10 Behind the Scenes: Software Programming

Understanding Software Programming

The Importance of Programming

OBJECTIVE

1. Why do I need to understand how to create software? **(p. 408)**

Sound Byte: Programming for End Users

The Life Cycle of an Information System

OBJECTIVE

2. What is a system development life cycle, and what are the phases in the cycle? **(pp. 409–411)**

The Life Cycle of a Program

OBJECTIVES

3. What is the life cycle of a program? **(p. 412)**
4. What role does a problem statement play in programming? **(pp. 412–414)**
5. How do programmers create algorithms and move from algorithm to code? **(pp. 414–426)**
6. What steps are involved in completing the program? **(pp. 426–427)**

Active Helpdesk: Understanding Software Programming

Programming Languages

Many Languages for Many Projects

OBJECTIVE

7. How do programmers select the right programming language for a specific task? **(pp. 431–432)**

Sound Byte: Programming with the Processing Language

Exploring Programming Languages

OBJECTIVE

8. What are the most popular programming languages for different types of application development? **(pp. 433–440)**

Sound Byte: Using the Arduino Microcontroller

Make This: Explore an App Builder Skill on **page 429**

For all media in this chapter go to **pearsonhighered.com/techinaction** or **MyITLab**.

(Stockbyte/Getty Images; Dondesigns/Getty Images; S John/Fotolia; Erik Dreyer/Stone/Getty Images; Erik Dreyer/Getty Images)

check your understanding// review & practice

For a quick review of what you've learned, answer the following questions. Visit **pearsonhighered.com/techinaction** to check your answers.

multiple choice

1. The individuals responsible for generating images for websites are referred to as
   - **a.** network administrators.
   - **b.** graphic designers.
   - **c.** web programmers.
   - **d.** interface designers.
2. What type of job involves working at client locations and the ability to work with little direct supervision?
   - **a.** field-based
   - **b.** project-based
   - **c.** office-based
   - **d.** home-based
3. Which position is *not* typically a part of the information systems department?
   - **a.** helpdesk analyst
   - **b.** telecommunications technician
   - **c.** network administrator
   - **d.** web server administrator
4. Outsourcing is thought to be an attractive option for many companies because of
   - **a.** the emphasis on employee training.
   - **b.** the cost savings that can be realized.
   - **c.** increased data security.
   - **d.** decreased travel and entertainment costs.
5. Which of the following IT positions is responsible for directing a company's strategy on sites such as Facebook, Twitter, and Yelp?
   - **a.** project manager
   - **b.** customer interaction technician
   - **c.** social media director
   - **d.** social web analyst
6. Which of the following statements about IT careers is *false*?
   - **a.** IT employers typically prefer experience over certification.
   - **b.** Women who have IT skills have ample opportunities for securing IT employment.
   - **c.** Many IT jobs are staying in the United States.
   - **d.** Most IT jobs require little interaction with other people.
7. Which task is *not* typically performed by a network administrator?
   - **a.** developing network usage policies
   - **b.** installing networks
   - **c.** planning for disaster recovery
   - **d.** web programming
8. The oldest scientific computing organization is
   - **a.** Institute of Electrical and Electronics Engineers.
   - **b.** Association for Computing Machinery.
   - **c.** Information Systems Security Association.
   - **d.** Association for Women in Computing.
9. If you are artistic and have mastered software packages such as Adobe Photoshop, Autodesk 3ds Max, and Autodesk Maya, you might consider a career as which of the following?
   - **a.** game programmer
   - **b.** game designer
   - **c.** game tester
   - **d.** web designer
10. Which position is a part of the systems development department?
    - **a.** webmaster
    - **b.** interface designer
    - **c.** programmer
    - **d.** database administrator

HOW COOL IS THIS?

Most major cities now have made a commitment to make the data they use to govern open and transparent—to human eyes and to your programs. Websites format information so it is useful for communicating with humans, but **web services** format the information in a way that makes it easy to communicate the data **to your program**. Chicago, Philadelphia, and other major cities have opened up thousands of government records through **Open Data initiatives**. Using this data, users can do such things as create a program to visualize neighborhood crime statistics, track their city's snowplows, or compare the current school budget expenditures.

It is not just cities becoming more transparent. **Data.gov** opens the U.S. government's data to you. Open government is here. Use it to **write a web app or mobile tool** that really has an **impact in your community**. Consider your own community. What kind of web app is needed? Get started working on it—the data is waiting for you. *(PM Images/Getty Images)*

Scan here for more info

Understanding Software Programming

Every day we face a wide array of tasks. Some tasks are complex and require creative thought and a human touch. But tasks that are repetitive, work with electronic information, and follow a series of clear steps are candidates for automation with computers—automation achieved through programming. In this part of the chapter, we'll look at some of the basics of programming.

the importance of PROGRAMMING

A career in programming offers many advantages—jobs are plentiful, salaries are strong, and telecommuting is often easy to arrange. But even if you're not planning a career in programming, knowing the basics of programming is important to help you use computers productively.

Why would I ever need to create a program? Computer programs already exist for many tasks. For example, if you want to write a paper, Microsoft Word has already been designed to translate the tasks you want to accomplish into computer instructions. However, for users who can't find an existing software product to accomplish a task, programming is mandatory (see Figure 10.1). For example, imagine that a medical company comes up with a new smart bandage designed to transmit medical information about a wound directly to a diagnostic mobile monitor, like a doctor's phone. No software product exists that will accumulate and relay information in just this manner. Therefore, a team of programmers will have to create smart bandage software.

