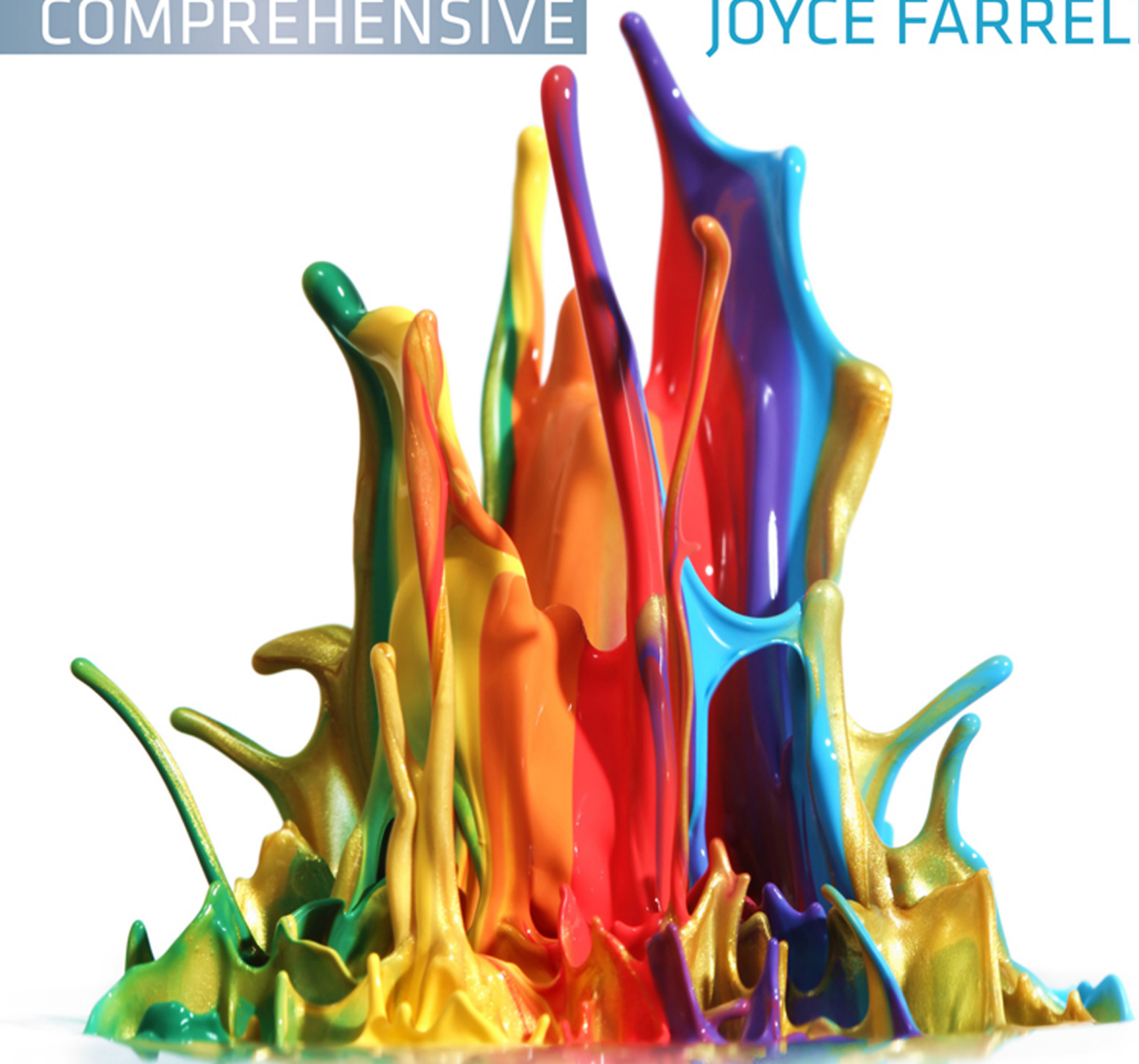
PROGRAMMING LOGIC AND DESIGN

COMPREHENSIVE

JOYCE FARRELL



SEVENTH EDITION

PROGRAMMING LOGIC AND DESIGN

COMPREHENSIVE VERSION

SEVENTH EDITION

PROGRAMMING LOGIC AND DESIGN

COMPREHENSIVE VERSION

JOYCE FARRELL

COURSE TECHNOLOGY
CENGAGE Learning®

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

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

Brief Contents

Contents

Preface

Programming Logic and Design, Comprehensive, Seventh 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, Seventh 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 gives students a solid foundation in 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 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 additional object-oriented programming issues: event-driven GUI programming, multithreading, and animation. Chapter 13 discusses system design issues and details the features of the Unified Modeling Language. Chapter 14 is a thorough introduction to important database concepts that business programmers should understand.

