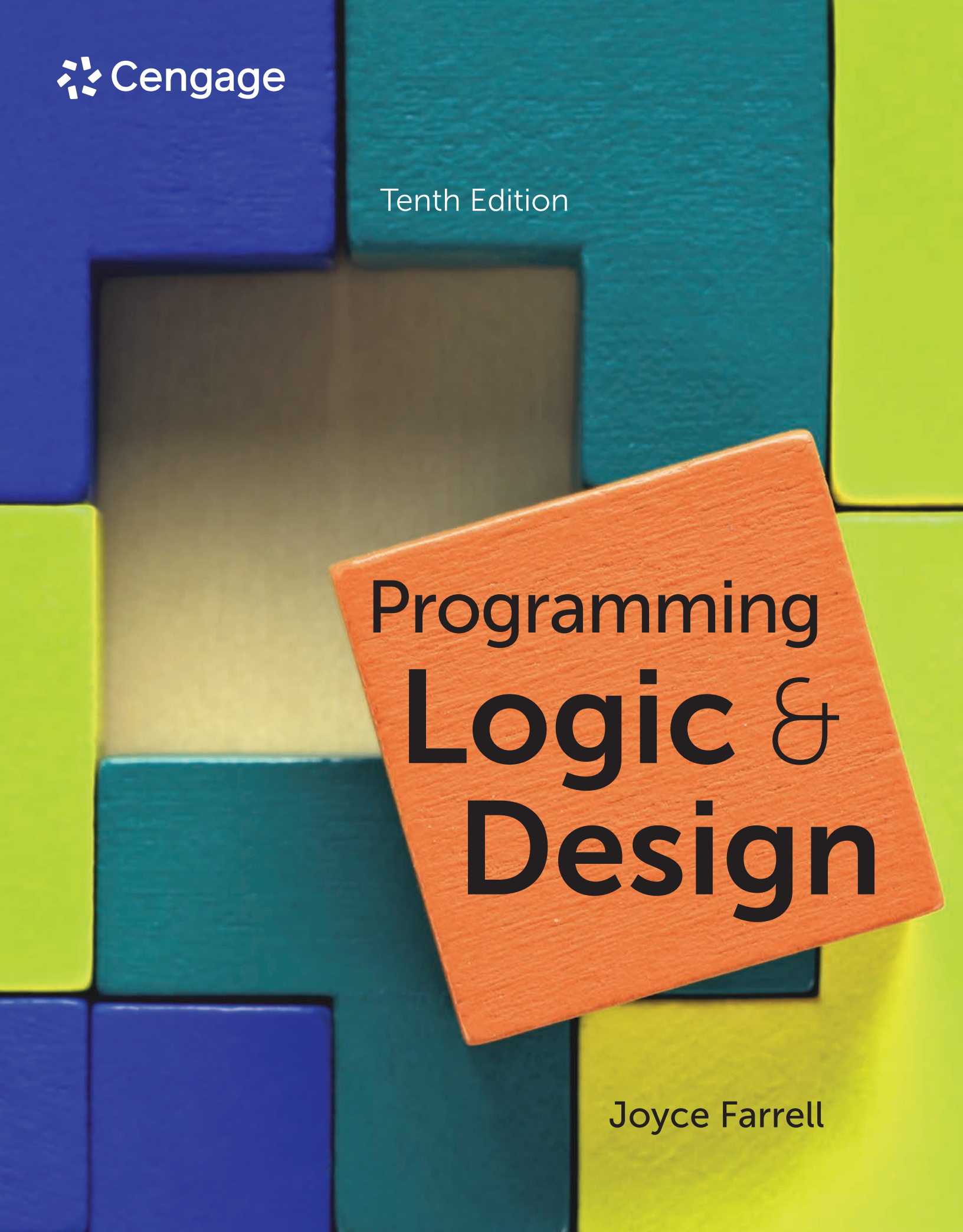

Cengage
Tenth Edition
Programming
Logic &
Design
Joyce Farrell

Programming Logic and Design

Tenth Edition

Joyce Farrell

Cengage

Australia • Brazil • Canada • Mexico • Singapore • United Kingdom • United States