If I'm not going to be a programmer, why should I learn about programming? If you plan to use only existing, off-the-shelf software, having a basic knowledge of programming enables you to add features that support your personal needs. For example, most modern software applications let you automate features by using custom-built miniprograms called *macros*. By creating macros, you can execute a complicated sequence of steps with a single command. Understanding how to program macros enables you to add custom commands to, for example, Word or Excel and lets you automate frequently performed tasks, providing a huge boost to your productivity. And if you plan to create custom applications from scratch, having a detailed knowledge of programming will be critical to the successful completion of your projects. ■

SOUND BYTE

Programming for End Users

In this Sound Byte, you'll be guided through the creation of a macro in the Microsoft Office suite. You'll learn how Office enables you to program with macros in order to customize and extend the capabilities it offers.

FIGURE 10.1 Congressman Bill Shuster takes a 33-mile ride to the Pittsburgh International Airport in a driverless car. Software both controls and drives the car with no human input at all. Legal in a number of states, driverless technology is here. *(Keith Srakocic/AP Images)*

the life cycle of an INFORMATION SYSTEM

Generally speaking, a *system* is a collection of pieces working together to achieve a common goal. Your body, for example, is a system of muscles, organs, and other groups of cells working together. An **information system** includes data, people, procedures, hardware, and software that help in planning and decision making. Information systems help run an office and coordinate online-purchasing systems and are behind database-driven applications used by Amazon and Netflix. But how are these complicated systems developed?

The System Development Life Cycle

Why do I need a process to develop a system? Because teams of people are required to develop information systems, there needs to be an organized process to ensure that development proceeds in an orderly fashion. Software applications also need to be available for multiple operating systems, to work over networked environments, and to be free of errors and well supported. Therefore, a process often referred to as the **system development life cycle (SDLC)** is used.

What steps constitute the SDLC? There are six steps in a common SDLC model, as shown in Figure 10.2. This system is sometimes referred to as a "waterfall" system because each step is dependent on the previous step being completed before it can be started.

1. *Problem and Opportunity Identification:* Corporations are always attempting to break into new markets, develop new customers, or launch new products. At other times, systems development is driven by a company's desire to serve its existing customers more efficiently or to respond to problems with a current system. Whether solving an existing problem or exploiting an opportunity, large corporations typically form a development committee to evaluate systems development proposals. The committee decides which projects to take forward based on available resources such as personnel and funding.
2. *Analysis:* In this phase, analysts explore the problem or need in depth and develop a program specification. The **program specification** is a clear statement of the goals and objectives of the project. The first feasibility assessment is also performed at this stage. The feasibility assessment determines whether the project should go forward. If the project is determined to be feasible, the analysis team defines the user requirements and recommends a plan of action.
3. *Design:* The design phase generates a detailed plan for programmers to follow. The proposed system is documented using flowcharts and data-flow diagrams. Flowcharts are visual diagrams of a process, including the decisions that need to be made along the way. **Data-flow diagrams** trace all data in an information system from the point at which data enters the system to its final resting place (storage or output). The data-flow diagram in Figure 10.3 shows the flow of concert ticket information.

   The design phase details the software, inputs and outputs, backups and controls, and processing requirements of the problem. Once the system plan is designed, a company evaluates existing software packages to determine whether it needs to develop a new piece of software or if it can buy something already on the market and adapt it to fit its needs.
4. *Development:* It is during this phase that actual programming takes place. This phase is also the first part of the program development life cycle, described in detail later in the chapter. The documentation work is begun in this phase by technical writers.

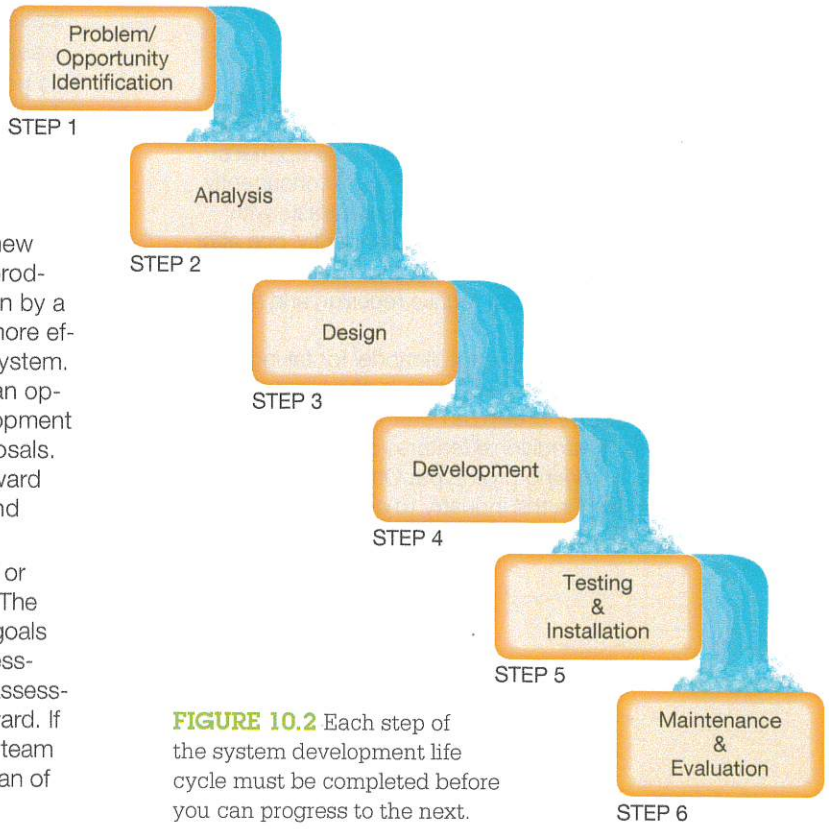


FIGURE 10.2 Each step of the system development life cycle must be completed before you can progress to the next.

