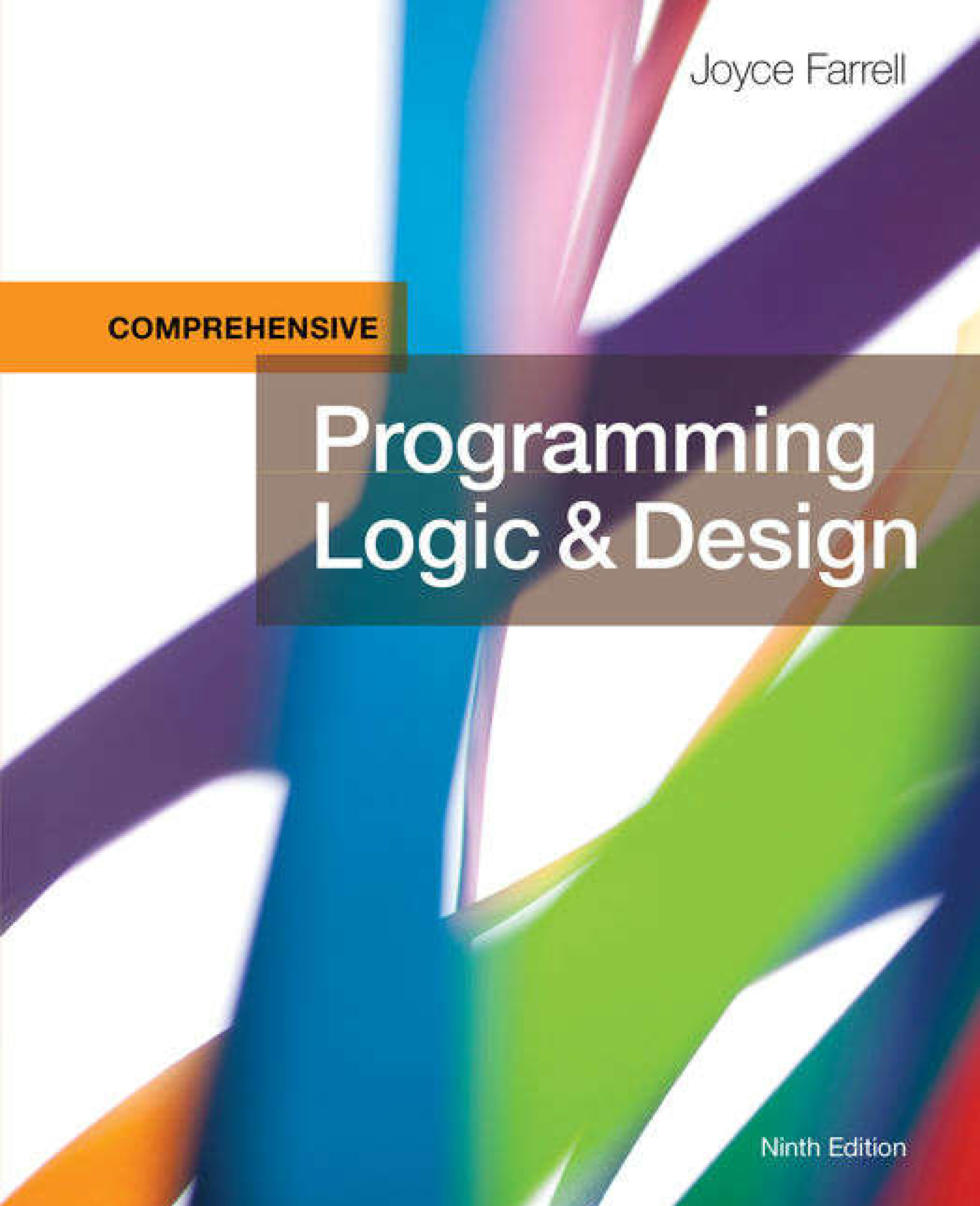


Joyce Farrell

COMPREHENSIVE

Programming Logic & Design

Ninth Edition

PROGRAMMING LOGIC AND DESIGN

COMPREHENSIVE

NINTH EDITION

PROGRAMMING LOGIC AND DESIGN

COMPREHENSIVE

JOYCE FARRELL

CENGAGE Learning®

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

CENGAGE
Learning®

Programming Logic and Design, Comprehensive
Ninth Edition
Joyce Farrell

Senior Product Director: Kathleen McMahon

Product Team Leader: Kristin McNary

Associate Product Manager: Kate Mason

Senior Content Developer: Alyssa Pratt

Senior Content Project Manager: Jennifer Feltri-George

Manufacturing Planner: Julio Esperas

Art Director: Diana Graham

Production Service/Composition: SPi Global

Cover Photo: Colormos/Photodisc/Getty Images

For product information and technology assistance, contact us at
Cengage Learning Customer & Sales Support, 1-800-354-9706

For permission to use material from this text or product,
submit all requests online at **www.cengage.com/permissions**
Further permissions questions can be emailed to
permissionrequest@cengage.com

Library of Congress Control Number: 2016959742

ISBN: 978-1-337-10207-0

Cengage Learning
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with employees residing in nearly 40 different countries and sales in more than 125 countries around the world. Find your local representative at **www.cengage.com**.