The first three appendices give students summaries of numbering systems, flowchart symbols, and structures. Additional appendices allow students to gain extra experience with structuring large unstructured programs, creating print charts, and understanding posttest loops and case structures.

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.

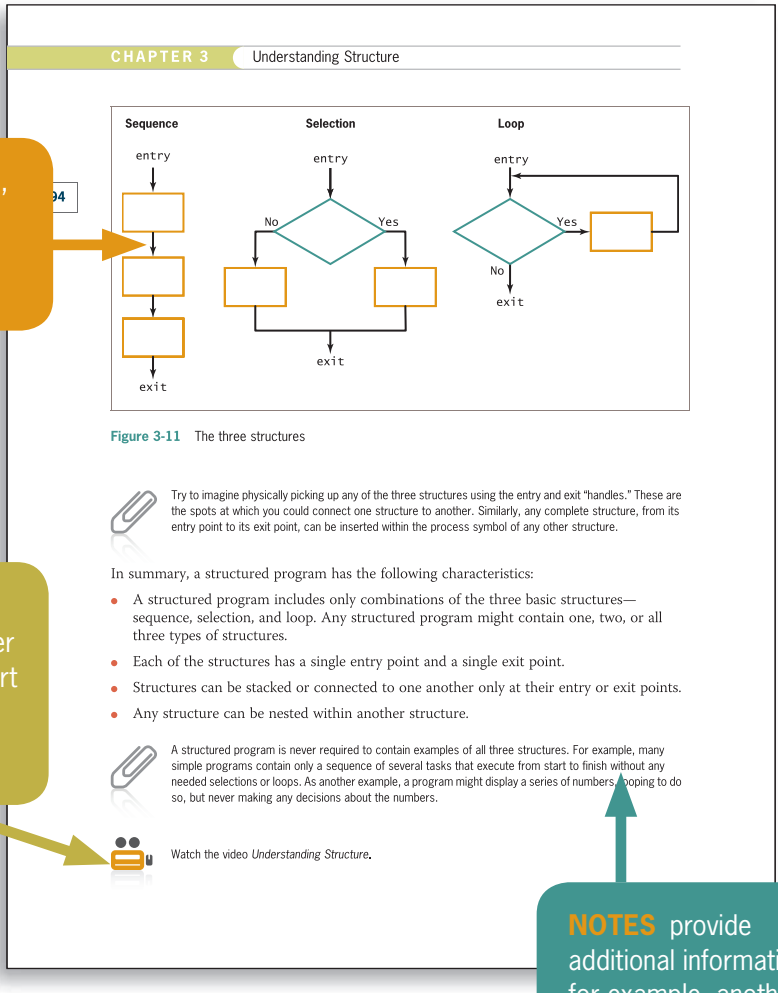
Programming Logic and Design distinguishes itself from other programming logic books in the following ways:

- 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 they will automatically create properly designed programs.
- Text explanation is interspersed with both flowcharts and pseudocode so 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 programs.

Features

This text focuses on helping students become better programmers and understand the big picture in program development through a variety of key features. In addition to chapter Objectives, Summaries, and Key Terms, these useful features will help students regardless of their learning style.

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



CHAPTER 3 Understanding Structure

Figure 3-11 The three structures

Try to imagine physically picking up any of the three structures using the entry and exit "handles." These are the spots at which you could connect one structure to another. Similarly, any complete structure, from its entry point to its exit point, can be inserted within the process symbol of any other structure.

In summary, a structured program has the following characteristics:

- A structured program includes only combinations of the three basic structures—sequence, selection, and loop. Any structured program might contain one, two, or all three types of structures.
- Each of the structures has a single entry point and a single exit point.
- Structures can be stacked or connected to one another only at their entry or exit points.
- Any structure can be nested within another structure.

