, you think of the output first. After you understand what the desired result is, you can plan the input and processing steps to achieve it.

Suppose the director of Human Resources says to a programmer, "Our department needs a list of all employees who have been here over five years, because we want to invite them to a special thank-you dinner." On the surface, this seems like a simple request. An experienced programmer, however, will know that the request is incomplete. For example, you might not know the answers to the following questions about which employees to include:

- Does the director want a list of full-time employees only, or a list of full- and part-time employees together?
- Does she want to include people who have worked for the company on a month-to-month contractual basis over the past five years, or only regular, permanent employees?
- Do the listed employees need to have worked for the organization for five years as of today, as of the date of the dinner, or as of some other cutoff date?
- What about an employee who worked three years, took a two-year leave of absence, and has been back for three years?

The programmer cannot make any of these decisions; the user (in this case, the Human Resources director) must address these questions.

More decisions still might be required. For example:

- What data should be included for each listed employee? Should the list contain both first and last names? Social Security numbers? Phone numbers? Addresses?
- Should the list be in alphabetical order? Employee ID number order? Length-of-service order? Some other order?
- Should the employees be grouped by any criteria, such as department number or years of service?

Several pieces of documentation are often provided to help the programmer understand the problem. **Documentation** consists of all the supporting paperwork for a program; it might include items such as original requests for the program from users, sample output, and descriptions of the data items available for input.

Fully understanding the problem may be one of the most difficult aspects of programming. On any job, the description of what the user needs may be vague—worse yet, users may not really know what they want, and users who think they know frequently change their minds after seeing sample output. A good programmer is often part counselor, part detective!

Watch the video *The Program Development Cycle, Part 1*.

Planning the Logic

The heart of the programming process lies in planning the program's logic. During this phase of the process, the programmer plans the steps of the program, deciding what steps to include and how to order them. You can plan the solution to a problem in many ways. The two most common planning tools are flowcharts and pseudocode. Both tools involve writing the steps of the program in English, much as you would plan a trip on paper before getting into the car or plan a party theme before shopping for food and favors.

You may hear programmers refer to planning a program as "developing an algorithm." An **algorithm** is the sequence of steps necessary to solve any problem.

In addition to flowcharts and pseudocode, programmers use a variety of other tools to help in program development. One such tool is an **IPO chart**, which delineates input, processing, and output tasks. Some object-oriented programmers also use **TOE charts**, which list tasks, objects, and events.

The programmer shouldn't worry about the syntax of any particular language during the planning stage, but should focus on figuring out what sequence of events will lead from the available input to the desired output. Planning the logic includes thinking carefully about all

the possible data values a program might encounter and how you want the program to handle each scenario. The process of walking through a program's logic on paper before you actually write the program is called **desk-checking**. You will learn more about planning the logic throughout this book; in fact, the book focuses on this crucial step almost exclusively.

Coding the Program

After the logic is developed, only then can the programmer write the source code for a program. Hundreds of programming languages are available. Programmers choose particular languages because some have built-in capabilities that make them more efficient than others at handling certain types of operations. Despite their differences, programming languages are quite alike in their basic capabilities—each can handle input operations, arithmetic processing, output operations, and other standard functions. The logic developed to solve a programming problem can be executed using any number of languages. Only after choosing a language must the programmer be concerned with proper punctuation and the correct spelling of commands—in other words, using the correct *syntax*.

Some experienced programmers can successfully combine logic planning and program coding in one step. This may work for planning and writing a very simple program, just as you can plan and write a postcard to a friend using one step. A good term paper or a Hollywood screenplay, however, needs planning before writing—and so do most programs.

Which step is harder: planning the logic or coding the program? Right now, it may seem to you that writing in a programming language is a very difficult task, considering all the spelling and syntax rules you must learn. However, the planning step is actually more difficult. Which is more difficult: thinking up the twists and turns to the plot of a best-selling mystery novel, or writing a translation of an existing novel from English to Spanish? And who do you think gets paid more, the writer who creates the plot or the translator? (Try asking friends to name any famous translator!)

Using Software to Translate the Program into Machine Language

Even though there are many programming languages, each computer knows only one language—its machine language, which consists of 1s and 0s. Computers understand machine language because they are made up of thousands of tiny electrical switches, each of which can be set in either the on or off state, which is represented by a 1 or 0, respectively.

Languages like Java or Visual Basic are available for programmers because someone has written a translator program (a compiler or interpreter) that changes the programmer's English-like **high-level programming language** into the **low-level machine language** that the computer understands. When you learn the syntax of a programming language, the commands work on any machine on which the language software has been installed. However, your commands then are translated to machine language, which differs in various computer makes and models.

If you write a programming statement incorrectly (for example, by misspelling a word, using a word that doesn't exist in the language, or using "illegal" grammar), the translator program doesn't know how to proceed and issues an error message identifying a syntax error. Although making errors is never desirable, syntax errors are not a major concern to programmers, because the compiler or interpreter catches every syntax error and displays a message that notifies you of the problem. The computer will not execute a program that contains even one syntax error.

Typically, a programmer develops logic, writes the code, and compiles the program, receiving a list of syntax errors. The programmer then corrects the syntax errors and compiles the program again. Correcting the first set of errors frequently reveals new errors that originally were not apparent to the compiler. For example, if you could use an English compiler and submit the sentence *The dg chase the cat*, the compiler at first might point out only one syntax error. The second word, *dg*, is illegal because it is not part of the English language. Only after you corrected the word to *dog* would the compiler find another syntax error on the third word, *chase*, because it is the wrong verb form for the subject *dog*. This doesn't mean *chase* is necessarily the wrong word. Maybe *dog* is wrong; perhaps the subject should be *dogs*, in which case *chase* is right. Compilers don't always know exactly what you mean, nor do they know what the proper correction should be, but they do know when something is wrong with your syntax.

Watch the video *The Program Development Cycle, Part 2*.

When writing a program, a programmer might need to recompile the code several times. An executable program is created only when the code is free of syntax errors. After a program has been translated into machine language, the machine language program is saved and can be run any number of times without repeating the translation step. You only need to retranslate your code if you make changes to your source code statements. Figure 1-2 shows a diagram of this entire process.