Programming Logic and Design, Tenth Edition
Joyce Farrell

SVP, Higher Education Product Management: Erin Joyner

VP, Product Management and Marketing, Learning Experiences: Thais Alencar

Portfolio Product Director: Mark Santee

Portfolio Product Manager: Tran Pham

Product Assistant: Ethan Wheel

Learning Designer: Mary Convertino

Senior Content Manager: Isabelle Berthet

Senior Content Manager: Michelle Ruelos Cannistraci

Content Manager: Katherine Russillo

Associate Digital Project Manager: John Smigielski

Technical Editor: Danielle Shaw

Developmental Editor: Dan Seiter

VP, Product Marketing: Jason Sakos

Director, Product Marketing: April Danaë

Portfolio Marketing Manager: Mackenzie Paine

Senior Subject Matter Assurance Engineer: Troy Dundas

IP Analyst: Ann Hoffman

IP Project Manager: Lumina Datamatics

Production Service: Straive

Senior Designer: Erin Griffin

Cover Image Source: Armagadon/Shutterstock.com

Library of Congress Control Number: 2022922035

ISBN: 978-0-357-88087-6

Cengage
200 Pier 4 Boulevard
Boston, MA 02210
USA

Cengage 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**.

To learn more about Cengage platforms and services, register or access your online learning solution, or purchase materials for your course, visit **www.cengage.com**

Printed in the United States of America
Print Number: 01 Print Year: 2024

Brief Contents

Contents

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Chapter 8

Chapter 9

Chapter 10

Chapter 11

Chapter 12

Appendix A

Appendix B

About the Author

Joyce Farrell has taught computer programming at McHenry County College, Crystal Lake, Illinois; the University of Wisconsin, Stevens Point, Wisconsin; and Harper College, Palatine, Illinois. Besides *Programming Logic and Design*, she has written books on Java, C#, and C++ for Cengage.

Preface for the Instructor

Programming Logic and Design, Tenth Edition provides the beginning programmer with a guide to developing structured program logic. This textbook assumes no programming language experience. The writing is non-technical 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 text using any programming language.

New to This Edition

Multiple improvements have been added to this edition of *Programming Logic and Design*. These include the following:

- The chapter objectives have been modified where necessary to use Bloom's taxonomy.
- The objectives, major section headings, and end-of-chapter exercises have been numbered to show how they correspond.
- Colors have been modified to make diagrams more accessible.
- The text has been modified to ensure clarity, currency, and inclusiveness.
- Code samples and figures have been edited to create consistency in spacing.
- Care has been taken to use the more generic term *modules* for program segments until object-oriented programming is discussed. After that, the term *methods* is used because that is the commonly used name for modules in object-oriented languages.
- The discussion of constructors has been expanded and uses the `new` operator, which is common in object-oriented languages. The `new` operator is also used for constructing `Exception` objects.

Inclusivity and Diversity

Cengage is committed to providing educational content that is inclusive and welcoming to all learners. Research demonstrates that students who experience a sense of belonging in class more successfully make meaning out of, and find relevance in, what they encounter in learning content. To improve both the learning process and outcomes, our materials seek to affirm the fullness of human diversity with respect to ability, language, culture, gender, age, socioeconomics, and other forms of human difference that students may bring to the classroom.

Across the computing industry, standard language such as *master* and *slave* is being retired in favor of language that is more inclusive, such as *primary/replica* or *leader/follower*. Different software development and web media companies are adopting their own replacement language, and currently there is no shared standard.

In addition, the terms *master* and *slave* remain deeply embedded in legacy code, and understanding this terminology remains necessary for new programmers. When required for understanding, Cengage will introduce the noninclusive term in the first instance but will then provide an appropriate replacement term for the remainder of the discussion or example. We appreciate your feedback as we work to make our products more inclusive for all.

For more information about Cengage's commitment to inclusivity and diversity, please visit *www.cengage.com/inclusion-diversity/*.