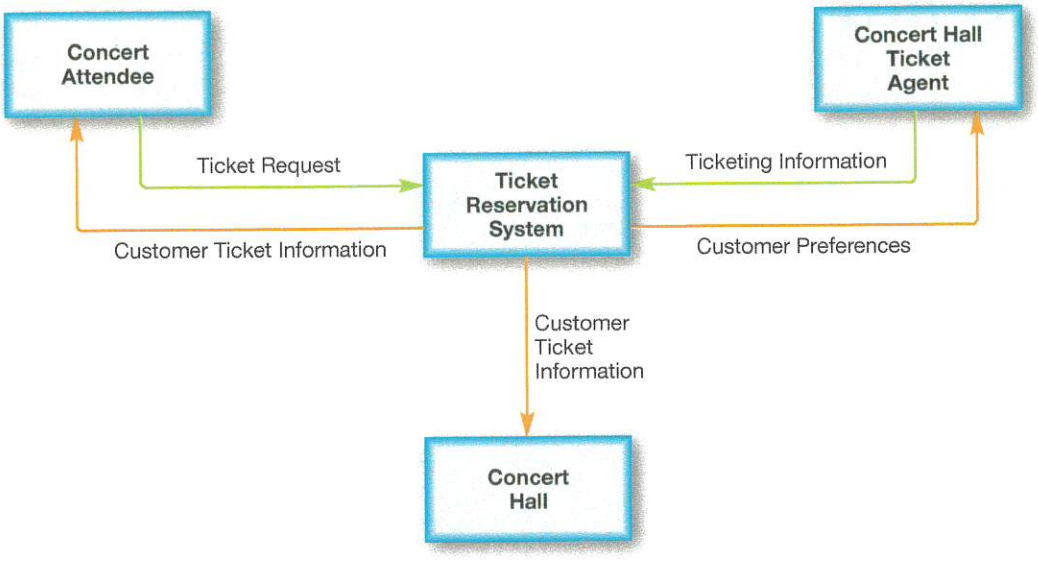


FIGURE 10.3 Data-flow diagrams illustrate the way data travels in an information system.

5. *Testing and Installation:* Testing the program ensures it works properly. It is then installed for official use.
6. *Maintenance and Evaluation:* In this phase, program performance is monitored to determine whether the program is still meeting the needs of end users. Errors that weren't detected in the testing phase but that are discovered during use are corrected. Additional enhancements that users request are evaluated so that appropriate program modifications can be made.

The waterfall model is an idealized view of software development. Most developers follow some variation of it. For example, a design team may "spiral," where a group that's supporting the work of another group will work concurrently with the other group on development. This contrasts with workflows in which the groups work independently, one after the other. Often there is a "backflow" up the waterfall, because even well-designed projects can require redesign and specification changes midstream.

Some people criticize the waterfall model for taking too long to provide actual working software to the client. This may contribute to **scope creep**, an ever-changing set of requests from clients for additional features as they wait longer and longer to see a working prototype. Other developmental models are being used in the industry to address these issues (see the Bits & Bytes sidebar, "The More Minds, the Better"). ■

BITS&BYTES

The More Minds, the Better

In each phase of the SDLC, a style of interaction called *joint application development (JAD)* can be useful in creating successful, flexible results. JAD is popular because it helps designers adapt quickly to changes in program specifications. In JAD, the customer is intimately involved in the project, right from the beginning. Slow communication and lengthy feedback time make the traditional development process extremely time-consuming. In JAD "workshops," there are no communication delays. Such workshops usually include end users, developers, subject experts, observers (such as senior managers), and a facilitator. The facilitator enforces the rules of the meeting to make sure all voices are heard and that agreement is reached as quickly as possible. Also called *accelerated design* or *facilitated team techniques*, JAD's goal is to improve design quality by fostering clear communication. For more details, search the Internet using keywords like *JAD, JAD methodology,* and *JAD tutorial.*

the life cycle of a PROGRAM

Programming is the process of translating a task into a series of commands that a computer will use to perform that task. It involves identifying which parts of a task a computer can perform, describing those tasks in a highly specific and complete manner, and, finally, translating this description into the language spoken by the computer's central processing unit (CPU).

Programming often begins with nothing more than a problem or a request, such as, "Can you tell me how many transfer students have applied to our college?" A proposal will then be developed for a system to solve this problem, if one does not already exist. Once a project has been deemed feasible and a plan is in place, the work of programming begins.

How do programmers tackle a programming project? Each programming project follows several stages, from conception to final deployment. This process is sometimes referred to as the **program development life cycle (PDLC)**:

1. *Describing the Problem:* First, programmers must develop a complete description of the problem. The problem statement identifies the task to be automated and describes how the software program will behave.
2. *Making a Plan:* The problem statement is next translated into a set of specific, sequential steps that describe exactly what the computer program must do to complete the work. The steps are known as an **algorithm**. At this stage, the algorithm is written in natural, ordinary language (such as English).
3. *Coding:* The algorithm is then translated into programming code, a language that is friendlier to humans than the *1*s and *0*s that the CPU speaks but is still highly structured. By coding the algorithm, programmers must think in terms of the operations that a CPU can perform.
4. *Debugging:* The code then goes through a process of debugging in which the programmers repair any errors found in the code.
5. *Testing and Documentation:* The software is tested by both the programming team and the people who will use the program. The results of the entire project are documented for the users and the development team. Finally, users are trained so that they can use the program efficiently.