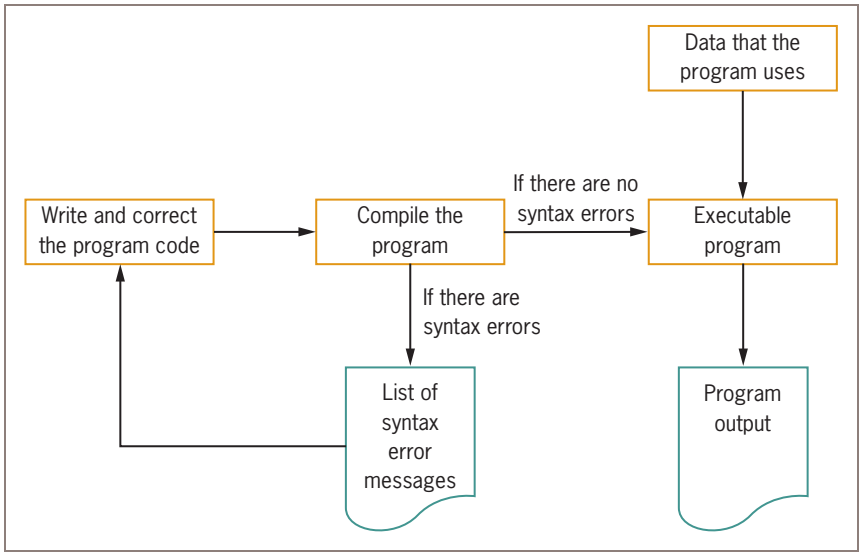


Figure 1-2 Creating an executable program

Testing the Program

A program that is free of syntax errors is not necessarily free of logical errors. A logical error results when you use a syntactically correct statement but use the wrong one for the current context. For example, the English sentence *The dog chases the cat*, although syntactically perfect, is not logically correct if the dog chases a ball or the cat is the aggressor.

Once a program is free of syntax errors, the programmer can test it—that is, execute it with some sample data to see whether the results are logically correct. Recall the number-doubling program:

```
input myNumber
set myAnswer = myNumber * 2
output myAnswer
```

If you execute the program, provide the value 2 as input to the program, and the answer 4 is displayed, you have executed one successful test run of the program.

However, if the answer 40 is displayed, maybe the program contains a logical error. Maybe the second line of code was mistyped with an extra zero, so that the program reads:

```
input myNumber
set myAnswer = myNumber * 20
output myAnswer
```

Don't Do It
The programmer typed *20* instead of *2*.

Placing 20 instead of 2 in the multiplication statement caused a logical error. Notice that nothing is syntactically wrong with this second program—it is just as reasonable to multiply a number by 20 as by 2—but if the programmer intends only to double `myNumber`, then a logical error has occurred.

The process of finding and correcting program errors is called **debugging**. You debug a program by testing it using many sets of data. For example, if you write the program to double a number, then enter 2 and get an output value of 4, that doesn't necessarily mean you have a correct program. Perhaps you have typed this program by mistake:

```
input myNumber
set myAnswer = myNumber + 2
output myAnswer
```

Don't Do It
The programmer typed "+" instead of "*".

An input of 2 results in an answer of 4, but that doesn't mean your program doubles numbers—it actually only adds 2 to them. If you test your program with additional data and get the wrong answer—for example, if you enter 7 and get an answer of 9—you know there is a problem with your code.

Selecting test data is somewhat of an art in itself, and it should be done carefully. If the Human Resources department wants a list of the names of five-year employees, it would be a mistake to test the program with a small sample file of only long-term employees. If no newer employees are part of the data being used for testing, you do not really know if the program would have eliminated them from the five-year list. Many companies do not know that their software has a problem until an unusual circumstance occurs—for example, the first time an employee has more than nine dependents, the first time a customer orders more than 999 items at a time, or when the Internet runs out of allocated IP addresses, a problem known as *IPV4 exhaustion*.

Putting the Program into Production

Once the program is thoroughly tested and debugged, it is ready for the organization to use. Putting the program into production might mean simply running the program once, if it was written to satisfy a user's request for a special list. However, the process might take months if the program will be run on a regular basis, or if it is one of a large system of programs being developed. Perhaps data-entry people must be trained to prepare the input for the new program, users must be trained to understand the output, or existing data in the company must be changed to an entirely new format to accommodate this program. **Conversion**, the entire set of actions an organization must take to switch over to using a new program or set of programs, can sometimes take months or years to accomplish.

Maintaining the Program

After programs are put into production, making necessary changes is called **maintenance**. Maintenance can be required for many reasons: for example, because new tax rates are legislated, the format of an input file is altered, or the end user requires additional information not included in the original output specifications. Frequently, your first programming job will require maintaining previously written programs. When you maintain the programs others have written, you will appreciate the effort the original programmer put into writing clear

code, using reasonable variable names, and documenting his or her work. When you make changes to existing programs, you repeat the development cycle. That is, you must understand the changes, then plan, code, translate, and test them before putting them into production. If a substantial number of program changes are required, the original program might be retired, and the program development cycle might be started for a new program.

Watch the video *The Program Development Cycle, Part 3*.

TWO TRUTHS & A LIE

Understanding the Program Development Cycle

1. Understanding the problem that must be solved can be one of the most difficult aspects of programming.
2. The two most commonly used logic-planning tools are flowcharts and pseudocode.
3. Flowcharting a program is a very different process if you use an older programming language instead of a newer one.

The false statement is #3. Despite their differences, programming languages are quite alike in their basic capabilities—each can handle input operations, arithmetic processing, output operations, and other standard functions. The logic developed to solve a programming problem can be executed using any number of languages.

Using Pseudocode Statements and Flowchart Symbols

When programmers plan the logic for a solution to a programming problem, they often use one of two tools: pseudocode (pronounced *sue-doe-code*) or flowcharts.

- **Pseudocode** is an English-like representation of the logical steps it takes to solve a problem. *Pseudo* is a prefix that means *false*, and to *code* a program means to put it in a programming language; therefore, *pseudocode* simply means *false code*, or sentences that appear to have been written in a computer programming language but do not necessarily follow all the syntax rules of any specific language.
- A **flowchart** is a pictorial representation of the same thing.

Writing Pseudocode

You have already seen examples of statements that represent pseudocode earlier in this chapter, and there is nothing mysterious about them. The following five statements constitute a pseudocode representation of a number-doubling problem:

```
start
  input myNumber
  set myAnswer = myNumber * 2
  output myAnswer
stop
```

Using pseudocode involves writing down all the steps you will use in a program. Usually, programmers preface their pseudocode with a beginning statement like `start` and end it with a terminating statement like `stop`. The statements between `start` and `stop` look like English and are indented slightly so that `start` and `stop` stand out. Most programmers do not bother with punctuation such as periods at the end of pseudocode statements, although it would not be wrong to use them if you prefer that style. Similarly, there is no need to capitalize the first word in a sentence, although you might choose to do so. This book follows the conventions of using lowercase letters for verbs that begin pseudocode statements and omitting periods at the end of statements.

Pseudocode is fairly flexible because it is a planning tool, and not the final product. Therefore, for example, you might prefer any of the following:

- Instead of `start` and `stop`, some pseudocode developers would use other terms such as `begin` and `end`.
- Instead of writing `input myNumber`, some developers would write `get myNumber` or `read myNumber`.
- Instead of writing `set myAnswer = myNumber * 2`, some developers would write `calculate myAnswer = myNumber times 2` or `compute myAnswer as myNumber doubled`.
- Instead of writing `output myAnswer`, many pseudocode developers would write `display myAnswer`, `print myAnswer`, or `write myAnswer`.