A structured program is never required to contain examples of all three structures. For example, many simple programs contain only a sequence of several tasks that execute from start to finish without any needed selections or loops. As another example, a program might display a series of numbers, looping to do so, but never making any decisions about the numbers.

Watch the video *Understanding Structure*.

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

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

The use of flowcharts is excellent. This is a must-have book for learning programming logic before tackling the various languages.

—Lori Selby, University of Arkansas at Monticello

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 through the enhanced CourseMate site.

TWO TRUTHS & A LIE

Understanding the Three Basic Structures

1. Each structure in structured programming is a sequence, selection, or loop.
2. All logic problems can be solved using only three structures—sequence, selection, and loop.
3. The three structures cannot be combined in a single program.

The false statement is #3. The three structures can be stacked or nested in an infinite number of ways.

Using a Priming Input to Structure a Program

Recall the number-doubling program discussed in Chapter 2; Figure 3-12 shows a similar program. The program inputs a number and checks for the end-of-file condition. If the condition is not met, then the number is doubled, the answer is displayed, and the next number is input.

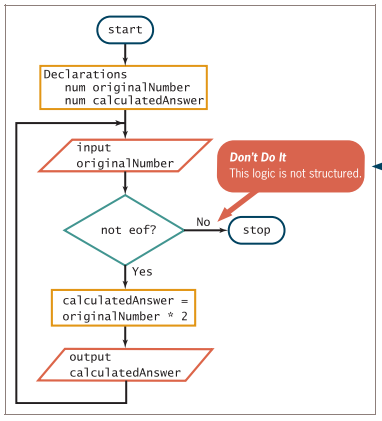


Figure 3-12 Unstructured flowchart of a number-doubling program

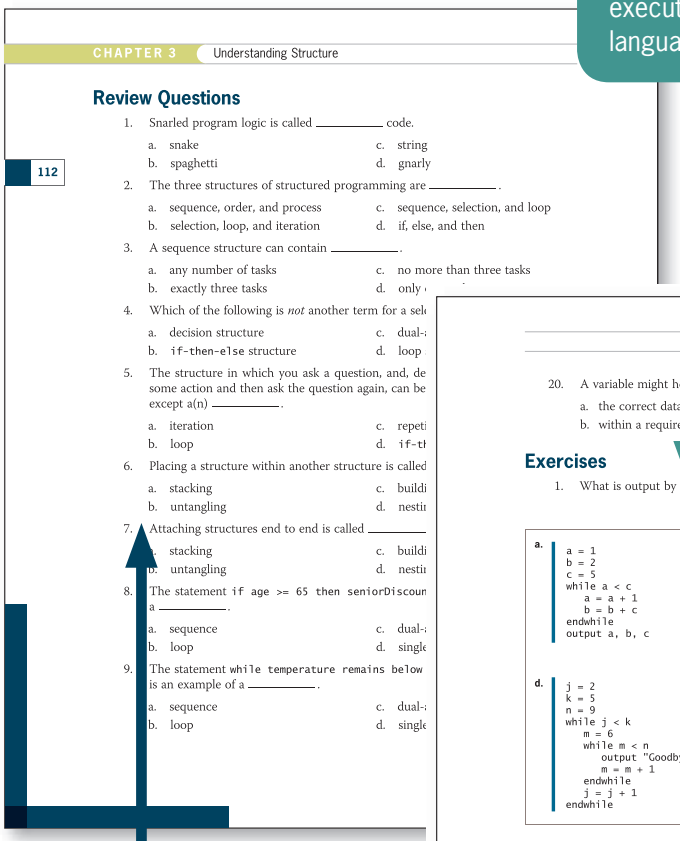
THE DON'T DO IT ICON illustrates how NOT to do something—for example, having a dead code path in a program. This icon provides a visual jolt to the student, emphasizing that particular figures are NOT to be emulated and making students more careful to recognize problems in existing code.

Assessment

The material is very well written, clearly presented, and up to date. All explanations are very solid, and Farrell's language is clean, cogent, and easy to follow.

—Judy Woodruff, Indiana University-Purdue University Indianapolis

EXERCISES provide opportunities to practice concepts. These exercises increase in difficulty and allow students to explore logical programming concepts. Each exercise 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.