Organization of the Text

Programming Logic and Design, Tenth 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 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, Tenth Edition 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, Tenth Edition 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 flowcharts and pseudocode so that students can become comfortable with these logic development tools and understand their interrelationship. Screenshots 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 of the Text

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 prepare for and organize their learning experience. Review Questions, Programming Exercises, Performing Maintenance, and Game Zone exercises are aligned with the objectives listed at the beginning of each chapter. This alignment is clearly indicated in parentheses at the end of each question. Debugging Exercises, while internally aligned with the chapter objectives, do not display this alignment so as not to reveal the error in the program sample.

Note These notes provide additional information—for example, a common error to watch out for or background information on a topic.

Two Truths & a Lie

These quizzes appear after each chapter section, with answers provided. Each 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 in MindTap.

Don't Do It

These icons illustrate how *not* to do something—for example, having a dead code path in a program. They provide a visual jolt to the student, emphasizing that particular practices are *not* to be emulated and making students more careful to recognize problems in existing code.

Assessment

Review Questions

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

Programming Exercises

Programming exercises provide opportunities to practice concepts. These exercises allow students to explore each major programming concept presented in the chapter. Additional coding labs and samples are available in MindTap.

Performing Maintenance

These 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.

Debugging Exercises

Debugging exercises are included with each chapter because examining programs critically and closely is a crucial programming skill. Students and instructors can download these exercises at *www.cengage.com*.

Game Zone Exercises

These exercises are included at the end of each chapter. Students can create the logic for games as an additional entertaining way to understand key programming concepts.

Additional 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.

- **Emphasis on modularity.** 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.
- **Emphasis on structure.** More than its competitors, this book emphasizes structure. Chapter 3 provides an early picture of the major concepts of structured programming.
- **Flowcharts, figures, and illustrations.** These graphics provide the reader with a visual learning experience.
- **Clear explanations.** The language and explanations in this book have been refined over nine previous editions, providing the clearest possible explanations of difficult concepts.
- **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.
- **Summaries.** Following each chapter is a summary that recaps the concepts and techniques covered in the chapter.
- **Key terms.** Each chapter identifies key terms and provides a list of the terms at the end of the chapter. A glossary at the end of the book lists all the key terms in alphabetical order, along with their working definitions.

Course Solutions

Online Learning Platform: MindTap

Today's leading online learning platform, MindTap for *Programming Logic and Design, Tenth Edition* gives you complete control of your course to craft a personalized, engaging learning experience that challenges students, builds confidence, and elevates performance.

MindTap introduces students to core concepts from the beginning of your course using a simplified learning path that progresses from understanding to application and delivers access to eBooks, study tools, interactive media, auto-graded assessments, and performance analytics.

Use MindTap for *Programming Logic and Design, Tenth Edition* as is, or personalize it to meet your specific course needs. You can also easily integrate MindTap into your Learning Management System (LMS).

The MindTap course for *Programming Logic and Design, Tenth Edition* 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. In addition to the readings, the *Programming Logic and Design, Tenth Edition* MindTap course includes the following:

- **Coding labs.** These supplemental assignments provide real-world application and encourage students to practice new programming logic and design concepts in a complete online IDE in any one of three programming languages: Java, Python, or C++. New and improved Guided Feedback provides personalized and immediate feedback to students as they proceed through their coding assignments so that they can understand and correct errors in their code.
- **Gradeable assessments and activities.** All assessments and activities from the readings will be available as gradeable assignments within MindTap, including Review Questions, Performing Maintenance, Debugging Exercises, Game Zone, and Two Truths & a Lie.
- **Programming and Learning Guides.** Supplemental Programming and Learning Guides (PAL Guides) are provided as downloads in the learning path for those interested in applying the programming learning and design concepts to any of the three major programming languages: Java, Python, and C++. The guides have been carefully developed to extend learning from conceptual understanding to application through exercises and labs. The guides follow along chapter by chapter with the core narrative of the textbook.
- **Interactive study aids.** Flashcards and PowerPoint lectures help users review main concepts from the units.