The point is, the pseudocode statements are instructions to retrieve an original number from an input device and store it in memory where it can be used in a calculation, and then to get the calculated answer from memory and send it to an output device so a person can see it. When you eventually convert your pseudocode to a specific programming language, you do not have such flexibility because specific syntax will be required. For example, if you use the C# programming language and write the statement to output the answer to the monitor, you will code the following:

```
Console.Write(myAnswer);
```

The exact use of words, capitalization, and punctuation are important in the C# statement, but not in the pseudocode statement.

Drawing Flowcharts

Some professional programmers prefer writing pseudocode to drawing flowcharts, because using pseudocode is more similar to writing the final statements in the programming language. Others prefer drawing flowcharts to represent the logical flow, because flowcharts allow programmers to visualize more easily how the program statements will connect. Especially for beginning programmers, flowcharts are an excellent tool to help them visualize how the statements in a program are interrelated.

You can draw a flowchart by hand or use software, such as Microsoft Word and Microsoft PowerPoint, that contains flowcharting tools. You can use several other software programs, such as Visio and Visual Logic, specifically to create flowcharts. When you create a flowchart, you draw geometric shapes that contain the individual statements and that are connected with arrows. (Appendix B contains a summary of all the flowchart symbols you will see in this book.) You use a parallelogram to represent an **input symbol**, which indicates an input operation. You write an input statement in English inside the parallelogram, as shown in Figure 1-3.

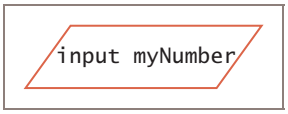


Figure 1-3 Input symbol

Arithmetic operation statements are examples of processing. In a flowchart, you use a rectangle as the **processing symbol** that contains a processing statement, as shown in Figure 1-4.

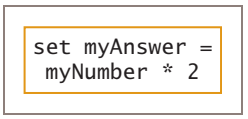


Figure 1-4 Processing symbol

To represent an output statement, you use the same symbol as for input statements—the **output symbol** is a parallelogram, as shown in Figure 1-5. Because the parallelogram is used for both input and output, it is often called the **input/output symbol** or **I/O symbol**.

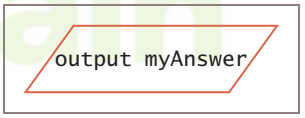


Figure 1-5 Output symbol

Some software programs that use flowcharts (such as Visual Logic) use a left-slanting parallelogram to represent output. As long as the flowchart creator and the flowchart reader are communicating, the actual shape used is irrelevant. This book will follow the most standard convention of using the right-slanting parallelogram for both input and output.

To show the correct sequence of these statements, you use arrows, or **flowlines**, to connect the steps. Whenever possible, most of a flowchart should read from top to bottom or from left to right on a page. That's the way we read English, so when flowcharts follow this convention, they are easier for us to understand.

To be complete, a flowchart should include two more elements: **terminal symbols**, or start/stop symbols, at each end. Often, you place a word like `start` or `begin` in the first terminal symbol and a word like `end` or `stop` in the other. The standard terminal symbol is shaped like a racetrack; many programmers refer to this shape as a lozenge, because it resembles the shape of the medication you might use to soothe a sore throat. Figure 1-6 shows a complete flowchart for the program that doubles a number, and the pseudocode for the same problem.

You can see from the figure that the flowchart and pseudocode statements are the same—only the presentation format differs.

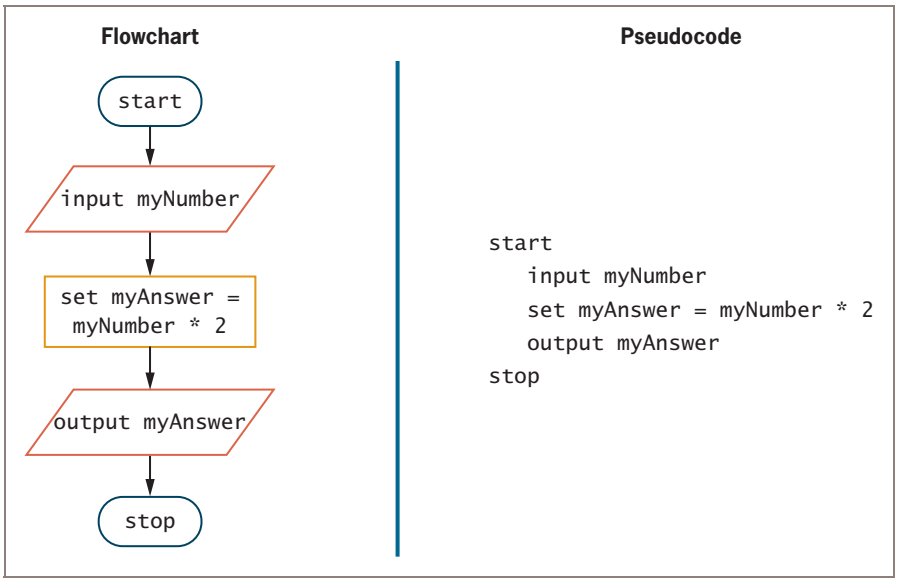


Figure 1-6 Flowchart and pseudocode of program that doubles a number

Programmers seldom create both pseudocode and a flowchart for the same problem. You usually use one or the other. In a large program, you might even prefer to write pseudocode for some parts and to draw a flowchart for others.

When you tell a friend how to get to your house, you might write a series of instructions or you might draw a map. Pseudocode is similar to written, step-by-step instructions; a flowchart, like a map, is a visual representation of the same thing.

Repeating Instructions

After the flowchart or pseudocode has been developed, the programmer only needs to: (1) buy a computer, (2) buy a language compiler, (3) learn a programming language, (4) code the program, (5) attempt to compile it, (6) fix the syntax errors, (7) compile it again, (8) test it with several sets of data, and (9) put it into production.

"Whoa!" you are probably saying to yourself. "This is simply not worth it! All that work to create a flowchart or pseudocode, and *then* all those other steps? For five dollars, I can buy a pocket calculator that will double any number for me instantly!" You are absolutely right. If this were a real computer program, and all it did was double the value of a number, it would not be worth the effort. Writing a computer program would be worthwhile only if you had many numbers (let's say 10,000) to double in a limited amount of time—let's say the next two minutes.

Unfortunately, the program represented in Figure 1-6 does not double 10,000 numbers; it doubles only one. You could execute the program 10,000 times, of course, but that would require you to sit at the computer and run the program over and over again. You would be better off with a program that could process 10,000 numbers, one after the other.