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

To learn more about Cengage Learning Solutions, visit **www.cengage.com**

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
Print Number: 01   Print Year: 2018

Brief Contents

Contents

Preface

Programming Logic and Design, Comprehensive, Ninth Edition, provides the beginning programmer with a guide to developing structured program logic. This textbook assumes no programming language experience. The writing is nontechnical and emphasizes good programming practices. The examples are business examples; they do not assume mathematical background beyond high school business math.

Additionally, the examples illustrate one or two major points; they do not contain so many features that students become lost following irrelevant and extraneous details. The examples in this book have been created to provide students with a sound background in logic, no matter what programming languages they eventually use to write programs. This book can be used in a stand-alone logic course that students take as a prerequisite to a programming course, or as a companion book to an introductory programming text using any programming language.

Organization and Coverage

Programming Logic and Design, Comprehensive, Ninth Edition, introduces students to programming concepts and enforces good style and logical thinking. General programming concepts are introduced in Chapter 1.

Chapter 2 discusses using data and introduces two important concepts: modularization and creating high-quality programs. It is important to emphasize these topics early so that students start thinking in a modular way and concentrate on making their programs efficient, robust, easy to read, and easy to maintain.

Chapter 3 covers the key concepts of structure, including what structure is, how to recognize it, and most importantly, the advantages to writing structured programs. This chapter's content is unique among programming texts. The early overview of structure presented here provides students a solid foundation for thinking in a structured way.

Chapters 4, 5, and 6 explore the intricacies of decision making, looping, and array manipulation. Chapter 7 provides details of file handling so that students can create programs that process a significant amount of data.

In Chapters 8 and 9, students learn more advanced techniques in array manipulation and modularization. Chapters 10 and 11 provide a thorough, yet accessible, introduction to concepts and terminology used in object-oriented programming. Students learn about classes, objects, instance and static class members, constructors, destructors, inheritance, and the advantages of object-oriented thinking. Chapter 12 explores some additional object-oriented programming issues: event-driven GUI programming, multithreading, and animation.

Two appendices instruct students on working with numbering systems and providing structure for large programs.

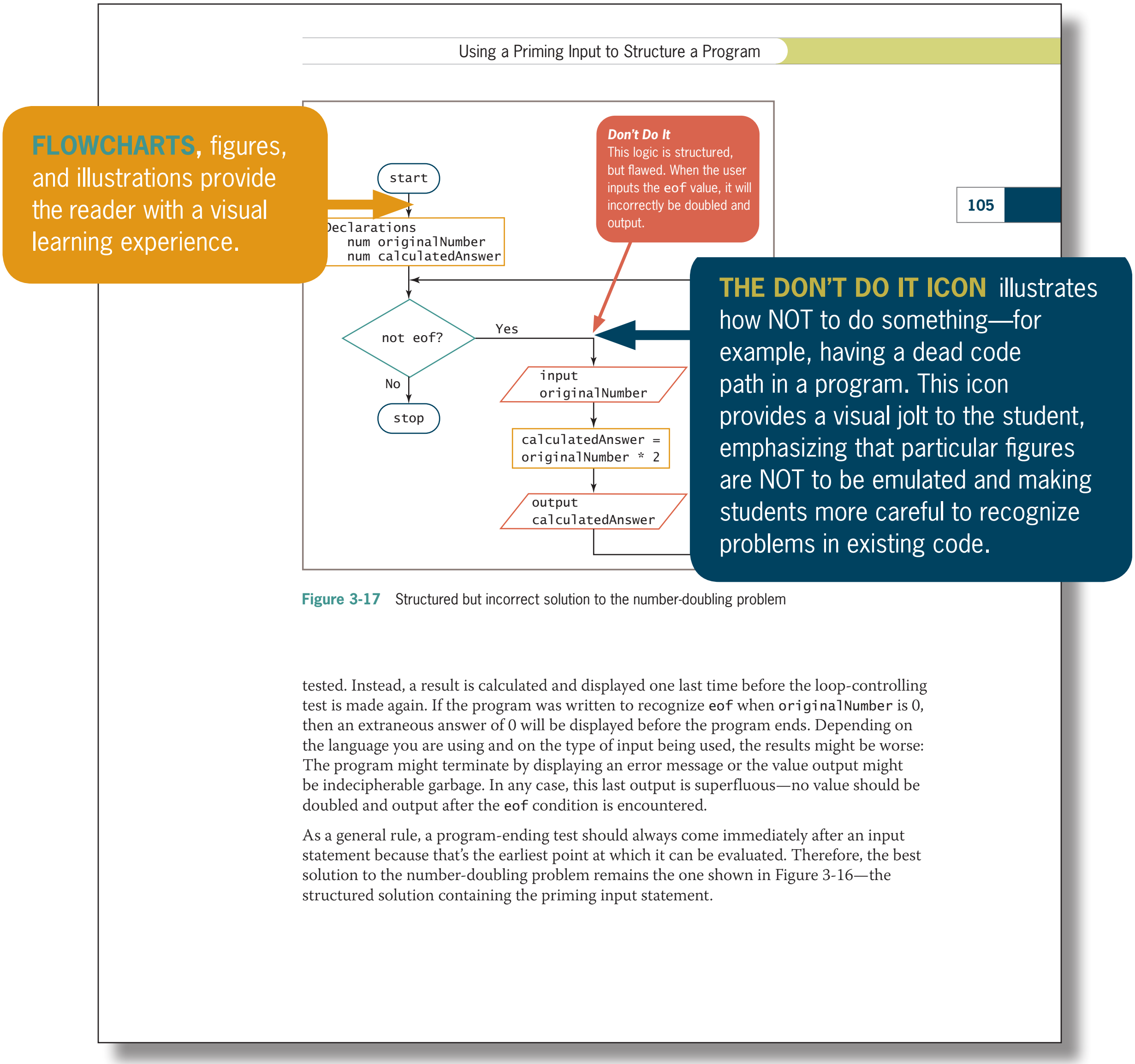
Programming Logic and Design combines text explanation with flowcharts and pseudocode examples to provide students with alternative means of expressing structured logic. Numerous detailed, full-program exercises at the end of each chapter illustrate the concepts explained within the chapter, and reinforce understanding and retention of the material presented.