Ancillary Package

Additional instructor resources for this product are available online. Instructor assets include an Instructor Manual, Educator's Guide, PowerPoint® slides, and a test bank powered by Cognero®. Sign up or sign in at *www.cengage.com* to search for and access this product and its online resources.

- **Instructor Manual.** The Instructor 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 additional resources.
- **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.
- **Solution and Answer Guide (SAG).** Solutions and rationales to review questions and exercises are provided to assist with grading and student understanding.
- **Solutions.** Solutions to all programming exercises are available. If an input file is needed to understand a programming exercise, it is included with the solution file.
- **Data files.** Data files necessary to complete some of the steps and projects in the course are available. The Data Files folder also includes Java, Python, and C++ files for every program that appears in the Programming and Learning (PAL) Guide.
- **Educator's Guide.** The Educator's Guide contains a detailed outline of the corresponding MindTap course.
- **Transition Guide.** The Transition Guide outlines information on what has changed from the Ninth Edition.

- **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.
  - Deliver tests from your LMS, your classroom, or anywhere you want.

Acknowledgments

I would like to thank all of the people who helped to make this book a reality, especially Tran Pham, Isabelle Berthet, Katherine Russillo, Michelle Ruelos Cannistraci, Troy Dundas, Mary Convertino, and all the other professionals at Cengage who made this book possible. Thank you to Technical Editor Danielle Shaw, who ensures that all technical content in the narrative and accompanying data files is accurate and error free. Thank you to my Development Editor, Dan Seiter, who is the most thorough, competent editor I have ever met, and who made this a better book. Thank you to Keith Morneau of ECPI University and Dwight Watt of Georgia Northwestern Technical College for their reviews of the manuscript. 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.

Chapter 1

An Overview of Computers and Programming

Learning Objectives

When you complete this chapter, you will be able to:

- **1.1** Describe the components of computer systems
- **1.2** Develop simple program logic
- **1.3** List the steps involved in the program development cycle
- **1.4** Write pseudocode statements and draw flowchart symbols
- **1.5** Use a sentinel value to end a program
- **1.6** Describe programming and user environments
- **1.7** Describe the evolution of programming models

1.1 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 phones, 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 tell the hardware what to do. Software is programs. A **program** is a set of computer instructions written by a programmer. You can buy prewritten programs that are stored on a CD or DVD 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 course focuses on developing program logic. The **logic** of a computer program is the complete sequence of tasks that lead to a problem's solution. A program's logic is similar to a story's plot—it is a series of steps that lead to a conclusion.

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**," they are simply using an abbreviation of *application software*.
- **System software** comprises the programs that you use to manage your computer, including operating systems. An **operating system** is the software that supports a computer's basic functions, including controlling devices such as keyboards and mice and scheduling tasks. Some operating systems you might know include Windows, Linux, or UNIX for larger computers and Google Android and Apple iOS for smartphones.

This course focuses on the logic used to write application software programs rather than system software programs, but 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** include all the text, numbers, and other raw material that are entered into and processed by a computer. 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. 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 means working with them—for example, 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 (CPU)**. Some devices, such as tablets and smartphones, usually contain multiple processors, including those that specialize in processing graphics; efficiently using several CPUs requires special programming techniques.
- **Output**: Programming professionals often use the term *data* for input items, but they use the term **information** for data items that have been processed and 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. Sometimes you place output on a **storage device**, 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 or interpret 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 set of instructions you write using a programming language is 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.

Table 1-1 shows how you can use several common programming languages to write a statement that displays the word *Hello* on a single line on a computer monitor. Notice that the syntax of some languages requires 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 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.

Table 1-1 Displaying the word *Hello* in some common programming 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".`

Note

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 (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; it's because your system was configured to automatically save your work at regular intervals on a nonvolatile storage device—often your hard drive or a remote drive on the Web.

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.

Note

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.