One solution is to write the program shown in Figure 1-7 and execute the same steps 10,000 times. Of course, writing this program would be very time consuming; you might as well buy the calculator.

```
start
  input myNumber
  set myAnswer = myNumber * 2
  output myAnswer
  input myNumber
  set myAnswer = myNumber * 2
  output myAnswer
  input myNumber
  set myAnswer = myNumber * 2
  output myAnswer
  …and so on for 9,997 more times
```

Don't Do It
You would never want to write such a repetitious list of instructions.

Figure 1-7 Inefficient pseudocode for program that doubles 10,000 numbers

A better solution is to have the computer execute the same set of three instructions over and over again, as shown in Figure 1-8. The repetition of a series of steps is called a **loop**. With this approach, the computer gets a number, doubles it, displays the answer, and then starts again with the first instruction. The same spot in memory, called `myNumber`, is reused for the second number and for any subsequent numbers. The spot in memory named `myAnswer` is reused each time to store the result of the multiplication operation. However, the logic illustrated in the flowchart in Figure 1-8 contains a major problem—the sequence of instructions never ends. This programming situation is known as an **infinite loop**—a repeating flow of logic with no end. You will learn one way to handle this problem later in this chapter; you will learn a superior way in Chapter 3.

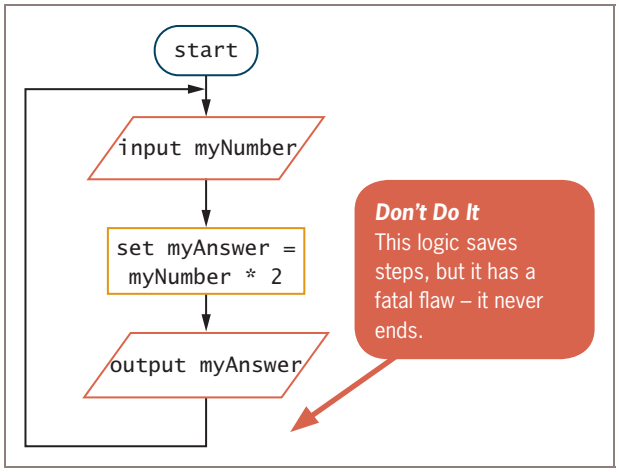


Figure 1-8 Flowchart of infinite number-doubling program

TWO TRUTHS & A LIE

Using Pseudocode Statements and Flowchart Symbols

1. When you draw a flowchart, you use a parallelogram to represent an input operation.
2. When you draw a flowchart, you use a parallelogram to represent a processing operation.
3. When you draw a flowchart, you use a parallelogram to represent an output operation.

The false statement is #2. When you draw a flowchart, you use a rectangle to represent a processing operation.

Using a Sentinel Value to End a Program

The logic in the flowchart for doubling numbers, shown in Figure 1-8, has a major flaw—the program contains an infinite loop. If, for example, the input numbers are being entered at the keyboard, the program will keep accepting numbers and outputting their doubled values forever. Of course, the user could refuse to type any more numbers. But the program cannot progress any further while it is waiting for input; meanwhile, the program is occupying computer memory and tying up operating system resources. Refusing to enter any more numbers is not a practical solution. Another way to end the program is simply to turn off the

computer. But again, that's neither the best solution nor an elegant way for the program to end.

A superior way to end the program is to set a predetermined value for `myNumber` that means "Stop the program!" For example, the programmer and the user could agree that the user will never need to know the double of 0 (zero), so the user could enter a 0 to stop. The program could then test any incoming value contained in `myNumber` and, if it is a 0, stop the program. Testing a value is also called **making a decision**.

You represent a decision in a flowchart by drawing a **decision symbol**, which is shaped like a diamond. The diamond usually contains a question, the answer to which is one of two mutually exclusive options—often yes or no. All good computer questions have only two mutually exclusive answers, such as yes and no or true and false. For example, "What day of the year is your birthday?" is not a good computer question because there are 366 possible answers. However, "Is your birthday June 24?" is a good computer question because the answer is always either yes or no.

The question to stop the doubling program should be "Is the value of `myNumber` just entered equal to 0?" or "`myNumber = 0`?" for short. The complete flowchart will now look like the one shown in Figure 1-9.

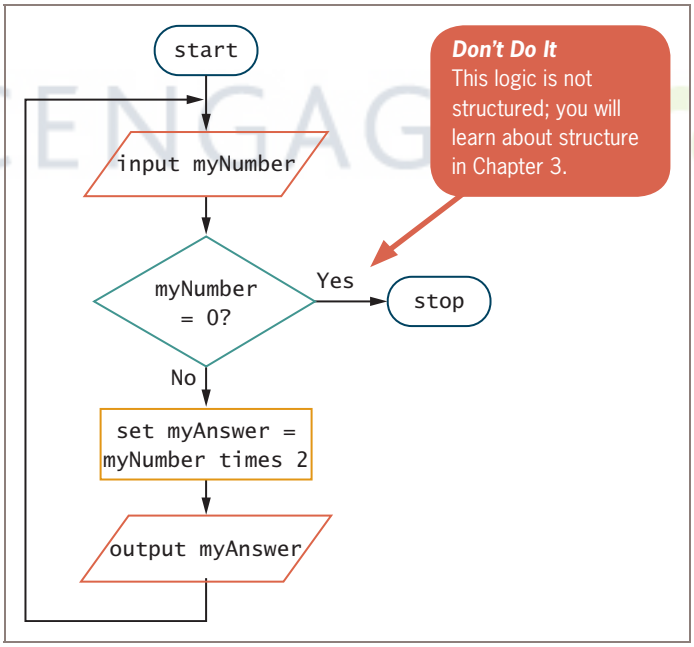


Figure 1-9 Flowchart of number-doubling program with sentinel value of 0

One drawback to using 0 to stop a program, of course, is that it won't work if the user does need to find the double of 0. In that case, some other data-entry value that the user never will

need, such as 999 or –1, could be selected to signal that the program should end. A preselected value that stops the execution of a program is often called a **dummy value** because it does not represent real data, but just a signal to stop. Sometimes, such a value is called a **sentinel value** because it represents an entry or exit point, like a sentinel who guards a fortress.

Not all programs rely on user data entry from a keyboard; many read data from an input device, such as a disk. When organizations store data on a disk or other storage device, they do not commonly use a dummy value to signal the end of the file. For one thing, an input record might have hundreds of fields, and if you store a dummy record in every file, you are wasting a large quantity of storage on "nondata." Additionally, it is often difficult to choose sentinel values for fields in a company's data files. Any `balanceDue`, even a zero or a negative number, can be a legitimate value, and any `customerName`, even "ZZ", could be someone's name. Fortunately, programming languages can recognize the end of data in a file automatically, through a code that is stored at the end of the data. Many programming languages use the term **eof** (for *end of file*) to refer to this marker that automatically acts as a sentinel. This book, therefore, uses `eof` to indicate the end of data whenever using a dummy value is impractical or inconvenient. In the flowchart shown in Figure 1-10, the `eof` question is shaded.