Programming Logic and Design distinguishes itself from other programming logic books in the following ways:

- It is written and designed to be non-language specific. The logic used in this book can be applied to any programming language.
- The examples are everyday business examples: no special knowledge of mathematics, accounting, or other disciplines is assumed.
- The concept of structure is covered earlier than in many other texts. Students are exposed to structure naturally, so that they will automatically create properly designed programs.
- Text explanation is interspersed with both flowcharts and pseudocode so that students can become comfortable with these logic development tools and understand their interrelationship. Screen shots of running programs also are included, providing students with a clear and concrete image of the programs' execution.
- Complex programs are built through the use of complete business examples. Students see how an application is constructed from start to finish, instead of studying only segments of a program.

Features

This text focuses on helping students become better programmers, as well as helping them understand the big picture in program development through a variety of features. Each chapter begins with objectives and ends with a list of key terms and a summary; these useful features will help students organize their learning experience.



Using a Priming Input to Structure a Program

105

FLOWCHARTS, figures, and illustrations provide the reader with a visual learning experience.

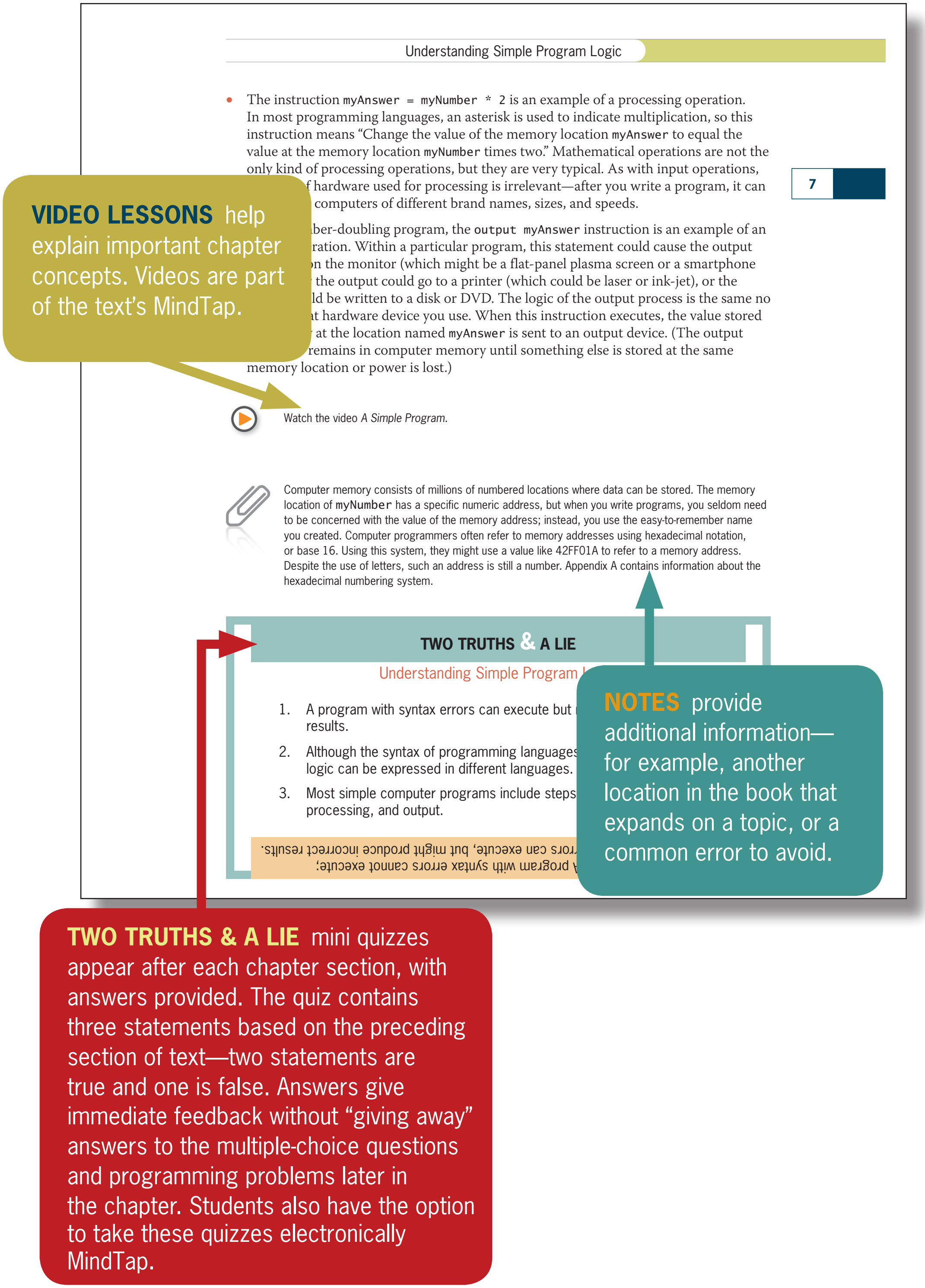
Don't Do It
This logic is structured, but flawed. When the user inputs the `eof` value, it will incorrectly be doubled and output.

THE DON'T DO IT ICON illustrates how NOT to do something—for example, having a dead code path in a program. This icon provides a visual jolt to the student, emphasizing that particular figures are NOT to be emulated and making students more careful to recognize problems in existing code.