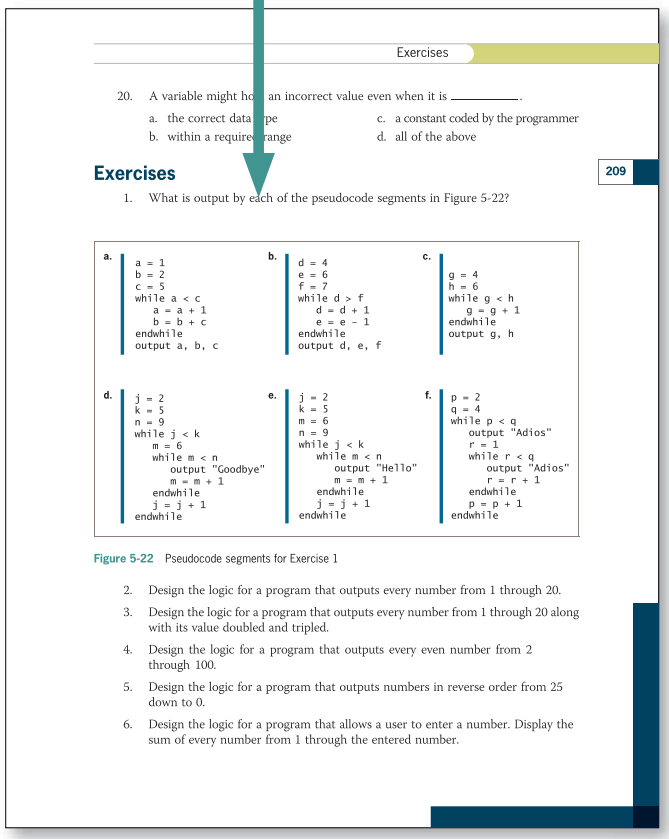


CHAPTER 3 Understanding Structure

112

Review Questions

1. Snarled program logic is called _____ code.
   a. snake c. string
   b. spaghetti d. gnarly
2. The three structures of structured programming are _____.
   a. sequence, order, and process c. sequence, selection, and loop
   b. selection, loop, and iteration d. if, else, and then
3. A sequence structure can contain _____.
   a. any number of tasks c. no more than three tasks
   b. exactly three tasks d. only
4. Which of the following is *not* another term for a sel
   a. decision structure c. dual-
   b. `if-then-else` structure d. loop
5. The structure in which you ask a question, and, de some action and then ask the question again, can be except a(n) _____.
   a. iteration c. repeti
   b. loop d. `if-th`
6. Placing a structure within another structure is called
   a. stacking c. buildi
   b. untangling d. nesti
7. Attaching structures end to end is called _____
   a. stacking c. buildi
   b. untangling d. nesti
8. The statement `if age >= 65 then seniorDiscoun` a _____.
   a. sequence c. dual-
   b. loop d. single
9. The statement `while temperature remains below` is an example of a _____.
   a. sequence c. dual-
   b. loop d. single



Exercises

20. A variable might ho an incorrect value even when it is _____.
    a. the correct data ype c. a constant coded by the programmer
    b. within a require range d. all of the above

Exercises

209

1. What is output by each of the pseudocode segments in Figure 5-22?

a.
```
a = 1
b = 2
c = 5
while a < c
   a = a + 1
   b = b + c
endwhile
output a, b, c
```

b.
```
d = 4
e = 6
f = 7
while d > f
   d = d + 1
   e = e - 1
endwhile
output d, e, f
```

c.
```
g = 4
h = 6
while g < h
   g = g + 1
endwhile
output g, h
```

d.
```
j = 2
k = 5
n = 9
while j < k
   m = 6
   while m < n
      output "Goodbye"
      m = m + 1
   endwhile
   j = j + 1
endwhile
```

e.
```
j = 2
k = 5
m = 6
n = 9
while j < k
   while m < n
      output "Hello"
      m = m + 1
   endwhile
   j = j + 1
endwhile
```