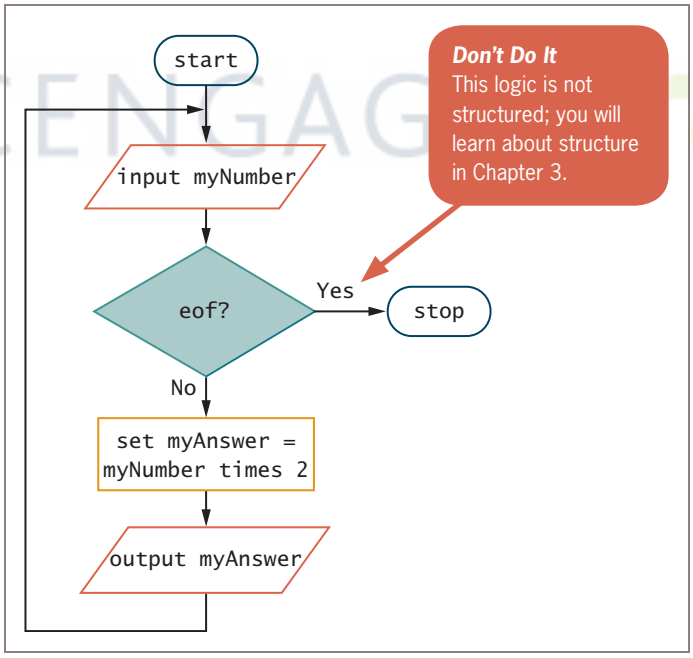


Figure 1-10 Flowchart using `eof`

TWO TRUTHS & A LIE

Using a Sentinel Value to End a Program

1. A program that contains an infinite loop is one that never ends.
2. A preselected value that stops the execution of a program is often called a dummy value or a sentinel value.
3. Many programming languages use the term `fe` (for *file end*) to refer to a marker that automatically acts as a sentinel.

The false statement is #3. The term `eof` (for *end of file*) is the common term for a file sentinel.

Understanding Programming and User Environments

Many approaches can be used to write and execute a computer program. When you plan a program's logic, you can use a flowchart, pseudocode, or a combination of the two. When you code the program, you can type statements into a variety of text editors. When your program executes, it might accept input from a keyboard, mouse, microphone, or any other input device, and when you provide a program's output, you might use text, images, or sound. This section describes the most common environments you will encounter as a new programmer.

Understanding Programming Environments

When you plan the logic for a computer program, you can use paper and pencil to create a flowchart, or you might use software that allows you to manipulate flowchart shapes. If you choose to write pseudocode, you can do so by hand or by using a word-processing program. To enter the program into a computer so you can translate and execute it, you usually use a keyboard to type program statements into an editor. You can type a program into one of the following:

- A plain text editor
- A text editor that is part of an integrated development environment

A **text editor** is a program that you use to create simple text files. It is similar to a word processor, but without as many features. You can use a text editor such as Notepad that is included with Microsoft Windows. Figure 1-11 shows a C# program in Notepad that accepts a number and doubles it. An advantage to using a simple text editor to type and save a program is that the completed program does not require much disk space for storage. For example, the file shown in Figure 1-11 occupies only 314 bytes of storage.

This line contains a prompt that tells the user what to enter. You will learn more about prompts in Chapter 2.

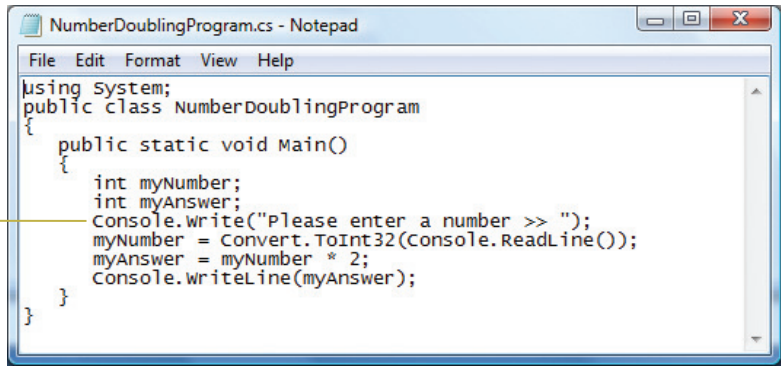


Figure 1-11 A C# number-doubling program in Notepad

You can use the editor of an **integrated development environment** (**IDE**) to enter your program. An IDE is a software package that provides an editor, compiler, and other programming tools. For example, Figure 1-12 shows a C# program in the **Microsoft Visual Studio IDE**, an environment that contains tools useful for creating programs in Visual Basic, C++, and C#.

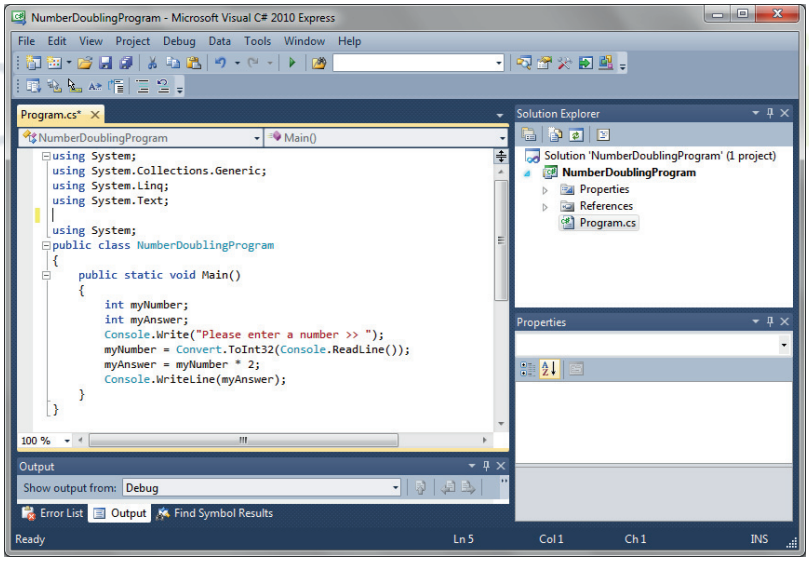


Figure 1-12 A C# number-doubling program in Visual Studio

Using an IDE is helpful to programmers because usually it provides features similar to those you find in many word processors. In particular, an IDE's editor commonly includes such features as the following:

- It uses different colors to display various language components, making elements like data types easier to identify.
- It highlights syntax errors visually for you.
- It employs automatic statement completion; when you start to type a statement, the IDE suggests a likely completion, which you can accept with a keystroke.
- It provides tools that allow you to step through a program's execution one statement at a time so you can more easily follow the program's logic and determine the source of any errors.

