Understanding Computer Systems

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

- **Hardware** is the equipment, or the physical devices, associated with a computer. For example, keyboards, mice, speakers, and printers are all hardware. The devices are manufactured differently for computers of varying sizes—for example, large mainframes, laptops, and very small devices embedded into products such as telephones, cars, and thermostats. However, 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 games. When you hear people say they have "downloaded an **app** onto a mobile device," they are simply using an abbreviation of *application*.
- **System software** comprises the programs that you use to manage your computer, including operating systems such as Windows, Linux, or UNIX for larger computers and Google Android and Apple iOS for smartphones.

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

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

- **Input**—Data items enter the computer system and are placed in memory, where they can be processed. 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 or finger-swiping movements.
- **Processing**—Processing data items may involve organizing or sorting them, checking them for accuracy, or performing calculations with them. The hardware component that performs these types of tasks is the **central processing unit**, or **CPU**. Some devices, such as

tablets and smartphones, usually contain multiple processors. Writing programs that efficiently use several CPUs requires special techniques.

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

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

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

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

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—often your hard drive.

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

Each programming language uses a piece of software, called a **compiler** or an **interpreter**, to translate your source code into machine language. Machine language is also called **binary**

language, and is represented as a series of 0s and 1s. The compiler or interpreter that translates your code tells you if any programming language component has been used incorrectly. Syntax errors are relatively easy to locate and correct because your compiler or interpreter highlights them. If you write a computer program using a language such as C++ but spell one of its words incorrectly or reverse the proper order of two words, the software lets you know that it found a mistake by displaying an error message as soon as you try to translate the program.

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

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

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

TWO TRUTHS & A LIE

Understanding Computer Systems

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

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

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

Understanding Simple Program Logic

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

Suppose you instruct someone to make a cake as follows:

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

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

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

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

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

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

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

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

The number-doubling process includes three instructions:

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

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

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

- The instruction `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 smartphone display), or the output could go to a printer (which could be laser or ink-jet), or the output could be written to a disk or DVD. The logic of the output process is the same no matter what hardware device you use. When this instruction executes, the value stored in memory at the location named `myAnswer` is sent to an output device. (The output value also remains in computer memory until something else is stored at the same memory location or power is lost.)

Watch the video *A Simple Program*.

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

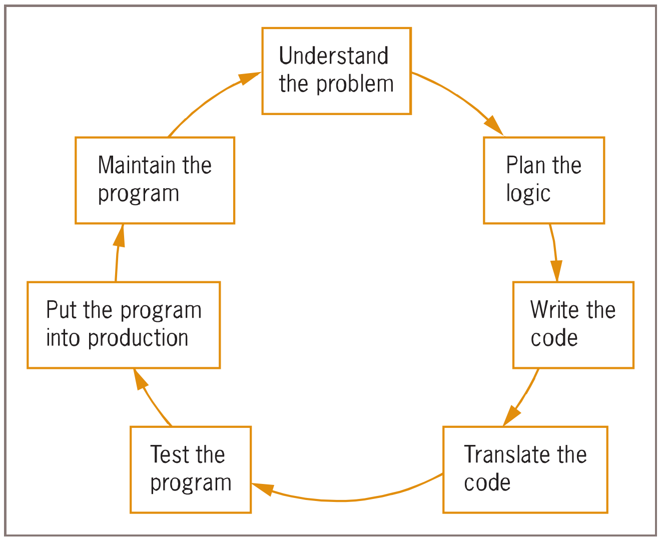

Figure 1-1 The program development cycle
© 2015 Cengage Learning

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.

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

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 or rules you follow to solve a problem.

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

Coding the Program

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

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

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

Using Software to Translate the Program into Machine Language

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

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

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

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

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

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

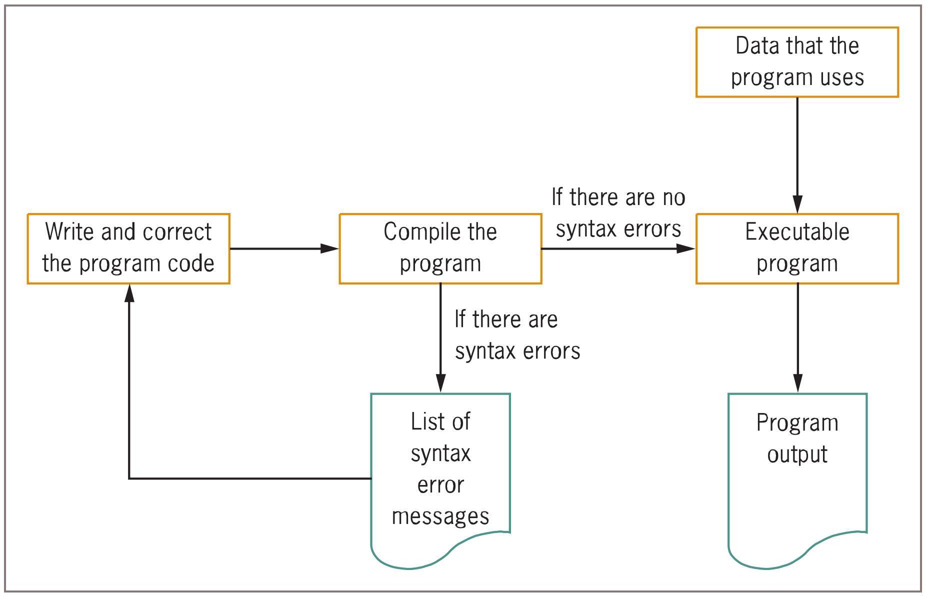
Data that the program uses
Write and correct the program code
Compile the program
If there are no syntax errors
Executable program
If there are syntax errors
List of syntax error messages
Program output

Figure 1-2 Creating an executable program

Testing the Program

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

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

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

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

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

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

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

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

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

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

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

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

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

Putting the Program into Production

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

Maintaining the Program

After programs are put into production, making necessary changes is called **maintenance**. Maintenance can be required for many reasons: for example, because new tax rates are legislated, the format of an input file is altered, or the end user requires additional information not included in the original output specifications. Frequently, your first programming job will require maintaining previously written programs. When you maintain the programs others have written, you will appreciate the effort the original programmer put into writing clear code, using reasonable variable names, and documenting his or her work. When you make changes to existing programs, you repeat the development cycle. That is, you must understand the changes, then plan, code, translate, and test them before putting them into production. If a substantial number of program changes are required, the original program might be retired, and the program development cycle might be started for a new program.

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

TWO TRUTHS & A LIE

Understanding the Program Development Cycle

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

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

Using Pseudocode Statements and Flowchart Symbols

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

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