Figure 10.4 illustrates the steps of a program life cycle.

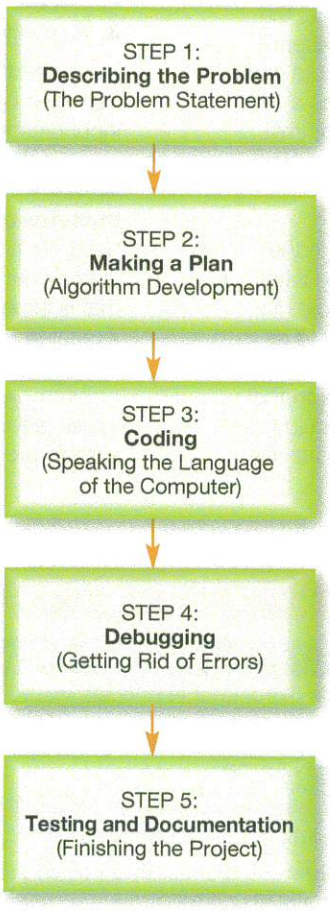


FIGURE 10.4 The stages that each programming project follows, from conception to final deployment, are collectively referred to as the program development life cycle (PDLC).

Now that you have an overview of the process involved in developing a program, let's look at each step in more detail.

Describing the Problem: The Problem Statement

Why is a problem statement necessary? The **problem statement** is the starting point of programming work. It's a clear description of what tasks the computer program must accomplish and how the program will execute those tasks and respond to unusual situations. Programmers develop problem statements so they can better understand the goals of their programming efforts.

What kind of problems can computer programs solve? As noted above, tasks that are repetitive, work with electronic information, and follow a series of clear steps are good candidates for computerization. This might sound as if computers only help us with the dullest and most simplistic tasks. However, many sophisticated problems can be broken down into a series of easily computerized tasks. For example, pharmaceutical companies design drugs using complex computer programs that model molecules. Using simulation software to perform "dry" chemistry, chemists can quickly "create" new drugs and determine whether they will have the desired pharmacological effects. Scientists then select the most promising choices and begin to test those compounds in the "wet" laboratory.

What kinds of problems can computers *not* solve? Computers can't yet act with intuition or be spontaneously creative. They can attack highly challenging problems, such as making weather predictions or playing chess, but only in a way that takes advantage of what computers do best—making fast, reliable computations. Computers don't "think" like humans do. They can only follow instructions and algorithms.

How do programmers create problem statements? The goal in creating a useful problem statement is to have programmers interact with users to describe three things relevant to creating a useful program:

1. **Data** is the raw input that users have at the start of the job.
2. **Information** is the result, or output, that the users require at the end of the job.

3. **Method**, described precisely, is the process of how the program converts the inputs into the correct outputs.

For example, say you want to compute how much money you'll earn working at a parking garage. Your salary is $7.50 per hour for an eight-hour shift, but if you work more than eight hours a day, you get time and a half, which is $11.25 per hour, for the overtime work. To determine how much money you make in any given day, you could multiply this in your mind, write it on a piece of paper, or use a calculator; alternatively, you could create a simple computer program to do the work for you. In this example, what are the three elements of the problem statement?

1. *Data (Input):* the data you have at the beginning of the problem, which is the number of hours you worked and the pay rate.
2. *Information (Output):* the information you need to have at the end of the problem, which is your total pay for the day.
3. *Method (Process):* the set of steps that take you from your input to your required output. In this case, the computer program would check if you worked more than eight hours. This is important because it would determine whether you would be paid overtime. If you didn't work overtime, the output would be $7.50 multiplied by the total number of hours you worked ($60.00 for eight hours). If you did work overtime, the program would calculate your pay as eight hours at $7.50 per hour for the regular part of your shift, plus an additional $11.25 multiplied by the number of overtime hours you worked.

How do programmers handle bad inputs? In the problem statement, programmers also must describe what the program should do if the input data is invalid or just gibberish. This part of the problem statement is referred to as **error handling**.

The problem statement also includes a **testing plan** that lists specific input numbers that the programmers would typically expect the user to enter. The plan then lists the precise output values that a perfect program would return for those input values. Later, in the testing process, programmers use the input and output data values from the testing plan to determine whether the program they created works the way it should. We discuss the testing process later in this chapter.

Does the testing plan cover every possible use of the program? The testing plan can't list every input the program could ever encounter. Instead, programmers work with users to identify the categories of inputs that will be encountered, find a typical example of each input category, and specify what kind of output must be generated. In the parking garage pay example, the error-handling process would describe what the program would do if you happened to enter "–8" (or any other nonsense character) for the number of hours you worked. The error handling would specify whether the program would return a negative value, prompt you to reenter the input, or shut down.

We could expect three categories of inputs in the parking garage example. The user might enter:

1. A negative number for hours worked that day
2. A positive number equal to or less than eight
3. A positive number greater than eight

The testing plan would describe how the error would be managed or how the output would be generated for each input category.