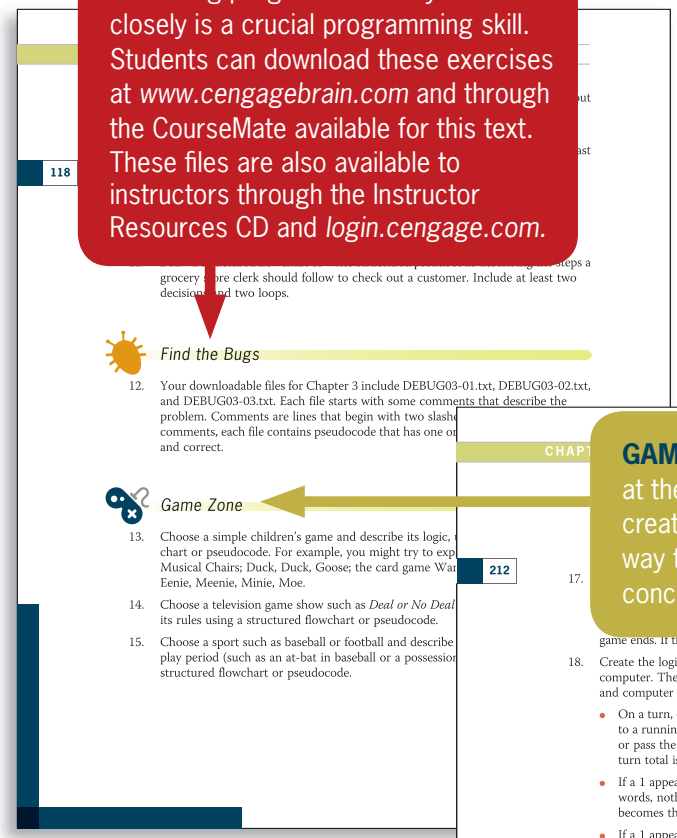
f.
```
p = 2
q = 4
while p < q
   output "Adios"
   r = 1
   while r < q
      output "Adios"
      r = r + 1
   endwhile
   p = p + 1
endwhile
```

Figure 5-22 Pseudocode segments for Exercise 1

2. Design the logic for a program that outputs every number from 1 through 20.
3. Design the logic for a program that outputs every number from 1 through 20 along with its value doubled and tripled.
4. Design the logic for a program that outputs every even number from 2 through 100.
5. Design the logic for a program that outputs numbers in reverse order from 25 down to 0.
6. Design the logic for a program that allows a user to enter a number. Display the sum of every number from 1 through the entered number.

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

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 the CourseMate available for this text. These files are also available to instructors through the Instructor Resources CD and *login.cengage.com*.



118

grocery store clerk should follow to check out a customer. Include at least two decision and two loops.

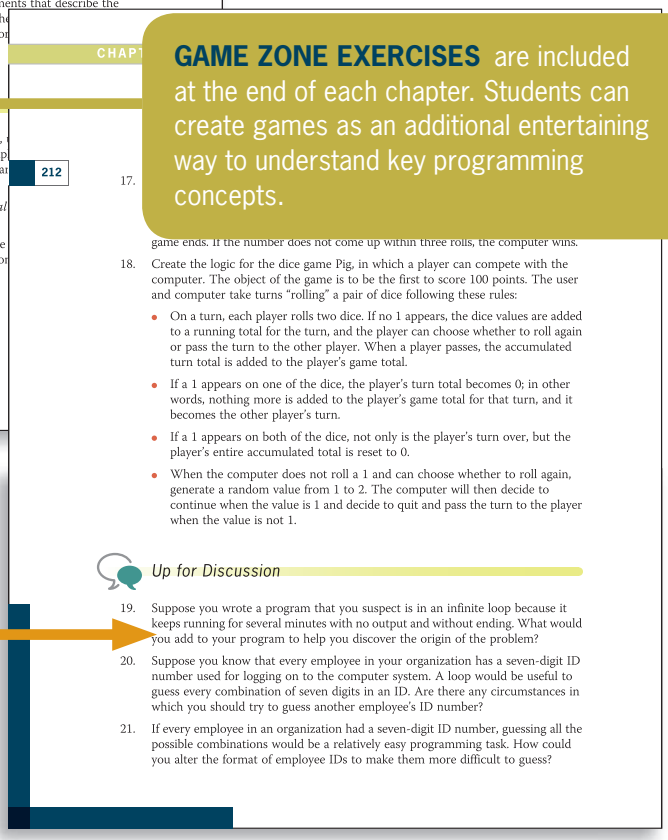
Find the Bugs

12. Your downloadable files for Chapter 3 include DEBUG03-01.txt, DEBUG03-02.txt, and DEBUG03-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slash comments, each file contains pseudocode that has one or and correct.