Figure 3-17 Structured but incorrect solution to the number-doubling problem

tested. Instead, a result is calculated and displayed one last time before the loop-controlling test is made again. If the program was written to recognize `eof` when `originalNumber` is 0, then an extraneous answer of 0 will be displayed before the program ends. Depending on the language you are using and on the type of input being used, the results might be worse: The program might terminate by displaying an error message or the value output might be indecipherable garbage. In any case, this last output is superfluous—no value should be doubled and output after the `eof` condition is encountered.

As a general rule, a program-ending test should always come immediately after an input statement because that's the earliest point at which it can be evaluated. Therefore, the best solution to the number-doubling problem remains the one shown in Figure 3-16—the structured solution containing the priming input statement.



Understanding Simple Program Logic

7

- The instruction `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, f hardware used for processing is irrelevant—after you write a program, it can computers of different brand names, sizes, and speeds.

ber-doubling program, the `output myAnswer` instruction is an example of an ration. Within a particular program, this statement could cause the output on the monitor (which might be a flat-panel plasma screen or a smartphone the output could go to a printer (which could be laser or ink-jet), or the ld be written to a disk or DVD. The logic of the output process is the same no at hardware device you use. When this instruction executes, the value stored at the location named `myAnswer` is sent to an output device. (The output 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 number. Appendix A contains information about the hexadecimal numbering system.