Is there a standard format for a problem statement? Most companies have their own format for documenting a problem statement. However, all problem statements include the same basic components: the data that is expected to be provided (inputs), the information that is expected to be produced (outputs), the rules for transforming the input into output (processing), an explanation of how the program will respond if users enter data that doesn't make sense (error handling), and a testing plan. Figure 10.5 shows a sample problem statement for our parking garage example.

FIGURE 10.5

Complete Problem Statement for the Parking Garage Example

	PROBLEM STATEMENT		
Program Goal	Compute the total pay for a fixed number of hours worked at a parking garage.		
Input	Number of hours worked (a positive number)		
Output	Total pay earned (a positive number)		
Process	Total pay earned is computed as $7.50 per hour for the first eight hours worked each day. Any hours worked consecutively beyond the first eight are calculated at $11.25 per hour.		
Error Handling	The input (number of hours worked) must be a positive real number. If it is a negative number or another unacceptable character, the program will force the user to reenter the information.		
Testing Plan	**INPUT**	**OUTPUT**	**NOTES**
	8	8*7.50	Testing positive input
	3	3*7.50	Testing positive input
	12	8*7.50 + 4*11.25	Testing overtime input
	–6	Error message/ask user to reenter value	Handling error

ACTIVE HELPDESK

Understanding Software Programming

In this Active Helpdesk, you'll play the role of a helpdesk staffer, fielding questions about the life cycle of a program, the role a problem statement plays in programming, how programmers create algorithms and move from algorithm to code to the *1*s and *0*s a CPU can understand, and the steps involved in completing the program.

Making a Plan: Algorithm Development

How does the process start? Once programmers understand exactly what the program must do and have created the final problem statement, they can begin developing a detailed algorithm—a set of specific, sequential steps that describe in natural language exactly what the computer program must do to complete its task. Let's look at some ways in which programmers design and test algorithms.

Do algorithms appear only in programming? Although the term *algorithm* may sound like it would fall only under the domain of computing, you design and execute algorithms in your daily life. For example, say you're planning your morning. You know you need to

1. Get gas for your car,
2. Pick up a mocha latte at the café, and
3. Stop by the bookstore and buy a textbook before your 9 a.m. accounting lecture

You quickly think over the costs and decide it will take $150 to buy all three. In what order will you accomplish all these tasks? How do you decide? Should you try to minimize the distance you'll travel or the time you'll spend driving? What happens if you forget your credit card?

Figure 10.6 presents an algorithm you could develop to make decisions about how to accomplish these tasks. This algorithm lays out a specific plan that encapsulates all of the choices you need to make in the course of completing a particular task and shows the specific sequence in which these tasks will occur. At any point in the morning, you could gather your current data (your inputs)—"I have $20 and my Visa card, but the ATM machine is down"—and the algorithm would tell you unambiguously what your next step should be.

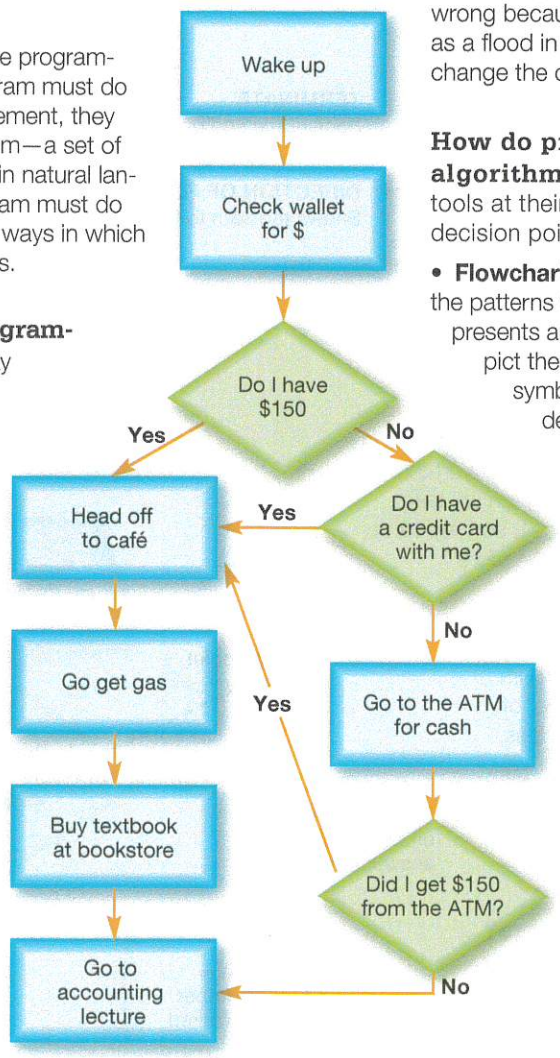


FIGURE 10.6 An algorithm you might use to plan your morning would include all of the decisions you might need to make and would show the specific sequence in which these steps would occur.

What are the limitations of algorithms? An algorithm is a series of steps that's completely known: At each point, we know *exactly* what step to take next. However, not all problems can be described as a fixed sequence of predetermined steps; some involve random and unpredictable events. For example, although the program that computes your parking garage take-home pay each day works flawlessly, programs that predict stock prices are often wrong because many random events (inputs), such as a flood in India or a shipping delay in Texas, can change the outcomes (outputs).