Game Zone

13. Choose a simple children's game and describe its logic, chart or pseudocode. For example, you might try to exp Musical Chairs; Duck, Duck, Goose; the card game Wat Eenie, Meenie, Minie, Moe.
14. Choose a television game show such as *Deal or No Deal* its rules using a structured flowchart or pseudocode.
15. Choose a sport such as baseball or football and describe play period (such as an at-bat in baseball or a possessio structured flowchart or pseudocode.

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.



CHAPT

212

17.

game ends. If the number does not come up within three rolls, the computer wins.

18. Create the logic for the dice game Pig, in which a player can compete with the computer. The object of the game is to be the first to score 100 points. The user and computer take turns "rolling" a pair of dice following these rules:
    - On a turn, each player rolls two dice. If no 1 appears, the dice values are added to a running total for the turn, and the player can choose whether to roll again or pass the turn to the other player. When a player passes, the accumulated turn total is added to the player's game total.
    - If a 1 appears on one of the dice, the player's turn total becomes 0; in other words, nothing more is added to the player's game total for that turn, and it becomes the other player's turn.
    - If a 1 appears on both of the dice, not only is the player's turn over, but the player's entire accumulated total is reset to 0.
    - When the computer does not roll a 1 and can choose whether to roll again, generate a random value from 1 to 2. The computer will then decide to continue when the value is 1 and decide to quit and pass the turn to the player when the value is not 1.

Up for Discussion

19. Suppose you wrote a program that you suspect is in an infinite loop because it keeps running for several minutes with no output and without ending. What would you add to your program to help you discover the origin of the problem?
20. Suppose you know that every employee in your organization has a seven-digit ID number used for logging on to the computer system. A loop would be useful to guess every combination of seven digits in an ID. Are there any circumstances in which you should try to guess another employee's ID number?
21. If every employee in an organization had a seven-digit ID number, guessing all the possible combinations would be a relatively easy programming task. How could you alter the format of employee IDs to make them more difficult to guess?

ESSAY QUESTIONS present personal and ethical issues that programmers must consider. These questions can be used for written assignments or as a starting point for classroom discussion.

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. In this edition, all explanations have been carefully reviewed to provide the clearest possible instruction. Material that previously was included in margin notes has most frequently been incorporated into the main text, giving the pages a more streamlined appearance. All the chapters in this edition contain new programming exercises. All Sixth Edition exercises that have been replaced are available on the Instructor Resources CD and through *login.cengage.com* so instructors can use them as additional assigned exercises or as topics for class discussions.

- **Clear explanations**. The language and explanations in this book have been refined over seven 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, giving students an overview of the principles before they are required to consider program details.
- **Emphasis on modularity**. From the second chapter, 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. This edition uses modularization early, using global variables instead of local passed and returned values, and saves parameter passing for later when the student has become more adept.
- **Methods as black boxes**. The use of methods is consistent with the languages in which the student is likely to have first programming experiences. In particular, this book emphasizes using methods as black boxes, declaring all variables and constants as local to methods, and passing arguments to and receiving returned values from methods as needed.
- **Objectives**. Each chapter begins with a list of objectives so 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.
- **Pseudocode**. This book includes numerous examples of pseudocode, which illustrate correct usage of the programming logic and design concepts being taught.
- **Chapter summaries**. Following each chapter is a summary that recaps the programming concepts and techniques covered in the chapter. This feature provides a concise means for students to review and check their understanding of the main points in each chapter.
- **Key terms**. Each chapter lists key terms and their definitions; the list appears in the order the terms are encountered in the chapter. Along with the chapter summary, the

list of key terms provides a snapshot overview of a chapter's main ideas. A glossary at the end of the book lists all the key terms in alphabetical order, along with working definitions.

CourseMate

The more you study, the better the results. Make the most of your study time by accessing everything you need to succeed in one place. Read your textbook, review flashcards, watch videos, and take practice quizzes online. CourseMate goes beyond the book to deliver what you need! Learn more at *www.cengage.com/coursemate*.