TWO TRUTHS & A LIE

Understanding Simple Program L

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

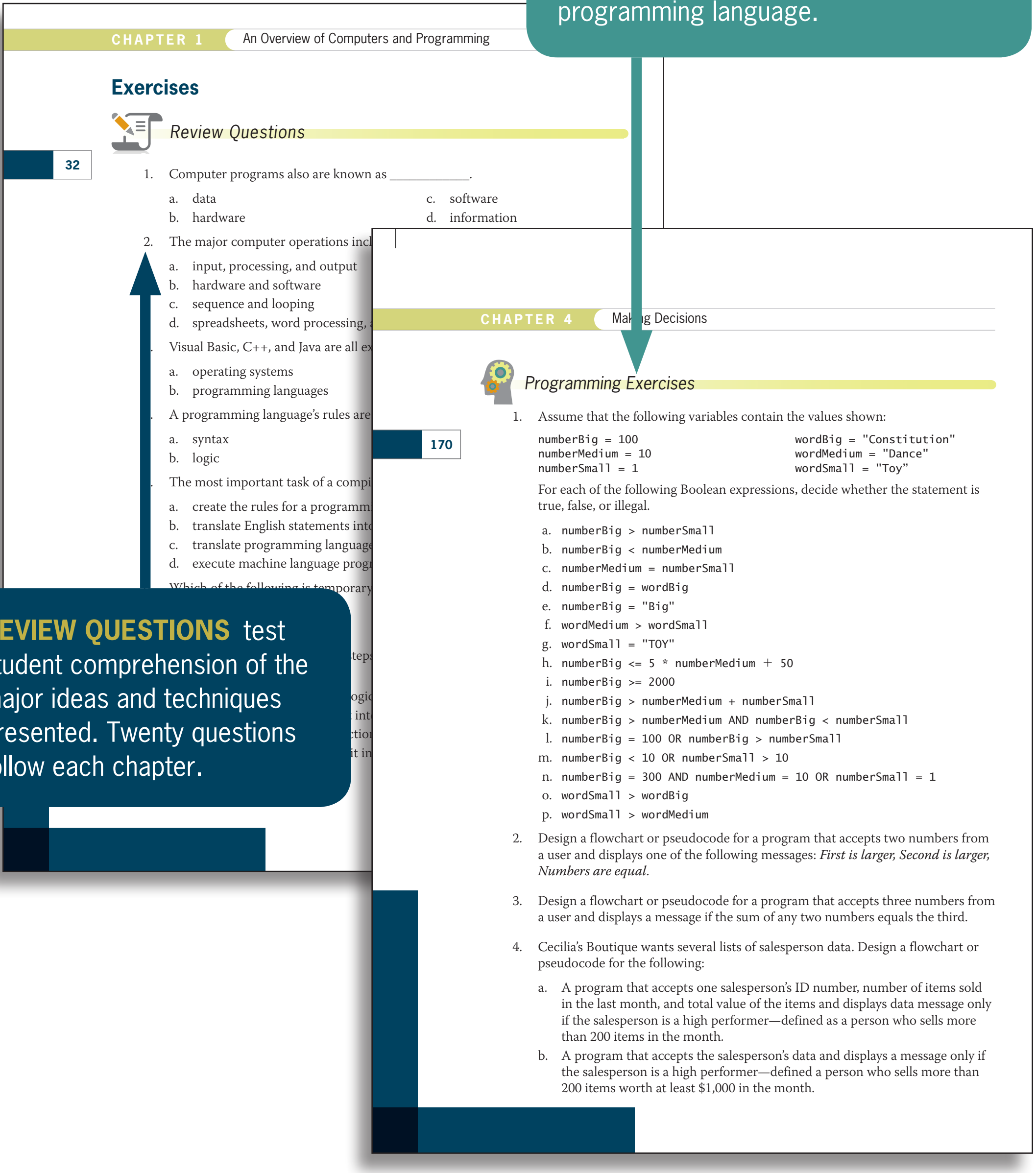
program with syntax errors cannot execute;
rors can execute, but might produce incorrect results.

VIDEO LESSONS help explain important chapter concepts. Videos are part of the text's MindTap.

NOTES provide additional information—for example, another location in the book that expands on a topic, or a common error to avoid.

TWO TRUTHS & A LIE mini quizzes appear after each chapter section, with answers provided. The quiz contains three statements based on the preceding section of text—two statements are true and one is false. Answers give immediate feedback without "giving away" answers to the multiple-choice questions and programming problems later in the chapter. Students also have the option to take these quizzes electronically MindTap.

Assessment



PROGRAMMING EXERCISES provide opportunities to practice chapter material. These exercises increase in difficulty and allow students to explore logical programming concepts. Most exercises can be completed using flowcharts, pseudocode, or both. In addition, instructors can assign the exercises as programming problems to be coded and executed in a particular programming language.

CHAPTER 1 An Overview of Computers and Programming

32

Exercises

Review Questions

1. Computer programs also are known as ____________.

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

2. The major computer operations incl

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

Visual Basic, C++, and Java are all ex

   a. operating systems
   b. programming languages

A programming language's rules are

   a. syntax
   b. logic

The most important task of a compi

   a. create the rules for a programm
   b. translate English statements int
   c. translate programming languag
   d. execute machine language progr

Which of the following is temporary

REVIEW QUESTIONS test student comprehension of the major ideas and techniques presented. Twenty questions follow each chapter.

CHAPTER 4 Making Decisions

170

Programming Exercises

1. Assume that the following variables contain the values shown:

```
numberBig = 100                     wordBig = "Constitution"
numberMedium = 10                   wordMedium = "Dance"
numberSmall = 1                     wordSmall = "Toy"
```

   For each of the following Boolean expressions, decide whether the statement is true, false, or illegal.