How do programmers represent an algorithm? Programmers have several visual tools at their disposal to help them document the decision points and flow of their algorithms:

- **Flowcharts** provide a visual representation of the patterns the algorithm comprises. Figure 10.6 presents an example of a flowchart used to depict the flow of an algorithm. Specific shape symbols indicate program behaviors and decision types. Diamonds indicate that a yes/no decision will be performed, and rectangles indicate an instruction to follow. Figure 10.7 lists additional flowcharting symbols and explains what they indicate. Many software packages make it easy for programmers to create and modify flowcharts. Microsoft Visio is one popular flowcharting program.
- **Pseudocode** is a text-based approach to documenting an algorithm. In pseudocode, words describe the actions that the algorithm will take. Pseudocode is organized like an outline, with differing levels of indentation to indicate the flow of actions within the program. There is no standard vocabulary for pseudocode. Programmers use a combination of common words in their natural language and the special words that are commands in the programming language they're using.

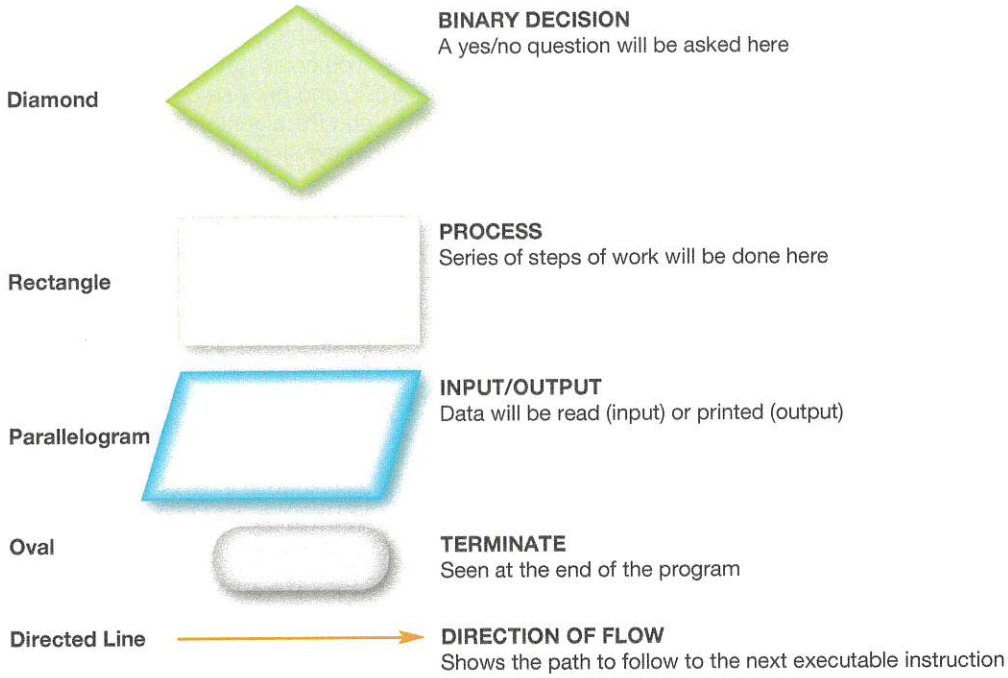


FIGURE 10.7 Standard Symbols Used in Flowcharts

Developing the Algorithm: Decision Making and Design

How do programmers handle complex algorithms?
When programmers develop an algorithm, they convert the problem statement into a list of steps the program will take.

Problems that are complex involve choices, so they can't follow a sequential list of steps. Algorithms therefore include **decision points**, places where the program must choose from a list of actions based on the value of a certain input.

Figure 10.8 shows the steps of the algorithm in our parking garage example. If the number of hours you worked in a given day is eight or less, the program performs one simple calculation: It multiplies the number of hours worked by $7.50. If you worked more than eight hours in a day, the program takes a different path and performs a different calculation.

What kinds of decision points are there?
There are two main types of decisions that change the flow of an algorithm:

1. *Binary decisions:* One decision point that appears often in algorithms is like a "fork in the road." Such decision points are called **binary decisions** because they can be answered in one of only two ways: *yes* (true) or *no* (false). For example, the answer to the question, "Did you work at most eight hours today?" (Is number of hours worked <= 8 hours?), shown in Figure 10.8, is a binary