Note

Besides the popular, comprehensive programming languages such as Java and C++, many programmers use a **scripting language** (also called a *scripting programming language* or *script language*) such as Python, Lua, Perl, or PHP. Scripts written in a scripting language usually can be typed directly from a keyboard and be 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, and 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.

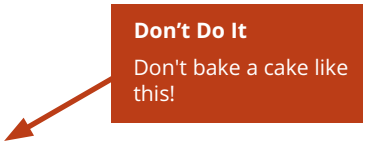
The false statement is #3. You write programs using a programming language such as Visual Basic or Java, and a translation program (called a compiler or an interpreter) converts the statements to machine language, which is 0s and 1s.

1.2 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
```



Note

The dangerous cake-baking instructions are shown with a Don't Do It icon. You will see this icon when the sample instructions contain 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 course (as well as in almost all programming languages), variable names will not contain embedded spaces; for example, a variable name might be `myNumber` instead of `my Number`.

Note

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. This course is concerned only with the logic of operations, not the minor details.

Note

A college classroom is similar to a named variable in that its name (for example, *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), the output could go to a printer (which could be laser or ink-jet), the output could be written to a disk or DVD, or the output could be spoken words sent to headphones. 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.)

Note

Computer memory consists of millions of numbered locations where data can be stored. The memory location of `myNumber` has a specific numeric address each time you run the program that contains it, 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.

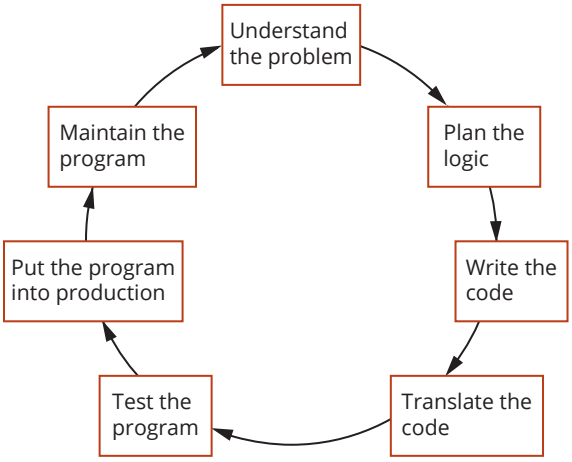
The false statement is #1. A program with syntax errors cannot execute; a program with no syntax errors can execute, but might produce incorrect results.

1.3 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 an 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 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 the director 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 specific online games or post photos to social media before such 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.

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!

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. Two common planning tools are flowcharts and pseudocode. You will learn more about these tools later in this chapter and you will work with many examples of flowcharts and pseudocode throughout this course. 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 or rules you follow to solve a problem.

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 a program's 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 course; in fact, the course focuses on this crucial step almost exclusively.

Coding the Program

The programmer can write the source code for a program only after the logic is developed. 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!)

Note

In some organizations, systems analysts, programmers, and coders have slightly different jobs. An analyst might talk with users to determine their needs, a programmer might design the logical steps in a program, and a coder might then write the designed instructions in a programming language. However, there is overlap in these jobs, and one employee might fill all three roles in some organizations.

Using Software to Translate the Program into Machine Language

Even though there are many programming languages, each computer works internally with 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 state or the off state, which are represented by a 1 or 0, respectively.

Languages such as Java or Visual Basic are available for programmers because someone has written a translator program (a compiler or an interpreter) that changes the programmer's English-like **high-level programming language** into machine language that the computer understands. (Machine language is an example of a **low-level programming language**; some other languages not far removed from machine language are also referred to as low-level languages.) When you learn the syntax of a programming language, the commands work on any machine on which the language software has been installed. Your commands, however, 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 programmer's deepest concern, because the compiler or interpreter catches every syntax error and displays a message about 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.

Programmers often compile their code one section at a time. It is far less overwhelming and easier to understand errors that are discovered in 20 lines of code at a time than to try to correct mistakes after 2,000 lines of code have been written. 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 need to retranslate your code only if you make changes to your source code statements. **Figure 1-2** shows a diagram of this entire process.