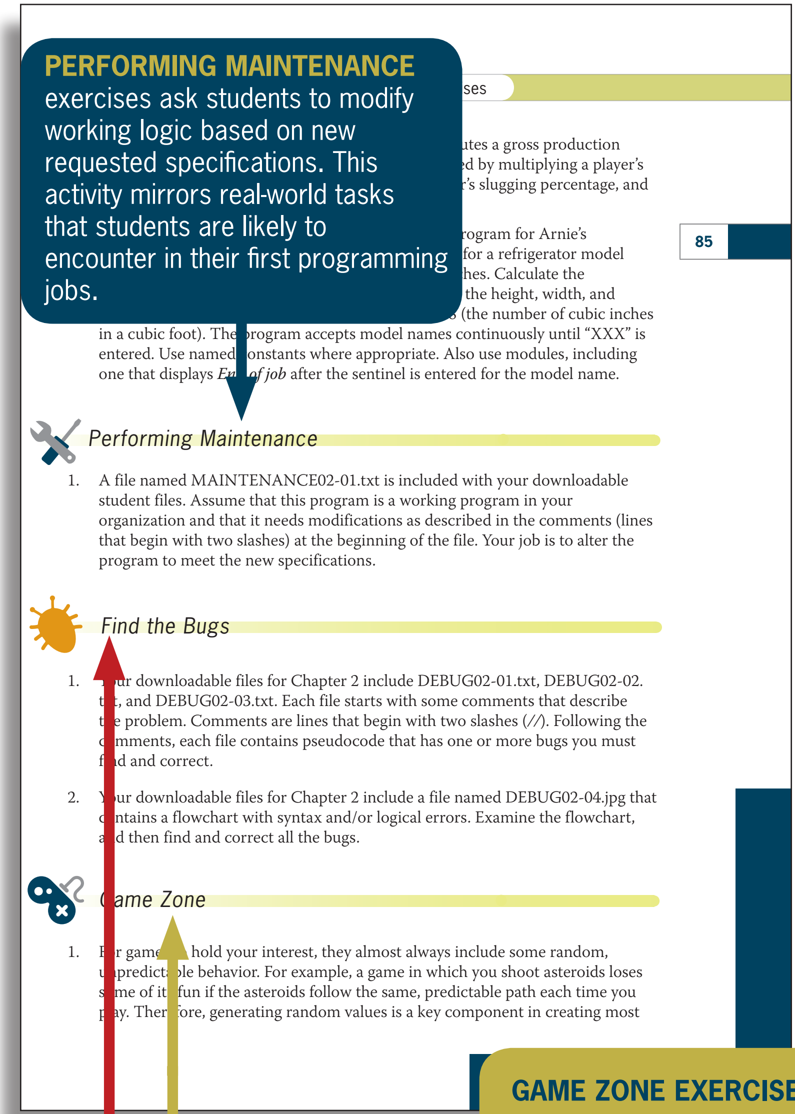
   a. numberBig > numberSmall
   b. numberBig < numberMedium
   c. numberMedium = numberSmall
   d. numberBig = wordBig
   e. numberBig = "Big"
   f. wordMedium > wordSmall
   g. wordSmall = "TOY"
   h. numberBig <= 5 * numberMedium + 50
   i. numberBig >= 2000
   j. numberBig > numberMedium + numberSmall
   k. numberBig > numberMedium AND numberBig < numberSmall
   l. numberBig = 100 OR numberBig > numberSmall
   m. numberBig < 10 OR numberSmall > 10
   n. numberBig = 300 AND numberMedium = 10 OR numberSmall = 1
   o. wordSmall > wordBig
   p. wordSmall > wordMedium

2. Design a flowchart or pseudocode for a program that accepts two numbers from a user and displays one of the following messages: *First is larger, Second is larger, Numbers are equal.*

3. Design a flowchart or pseudocode for a program that accepts three numbers from a user and displays a message if the sum of any two numbers equals the third.

4. Cecilia's Boutique wants several lists of salesperson data. Design a flowchart or pseudocode for the following:

   a. A program that accepts one salesperson's ID number, number of items sold in the last month, and total value of the items and displays data message only if the salesperson is a high performer—defined as a person who sells more than 200 items in the month.
   b. A program that accepts the salesperson's data and displays a message only if the salesperson is a high performer—defined a person who sells more than 200 items worth at least $1,000 in the month.



...utes a gross production ...ed by multiplying a player's ...r's slugging percentage, and

...rogram for Arnie's ...for a refrigerator model ...hes. Calculate the ...the height, width, and ...s (the number of cubic inches in a cubic foot). The program accepts model names continuously until "XXX" is entered. Use named constants where appropriate. Also use modules, including one that displays *End of job* after the sentinel is entered for the model name.