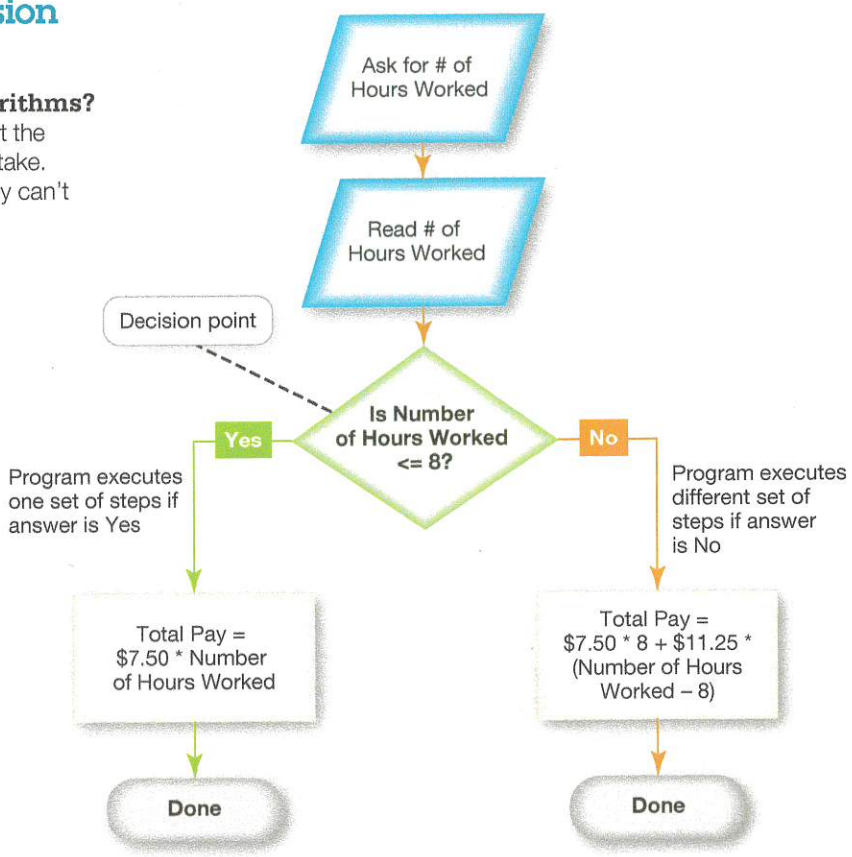


FIGURE 10.8 Decision points force the program to travel down one branch of the algorithm or another.

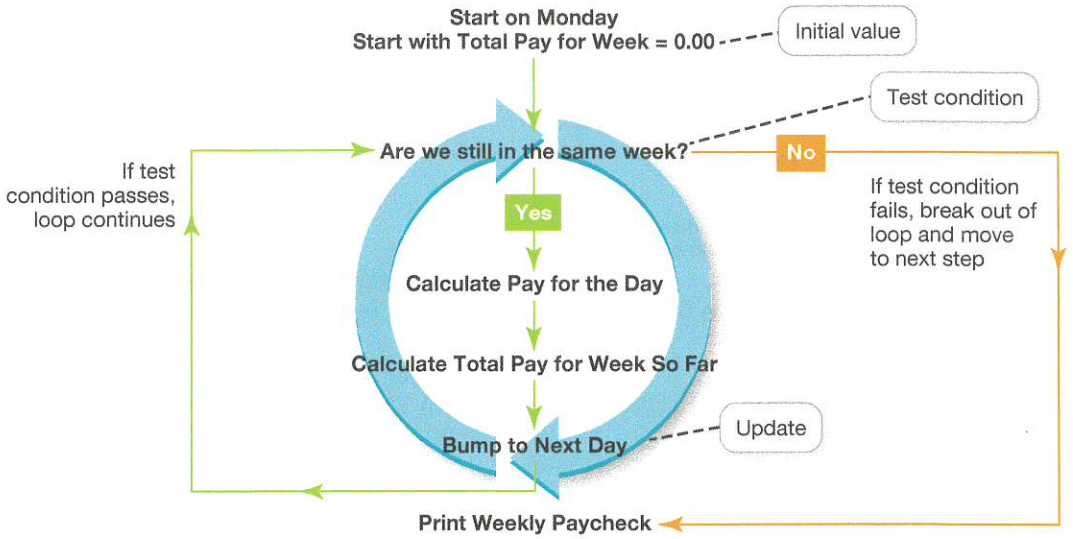


FIGURE 10.9 We stay in the loop until the test condition is no longer true. We then break free from the loop and move on to the next step in the algorithm, which is outside of the loop.

decision because the answer can be only *yes* or *no*. If the answer is *yes*, the program follows one sequence of steps; if the answer is *no*, it follows a different path.

2. *Loops:* A second decision point that often appears in algorithms is a repeating loop. In a **loop**, a question is asked, and if the answer is *yes*, a set of actions is performed. Once the set of actions has finished, the question is asked again, creating a loop. As long as the answer to the question is *yes*, the algorithm continues to loop around and repeat the same set of actions. When the answer to the question is *no*, the algorithm breaks free of the looping and moves on to the first step that follows the loop.

In our parking garage example, the algorithm would require a loop if you wanted to compute the total pay you earned in a full week of work rather than in just a single day. For each day of the week, you would want to perform the same set of steps. Figure 10.9 shows how the idea of looping would be useful in this part of our parking garage program. On Monday, the program would set the Total Pay value to $0.00. It would then perform the following set of steps:

1. Read the number of hours worked that day.
2. Determine whether you qualified for overtime pay.
3. Compute the pay earned that day.
4. Add that day's pay to the total pay for the week.

On Tuesday, the algorithm would loop back, repeating the same sequence of steps it performed on Monday, adding the amount you earned on Tuesday to the Total Pay amount. The algorithm would continue to perform this loop for each day (seven times) until it hits Monday again. At that point, the decision "Are we still in the same week?" would become false. The program would stop, calculate the total pay for the entire week of work, and print the weekly paycheck.

As you can see, there are three important features to look for in a loop:

1. A beginning point, or **initial value**. In our example, the total pay for the week starts at an initial value of $0.00.
2. A set of actions that will be performed. In our example, the algorithm computes the daily pay each time it passes through the loop.
3. A **test condition**, or a check to see whether the loop is completed. In our example, the algorithm should run the loop seven times, no more and no fewer.

Every higher-level programming language supports both making binary yes/no decisions and handling repeating loops. **Control structures** is the general term used for keywords in a programming language that allow the programmer to direct the flow of the program based on a decision.

How do programmers create algorithms for specific tasks? It's difficult for human beings to force their problem-solving skills into the highly structured, detailed algorithms that computing machines require. Therefore, several different methodologies have been developed to support programmers, including *top-down design* and *object-oriented analysis*.