When you use the IDE to create and save a program, you occupy much more disk space than when using a plain text editor. For example, the program in Figure 1-12 occupies more than 49,000 bytes of disk space.

Although various programming environments might look different and offer different features, the process of using them is very similar. When you plan the logic for a program using pseudocode or a flowchart, it does not matter which programming environment you will use to write your code, and when you write the code in a programming language, it does not matter which environment you use to write it.

Understanding User Environments

A user might execute a program you have written in any number of environments. For example, a user might execute the number-doubling program from a command line like the one shown in Figure 1-13. A **command line** is a location on your computer screen at which you type text entries to communicate with the computer's operating system. In the program in Figure 1-13, the user is asked for a number, and the results are displayed.

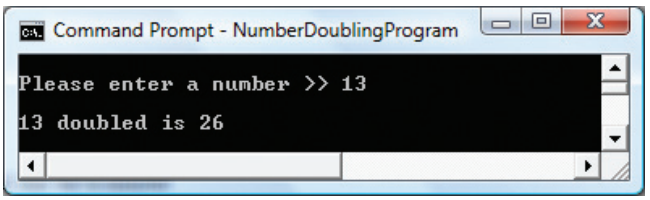


Figure 1-13 Executing a number-doubling program in a command-line environment

Many programs are not run at the command line in a text environment, but are run using a **graphical user interface**, or **GUI** (pronounced *gooey*), which allows users to interact with a program in a graphical environment. When running a GUI program, the user might type input into a text box or use a mouse or other pointing device to select options on the screen. Figure 1-14 shows a number-doubling program that performs exactly the same task as the one in Figure 1-13, but this program uses a GUI.

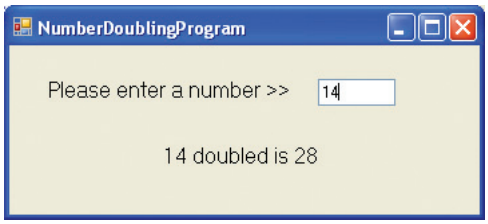


Figure 1-14 Executing a number-doubling program in a GUI environment

A command-line program and a GUI program might be written in the same programming language. (For example, the programs shown in Figures 1-13 and 1-14 were both written using C#.) However, no matter which environment is used to write or execute a program, the logical process is the same. The two programs in Figures 1-13 and 1-14 both accept input, perform multiplication, and perform output. In this book, you will not concentrate on which environment is used to type a program's statements, nor will you care about the type of environment the user will see. Instead, you will be concerned with the logic that applies to all programming situations.

TWO TRUTHS & A LIE

Understanding Programming and User Environments

1. You can type a program into an editor that is part of an integrated development environment, but using a plain text editor provides you with more programming help.
2. When a program runs from the command line, a user types text to provide input.
3. Although GUI and command-line environments look different, the logic of input, processing, and output apply to both program types.

The false statement is #1. An integrated development environment provides more programming help than a plain text editor.

Understanding the Evolution of Programming Models

People have been writing modern computer programs since the 1940s. The oldest programming languages required programmers to work with memory addresses and to memorize awkward codes associated with machine languages. Newer programming languages look much more like natural language and are easier to use, partly because they allow programmers to name variables instead of using unwieldy memory addresses. Also,

newer programming languages allow programmers to create self-contained modules or program segments that can be pieced together in a variety of ways. The oldest computer programs were written in one piece, from start to finish, but modern programs are rarely written that way—they are created by teams of programmers, each developing reusable and connectable program procedures. Writing several small modules is easier than writing one large program, and most large tasks are easier when you break the work into units and get other workers to help with some of the units.

Ada Byron Lovelace predicted the development of software in 1843; she is often regarded as the first programmer. The basis for most modern software was proposed by Alan Turing in 1935.

Currently, two major models or paradigms are used by programmers to develop programs and their procedures:

- **Procedural programming** focuses on the procedures that programmers create. That is, procedural programmers focus on the actions that are carried out—for example, getting input data for an employee and writing the calculations needed to produce a paycheck from the data. Procedural programmers would approach the job of producing a paycheck by breaking down the process into manageable subtasks.
- **Object-oriented programming** focuses on objects, or "things," and describes their features (also called attributes) and behaviors. For example, object-oriented programmers might design a payroll application by thinking about employees and paychecks, and by describing their attributes. Employees have names and Social Security numbers, and paychecks have names and check amounts. Then the programmers would think about the behaviors of employees and paychecks, such as employees getting raises and adding dependents and paychecks being calculated and output. Object-oriented programmers would then build applications from these entities.

With either approach, procedural or object oriented, you can produce a correct paycheck, and both models employ reusable program modules. The major difference lies in the focus the programmer takes during the earliest planning stages of a project. For now, this book focuses on procedural programming techniques. The skills you gain in programming procedurally—declaring variables, accepting input, making decisions, producing output, and so on—will serve you well whether you eventually write programs using a procedural approach, an object-oriented approach, or both. The programming language in which you write your source code might determine your approach. You can write a procedural program in any language that supports object orientation, but the opposite is not always true.

TWO TRUTHS & A LIE

Understanding the Evolution of Programming Models

1. The oldest computer programs were written in many separate modules.
2. Procedural programmers focus on actions that are carried out by a program.
3. Object-oriented programmers focus on a program's objects and their attributes and behaviors.

The false statement is #1. The oldest programs were written in a single piece; newer programs are divided into modules.

Chapter Summary

- Together, computer hardware (physical devices) and software (instructions) accomplish three major operations: input, processing, and output. You write computer instructions in a computer programming language that requires specific syntax; the instructions are translated into machine language by a compiler or interpreter. When both the syntax and logic of a program are correct, you can run, or execute, the program to produce the desired results.
- For a program to work properly, you must develop correct logic. Logical errors are much more difficult to locate than syntax errors.
- A programmer's job involves understanding the problem, planning the logic, coding the program, translating the program into machine language, testing the program, putting the program into production, and maintaining it.
- When programmers plan the logic for a solution to a programming problem, they often use flowcharts or pseudocode. When you draw a flowchart, you use parallelograms to represent input and output operations, and rectangles to represent processing. Programmers also use decisions to control repetition of instruction sets.
- To avoid creating an infinite loop when you repeat instructions, you can test for a sentinel value. You represent a decision in a flowchart by drawing a diamond-shaped symbol that contains a question, the answer to which is either yes or no.
- You can type a program into a plain text editor or one that is part of an integrated development environment. When a program's data values are entered from a keyboard, they can be entered at the command line in a text environment or in a GUI. Either way, the logic is similar.