Performing Maintenance

1. A file named MAINTENANCE02-01.txt is included with your downloadable student files. Assume that this program is a working program in your organization and that it needs modifications as described in the comments (lines that begin with two slashes) at the beginning of the file. Your job is to alter the program to meet the new specifications.

Find the Bugs

1. Your downloadable files for Chapter 2 include DEBUG02-01.txt, DEBUG02-02.txt, and DEBUG02-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or more bugs you must find and correct.

2. Your downloadable files for Chapter 2 include a file named DEBUG02-04.jpg that contains a flowchart with syntax and/or logical errors. Examine the flowchart, and then find and correct all the bugs.

Game Zone

1. For games to hold your interest, they almost always include some random, unpredictable behavior. For example, a game in which you shoot asteroids loses some of its fun if the asteroids follow the same, predictable path each time you play. Therefore, generating random values is a key component in creating most

PERFORMING MAINTENANCE exercises ask students to modify working logic based on new requested specifications. This activity mirrors real-world tasks that students are likely to encounter in their first programming jobs.

GAME ZONE EXERCISES are included at the end of each chapter. Students can create games as an additional entertaining way to understand key programming concepts.

DEBUGGING EXERCISES are included with each chapter because examining programs critically and closely is a crucial programming skill. Students can download these exercises at *www.cengagebrain.com* and through MindTap. These files are also available to instructors through sso.cengage.com.

Other Features of the Text

This edition of the text includes many features to help students become better programmers and understand the big picture in program development.

- **Clear explanations**. The language and explanations in this book have been refined over eight editions, providing the clearest possible explanations of difficult concepts.
- **Emphasis on structure**. More than its competitors, this book emphasizes structure. Chapter 3 provides an early picture of the major concepts of structured programming.
- **Emphasis on modularity**. From the second chapter onwards, students are encouraged to write code in concise, easily manageable, and reusable modules. Instructors have found that modularization should be encouraged early to instill good habits and a clearer understanding of structure.
- **Objectives**. Each chapter begins with a list of objectives so that the student knows the topics that will be presented in the chapter. In addition to providing a quick reference to topics covered, this feature provides a useful study aid.
- **Chapter summaries**. Following each chapter is a summary that recaps the programming concepts and techniques covered in the chapter.
- **Key terms**. Each chapter lists key terms and their definitions; the list appears in the order that the terms are encountered in the chapter. A glossary at the end of the book lists all the key terms in alphabetical order, along with their working definitions.

MindTap

MindTap is a personalized learning experience with relevant assignments that guide students in analyzing problems, applying what they have learned, and improving their thinking. MindTap allows instructors to measure skills and outcomes with ease.

For instructors: Personalized teaching becomes yours with a learning path that is built with key student objectives. You can control what students see and when they see it. You can use MindTap as-is, or match it to your syllabus by hiding, rearranging, or adding content.

For students: A unique learning path of relevant readings, multimedia, and activities is created to guide you through basic knowledge and comprehension of analysis and application.

For both: Better outcomes empower instructors and motivate students with analytics and reports that provide a snapshot of class progress, the time spent in the course, engagement levels, and completion rates.

The MindTap for Programming Logic and Design includes coding labs in C++, Java, and Python, study tools, videos, and interactive quizzing, all integrated into an eReader that includes the full content of the printed text.

Instructor Resources

The following teaching tools are available to the instructor for download through our Instructor Companion Site at *sso.cengage.com*.

- **Instructor's Manual**. The Instructor's Manual follows the text chapter by chapter to assist in planning and organizing an effective, engaging course. The manual includes learning objectives, chapter overviews, lecture notes, ideas for classroom activities, and abundant additional resources. A sample course syllabus is also available.
- **PowerPoint Presentations**. This text provides PowerPoint slides to accompany each chapter. Slides are included to guide classroom presentations, and can be made available to students for chapter review, or to print as classroom handouts.
- **Solutions**. Solutions to review questions and exercises are provided to assist with grading.
- **Test Bank®**. Cengage Learning Testing Powered by Cognero is a flexible, online system that allows you to:
    - author, edit, and manage test bank content from multiple Cengage Learning solutions,
    - create multiple test versions in an instant, and
    - deliver tests from your LMS, your classroom, or anywhere you want.

Additional Options

- **Visual Logic™ software**. Visual Logic is a simple but powerful tool for teaching programming logic and design without traditional high-level programming language syntax. Visual Logic also interprets and executes flowcharts, providing students with immediate and accurate feedback.

Acknowledgments

I would like to thank all of the people who helped to make this book a reality, especially Alyssa Pratt, Jennifer Feltri-George, Kristin McNary, Kate Mason, and all the other professionals at Cengage Learning who made this book possible. Thanks, too, to my husband, Geoff, and our daughters, Andrea and Audrey, for their support. This book, as were all its previous editions, is dedicated to them.