The *Programming Logic and Design* CourseMate includes:

- **Video Lessons**. Designed and narrated by the author, videos in each chapter explain and enrich important concepts.
- **Two Truths & A Lie** and **Debugging Exercises**. Complete popular exercises from the text online!
- An interactive eBook with highlighting and note-taking, flashcards, quizzing, study games, and more!

Instructors may add CourseMate to the textbook package, or students may purchase CourseMate directly at *www.cengagebrain.com*.

Instructor Resources

The following teaching tools are available to the instructor on a single CD-ROM. Many are also available for download through our Instructor Companion Site at *login.cengage.com*.

- **Electronic 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 presentation, to make available to students for chapter review, or to print as classroom handouts. Instructors may customize the slides, which include the complete figure files from the text, to best suit their courses.
- **Solutions**. Solutions to review questions and exercises are provided to assist with grading.

- **ExamView®**. This textbook is accompanied by ExamView, a powerful testing software package that allows instructors to create and administer printed, LAN-based, and Internet exams. ExamView includes hundreds of questions that correspond to the text, enabling students to generate detailed study guides that include page references for further review. The computer-based and Internet testing components allow students to take exams at their computers, and the components save the instructor time by grading each exam automatically. These test banks are also available in Blackboard and Angel compatible formats.

Additional Offerings

You have the option to bundle software with your text. Please contact your Cengage Learning sales representative for more information.

- **PAL Guides**. Together with *Programming Logic and Design*, these brief books, or PAL guides, provide an excellent opportunity to learn the fundamentals of programming while gaining exposure to a programming language. Readers will discover how real code behaves within the context of the traditionally language-independent logic and design course. PAL guides are available for C++, Java, and Visual Basic; please contact your sales rep for more information on how to add the PAL guides to your course.
- **Microsoft® Office Visio® Professional 2010**, 60-day version. Visio 2010 is a diagramming program that allows users to create flowcharts and diagrams easily while working through the text, enabling them to visualize concepts and learn more effectively.
- **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 uses flowcharts to explain the essential programming concepts discussed in this book, including variables, input, assignment, output, conditions, loops, procedures, graphics, arrays, and files. Visual Logic also interprets and executes flowcharts, providing students with immediate and accurate feedback. Visual Logic combines the power of a high-level language with the ease and simplicity of flowcharts.

Acknowledgments

I would like to thank all of the people who helped to make this book a reality, especially Dan Seiter, Development Editor. After seven editions, Dan still finds ways to improve my explanations so that we can create a book of the highest possible quality. Thanks also to Alyssa Pratt, Senior Product Manager; Brandi Shailer, Acquisitions Editor; Catherine DiMassa, Senior Content Project Manager; and Green Pen QA, Technical Editors. I am grateful to be able to work with so many fine people who are dedicated to producing quality instructional materials.

I am indebted to the many reviewers who provided helpful and insightful comments during the development of this book, including Linda Cohen, Forsyth Tech; Andrew Hurd, Hudson Valley Community College; George Reynolds, Strayer University; Lori Selby, University of Arkansas at Monticello; and Judy Woodruff, Indiana University–Purdue University Indianapolis.

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

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
```

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.

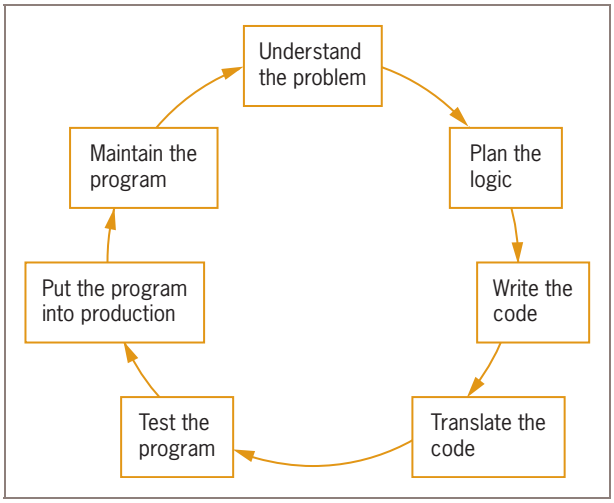


Figure 1-1 The program development cycle

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