- Procedural and object-oriented programmers approach problems differently. Procedural programmers concentrate on the actions performed with data. Object-oriented programmers focus on objects and their behaviors and attributes.

Key Terms

A **computer system** is a combination of all the components required to process and store data using a computer.

Hardware is the collection of physical devices that comprise a computer system.

Software consists of the programs that tell the computer what to do.

Programs are sets of instructions for a computer.

Programming is the act of developing and writing programs.

Application software comprises all the programs you apply to a task.

System software comprises the programs that you use to manage your computer.

Input describes the entry of data items into computer memory using hardware devices such as keyboards and mice.

Data items include all the text, numbers, and other information processed by a computer.

Processing data items may involve organizing them, checking them for accuracy, or performing mathematical operations on them.

The **central processing unit**, or **CPU**, is the hardware component that processes data.

Output describes the operation of retrieving information from memory and sending it to a device, such as a monitor or printer, so people can view, interpret, and work with the results.

Information is processed data.

Storage devices are types of hardware equipment, such as disks, that hold information for later retrieval.

Programming languages, such as Visual Basic, C#, C++, Java, or COBOL, are used to write programs.

Program code is the set of instructions a programmer writes in a programming language.

Coding the program is the act of writing programming language instructions.

The **syntax** of a language is its grammar rules.

A **syntax error** is an error in language or grammar.

Computer memory is the temporary, internal storage within a computer.

Random access memory (**RAM**) is temporary, internal computer storage.

Volatile describes storage whose contents are lost when power is lost.

Nonvolatile describes storage whose contents are retained when power is lost.

Machine language is a computer's on/off circuitry language.

Source code is the statements a programmer writes in a programming language.

Object code is translated machine language.

A **compiler** or **interpreter** translates a high-level language into machine language and indicates if you have used a programming language incorrectly.

Binary language is represented using a series of 0s and 1s.

To **run** or **execute** a program is to carry out its instructions.

Scripting languages (also called **scripting programming languages** or **script languages**) such as Python, Lua, Perl, and PHP are used to write programs that are typed directly from a keyboard. Scripting languages are stored as text rather than as binary executable files.

A **logical error** occurs when incorrect instructions are performed, or when instructions are performed in the wrong order.

You develop the **logic** of the computer program when you give instructions to the computer in a specific sequence, without omitting any instructions or adding extraneous instructions.

A **variable** is a named memory location whose value can vary.

The **program development cycle** consists of the steps that occur during a program's lifetime.

Users (or **end users**) are people who employ and benefit from computer programs.

Documentation consists of all the supporting paperwork for a program.

An **algorithm** is the sequence of steps necessary to solve any problem.

An **IPO chart** is a program development tool that delineates input, processing, and output tasks.

A **TOE chart** is a program development tool that lists tasks, objects, and events.

Desk-checking is the process of walking through a program solution on paper.

A **high-level programming language** supports English-like syntax.

A **low-level machine language** is made up of 1s and 0s and does not use easily interpreted variable names.

Debugging is the process of finding and correcting program errors.

Conversion is the entire set of actions an organization must take to switch over to using a new program or set of programs.

Maintenance consists of all the improvements and corrections made to a program after it is in production.

Pseudocode is an English-like representation of the logical steps it takes to solve a problem.

A **flowchart** is a pictorial representation of the logical steps it takes to solve a problem.

An **input symbol** indicates an input operation and is represented by a parallelogram in flowcharts.

A **processing symbol** indicates a processing operation and is represented by a rectangle in flowcharts.

An **output symbol** indicates an output operation and is represented by a parallelogram in flowcharts.

An **input/output symbol** or **I/O symbol** is represented by a parallelogram in flowcharts.

Flowlines, or arrows, connect the steps in a flowchart.

A **terminal symbol** indicates the beginning or end of a flowchart segment and is represented by a lozenge

A **loop** is a repetition of a series of steps.

An **infinite loop** occurs when repeating logic cannot end.

Making a decision is the act of testing a value.

A **decision symbol** is shaped like a diamond and used to represent decisions in flowcharts.

A **dummy value** is a preselected value that stops the execution of a program.

A **sentinel value** is a preselected value that stops the execution of a program.

The term **eof** means *end of file*.

A **text editor** is a program that you use to create simple text files; it is similar to a word processor, but without as many features.

An **integrated development environment** (**IDE**) is a software package that provides an editor, compiler, and other programming tools.

Microsoft Visual Studio IDE is a software package that contains useful tools for creating programs in Visual Basic, C++, and C#.

A **command line** is a location on your computer screen at which you type text entries to communicate with the computer's operating system.

A **graphical user interface**, or **GUI** (pronounced *gooey*), allows users to interact with a program in a graphical environment.

Procedural programming is a programming model that focuses on the procedures that programmers create.

Object-oriented programming is a programming model that focuses on objects, or "things," and describes their features (also called attributes) and behaviors.

Review Questions

1. Computer programs also are known as _____.

   a. hardware
   b. software
   c. data
   d. information

2. The major computer operations include _____.

   a. hardware and software
   b. input, processing, and output
   c. sequence and looping
   d. spreadsheets, word processing, and data communications

3. Visual Basic, C++, and Java are all examples of computer _____.

   a. operating systems
   b. hardware
   c. machine languages
   d. programming languages

4. A programming language's rules are its _____.

   a. syntax
   b. logic
   c. format
   d. options

5. The most important task of a compiler or interpreter is to _____.

   a. create the rules for a programming language
   b. translate English statements into a language such as Java
   c. translate programming language statements into machine language
   d. execute machine language programs to perform useful tasks

6. Which of the following is temporary, internal storage?

   a. CPU
   b. hard disk
   c. keyboard
   d. memory

7. Which of the following pairs of steps in the programming process is in the correct order?

   a. code the program, plan the logic
   b. test the program, translate it into machine language
   c. put the program into production, understand the problem
   d. code the program, translate it into machine language

8. A programmer's most important task before planning the logic of a program is to _____.

   a. decide which programming language to use
   b. code the problem
   c. train the users of the program
   d. understand the problem

9. The two most commonly used tools for planning a program's logic are _____.

   a. flowcharts and pseudocode
   b. ASCII and EBCDIC
   c. Java and Visual Basic
   d. word processors and spreadsheets

10. Writing a program in a language such as C++ or Java is known as _____ the program.

    a. translating
    b. coding
    c. interpreting
    d. compiling

11. An English-like programming language such as Java or Visual Basic is a _____ programming language.

    a. machine-level
    b. low-level
    c. high-level
    d. binary-level

12. Which of the following is an example of a syntax error?

    a. producing output before accepting input
    b. subtracting when you meant to add
    c. misspelling a programming language word
    d. all of the above

13. Which of the following is an example of a logical error?

    a. performing arithmetic with a value before inputting it
    b. accepting two input values when a program requires only one
    c. dividing by 3 when you meant to divide by 30
    d. all of the above