–Joyce Farrell

CHAPTER 1

An Overview of Computers and Programming

Upon completion of this chapter, you will be able to:

- Describe computer systems
- Understand simple program logic
- List the steps involved in the program development cycle
- Write pseudocode statements and draw flowchart symbols
- Use a sentinel value to end a program
- Understand programming and user environments
- Describe 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 computers of varying sizes—for example, large mainframes, laptops, and very small devices embedded into products such as telephones, cars, and thermostats. The types of operations performed by different-sized computers, however, are very similar. Computer hardware needs instructions that control its operations, including 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 tells 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 games. When you hear people say they have "downloaded an **app** onto a mobile device," they are simply using an abbreviation of *application software*.
- **System software** comprises the programs that you use to manage your computer, including operating systems such as Windows, Linux, or UNIX for larger computers and Google Android and Apple iOS for smartphones.

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. **Data items** include all the text, numbers, and other raw material that are entered into and processed by a computer. Hardware devices that perform input operations include keyboards and mice. In business, many of the data items used are facts and figures about such entities as products, customers, and personnel. Data, however, also can include items such as images, sounds, and a user's mouse or finger-swiping 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**. Some devices, such as tablets and smartphones, usually contain multiple processors, and efficiently using several CPUs requires special programming techniques.

- **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 items that have been processed and output. Sometimes you place output on **storage devices**, such as your hard drive, flash media, or a cloud-based device. (The **cloud** refers to devices at remote locations accessed through the Internet.) 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.

Figure 1-1 shows how the statement that displays the word *Hello* on a single line on a computer monitor looks in some common programming languages. Notice that the syntax of some languages require that a statement start with an uppercase letter, while the syntax of others does not. Notice that some languages end statements with a semicolon, some with a period, and some with no ending punctuation at all. Also notice that different verbs are used to mean *display*, and that some are spelled like their like English word counterparts, while others like *cout* and *System.out.println* are not regular English words. The different formats you see are just a hint of the various syntaxes used by languages.

Language	Statement that displays Hello on a single line
Java	`System.out.println(“Hello);`
C++	`cout << “Hello” << endl;`
Visual Basic	`Console.WriteLine(“Hello”);`
Python	`print “Hello”`
COBOL	`DISPLAY “Hello”.`

Figure 1-1 Displaying the word *Hello* in some common programming languages

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 other languages.

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 running and data items that are 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—often your hard drive.

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 also is 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, some languages can use both compilers and interpreters.

After a program's source code is translated successfully 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 & 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

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 out any instructions, and you must not add extraneous instructions. A program with syntax errors cannot be translated fully 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.

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.

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
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—that is, hold different values at different points in time. 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 concerned only with the logic of operations, not the minor details.

A college classroom is similar to a named variable in that its name (perhaps 204 Adams Building) can hold different contents at different times. For example, your Logic class might meet there on Monday night, and a math class might meet there on Tuesday morning.

- The instruction `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 smartphone display), 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 number. Appendix A contains information about the hexadecimal 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 1-2 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.

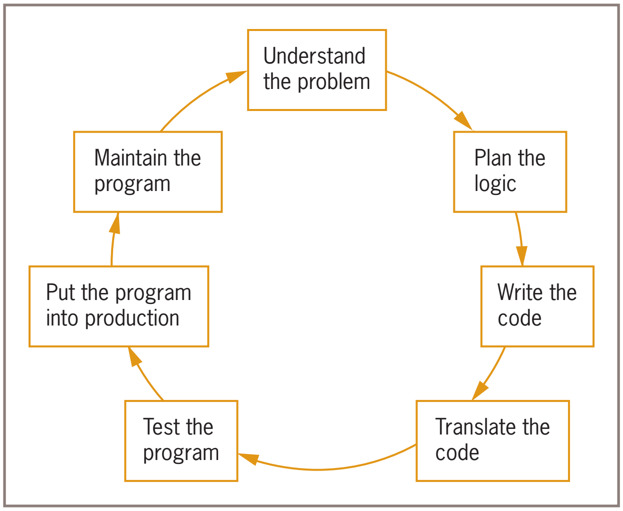


Figure 1-2 The program development cycle

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 website 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 often are 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.

Understanding the problem might be even more difficult if you are writing an app that you hope to market for mobile devices. Business developers usually are approached by a user with a need, but successful developers of mobile apps often try to identify needs that users aren't even aware of yet. For example, no one knew they wanted to play *Angry Birds* or leave messages on Facebook before those applications were developed. Mobile app developers also must consider a wider variety of user skills than programmers who develop applications that are used internally in a corporation. Mobile app developers must make sure their programs work with a range of screen sizes and hardware specifications because software competition is intense and the hardware changes quickly.