14. The parallelogram is the flowchart symbol representing _____.

    a. input
    b. output
    c. both a and b
    d. none of the above

15. In a flowchart, a rectangle represents _____.

    a. input
    b. a sentinel
    c. a question
    d. processing

16. In flowcharts, the decision symbol is a _____.

    a. parallelogram
    b. rectangle
    c. lozenge
    d. diamond

17. The term `eof` represents _____.

    a. a standard input device
    b. a generic sentinel value
    c. a condition in which no more memory is available for storage
    d. the logical flow in a program

18. When you use an IDE instead of a simple text editor to develop a program, _____.

    a. the logic is more complicated
    b. the logic is simpler
    c. the syntax is different
    d. some help is provided

19. When you write a program that will run in a GUI environment as opposed to a command-line environment, _____.

    a. the logic is very different
    b. some syntax is different
    c. you do not need to plan the logic
    d. users are more confused

20. As compared to procedural programming, with object-oriented programming, _____.

    a. the programmer's focus differs
    b. you cannot use some languages, such as Java
    c. you do not accept input
    d. you do not code calculations; they are created automatically

Exercises

1. Match the definition with the appropriate term.

1. Computer system devices	a. compiler
2. Another word for *programs*	b. syntax
3. Language rules	c. logic
4. Order of instructions	d. hardware
5. Language translator	e. software

2. In your own words, describe the steps to writing a computer program.

3. Match the term with the appropriate shape (see Figure 1-15).

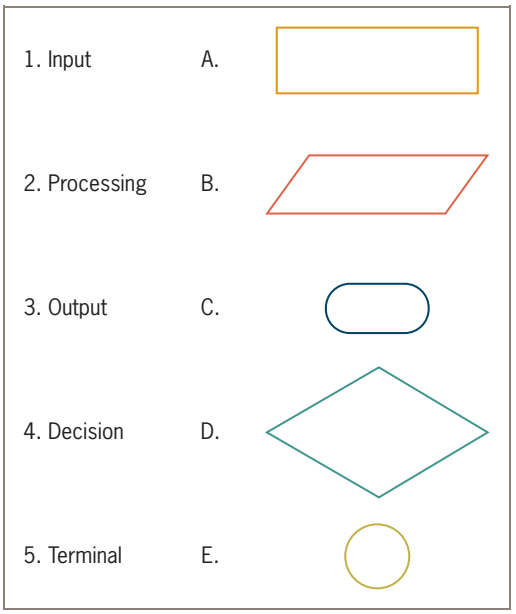


Figure 1-15 Identifying shapes

4. Draw a flowchart or write pseudocode to represent the logic of a program that allows the user to enter a value. The program divides the value by 2 and outputs the result.
5. Draw a flowchart or write pseudocode to represent the logic of a program that allows the user to enter a value for one edge of a cube. The program calculates the surface area of one side of the cube, the surface area of the cube, and its volume. The program outputs all the results.
6. Draw a flowchart or write pseudocode to represent the logic of a program that allows the user to enter two values. The program outputs the product of the two values.
7. a. Draw a flowchart or write pseudocode to represent the logic of a program that allows the user to enter values for the width and length of a room's floor in feet. The program outputs the area of the floor in square feet.

   b. Modify the program that computes floor area to compute and output the number of 12-inch square tiles needed to tile the floor.

8. a. Draw a flowchart or write pseudocode to represent the logic of a program that allows the user to enter values for the width and length of a wall in feet. The program outputs the area of the wall in square feet.

   b. Modify the program that computes wall area to allow the user to enter the price of a gallon of paint. Assume that a gallon of paint covers 350 square feet of a wall. The program outputs the number of gallons needed and the cost of the job. (For this exercise, assume that you do not need to account for windows or doors, and that you can purchase partial gallons of paint.)

9. Research current rates of monetary exchange. Draw a flowchart or write pseudocode to represent the logic of a program that allows the user to enter a number of dollars and convert it to Euros and Japanese yen.

10. Draw a flowchart or write pseudocode to represent the logic of a program that allows the user to enter values for a salesperson's base salary, total sales, and commission rate. The program computes and outputs the salesperson's pay by adding the base salary to the product of the total sales and commission rate.

Find the Bugs

11. Since the early days of computer programming, program errors have been called bugs. The term is often said to have originated from an actual moth that was discovered trapped in the circuitry of a computer at Harvard University in 1945. Actually, the term *bug* was in use prior to 1945 to mean trouble with any electrical apparatus; even during Thomas Edison's life, it meant an industrial defect. However, the term *debugging* is more closely associated with correcting program syntax and logic errors than with any other type of trouble.

    Your downloadable files for Chapter 1 include DEBUG01-01.txt, DEBUG01-02.txt, and DEBUG01-03.txt. Each file starts with some comments (lines that begin with two slashes) that describe the program. Examine the pseudocode that follows the introductory comments, then find and correct all the bugs.

Game Zone

12. In 1952, A. S. Douglas wrote his University of Cambridge Ph.D. dissertation on human-computer interaction, and created the first graphical computer game—a version of Tic-Tac-Toe. The game was programmed on an EDSAC vacuum-tube mainframe computer. The first computer game is generally assumed to be *Spacewar!*, developed in 1962 at MIT; the first commercially available video game was *Pong*, introduced by Atari in 1972. In 1980, Atari's *Asteroids* and *Lunar Lander* became the first video games to be registered with the U. S. Copyright Office. Throughout the

1980s, players spent hours with games that now seem very simple and unglamorous; do you recall playing *Adventure*, *Oregon Trail*, *Where in the World Is Carmen Sandiego?*, or *Myst*?

Today, commercial computer games are much more complex; they require many programmers, graphic artists, and testers to develop them, and large management and marketing staffs are needed to promote them. A game might cost many millions of dollars to develop and market, but a successful game might earn hundreds of millions of dollars. Obviously, with the brief introduction to programming you have had in this chapter, you cannot create a very sophisticated game. However, you can get started.

Mad Libs® is a children's game in which players provide a few words that are then incorporated into a silly story. The game helps children understand different parts of speech because they are asked to provide specific types of words. For example, you might ask a child for a noun, another noun, an adjective, and a past-tense verb. The child might reply with such answers as *table*, *book*, *silly*, and *studied*. The newly created Mad Lib might be:

Mary had a little *table*

Its *book* was *silly* as snow

And everywhere that Mary *studied*

The *table* was sure to go.

Create the logic for a Mad Lib program that accepts five words from input, then creates and displays a short story or nursery rhyme that uses them.

Up for Discussion

13. Which is the better tool for learning programming—flowcharts or pseudocode? Cite any educational research you can find.

14. What is the image of the computer programmer in popular culture? Is the image different in books than in TV shows and movies? Would you like that image for yourself?