Top-Down Design

What is top-down design? Top-down design is a systematic approach in which a problem is broken into a series of high-level tasks. In top-down design, programmers apply the same strategy repeatedly, breaking each task into successively more detailed subtasks. They continue until they have a sequence of steps that are close to the types of commands allowed by the programming language they'll use for coding. Previous coding experience helps programmers know the

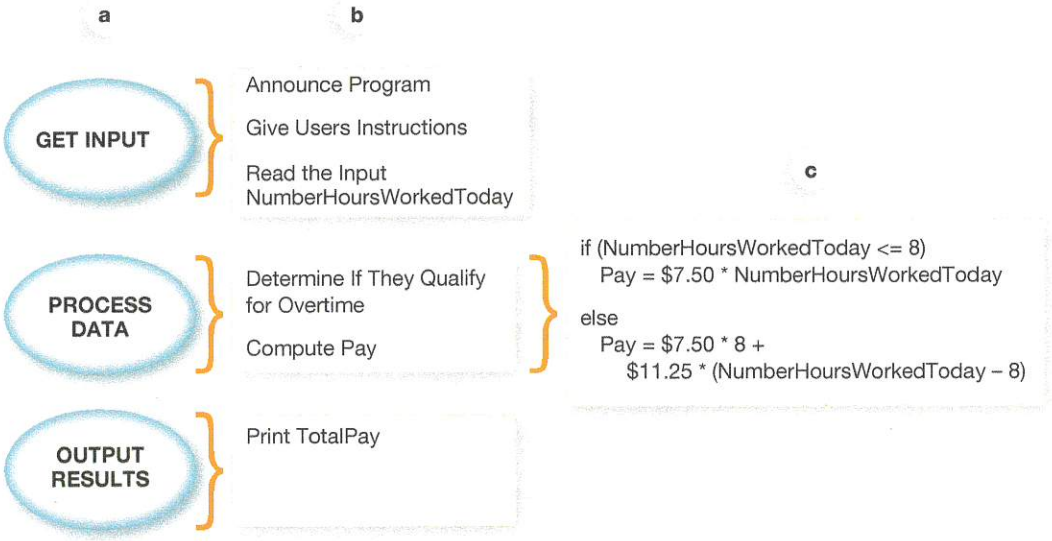


FIGURE 10.10 (a) A top-down design is applied to the highest level of task in our parking garage example, (b) the tasks are further refined into subtasks, and (c) subtasks are refined into a sequence of instructions—an algorithm.

appropriate level of detail to specify in an algorithm generated by top-down design.

How is top-down design used in programming? Let's consider our parking garage example again. Initially, top-down design would identify three high-level tasks: Get Input, Process Data, and Output Results (see Figure 10.10a).

Applying top-down design to the first operation, Get Input, we'd produce the more detailed sequence of steps shown in the first step of Figure 10.10b: Announce Program, Give Users Instructions, and Read the Input NumberHours WorkedToday. When we try to refine each of these steps, we find that they are just print-and-read statements and that most every programming language will support them. Therefore, the operation Get Input has been converted to an algorithm.

Next, we move to the second high-level task, Process Data, and break it into subtasks. In this case, we need to determine whether overtime hours were worked and to compute the pay accordingly. We continue to apply top-down design on all tasks until we can no longer break tasks into subtasks, as shown in Figure 10.10c.

Object-Oriented Analysis

What is object-oriented analysis? With **object-oriented analysis**, programmers first identify all the categories of inputs that are part of the problem the program is meant to solve. These categories are called **classes**. With the object-oriented approach, the majority of design time is spent identifying the classes required to solve the problem, modeling them, and thinking about what relationships they need to be able to have with each other. Constructing the algorithm becomes a process of enabling the objects to interact. For example, the classes in our parking garage example might include a TimeCard class and an Employee class.

BITS&BYTES

Hackathons

College hackathons—coding events that take place in a single day or a single weekend—are more popular than ever. The PennApps Hackathon at the University of Pennsylvania sees over 1,000 programmers attend from over 100 universities. Terrific cash prizes are awarded along with sought-after meetings with engineers at Google, Dropbox, and more. Hackathons to create software with a purpose are appearing more and more too. The National Day of Civic Hacking (**hackforchange.org**) and Code for America (**codeforamerica.org**) are both examples of hackers working to help develop solutions to local and global needs.

Hackathons can help you build your programming skills incredibly quickly. If you don't have a team, most hackathons link you up with other programmers on the spot. Visit ChallengePost (**challengepost.com**) to find the coding events nearest you—and join in the competition!

Classes are defined by

- Information (data) within the class and
- Actions (methods) associated with the class

For example, as shown in Figure 10.11, *data* for an Employee would include a Name, Address, and Social Security Number, whereas the *methods* for the Employee would be GoToWork(), LeaveWork(), and CollectPay(). The data of the class describes the class, so classes are often characterized as nouns, whereas methods are often characterized as verbs—the ways that the class acts and communicates with other classes.

Programmers may need to create several different examples of a class. Each of these examples is an **object**. In Figure 10.11, John Doe, Jane Doe, and Bill McGillicutty are each Employee objects (specific examples of the Employee class). Each object from a given class is described by the same pieces of data and has the same methods; for example, John, Jane, and Bill are all Employees and can use the GoToWork(), LeaveWork(), and CollectPay() methods. However, because they all have different pay grades (PayGrade 5, PayGrade 10, and PayGrade 4, respectively) and different Social Security numbers, they are all unique objects.

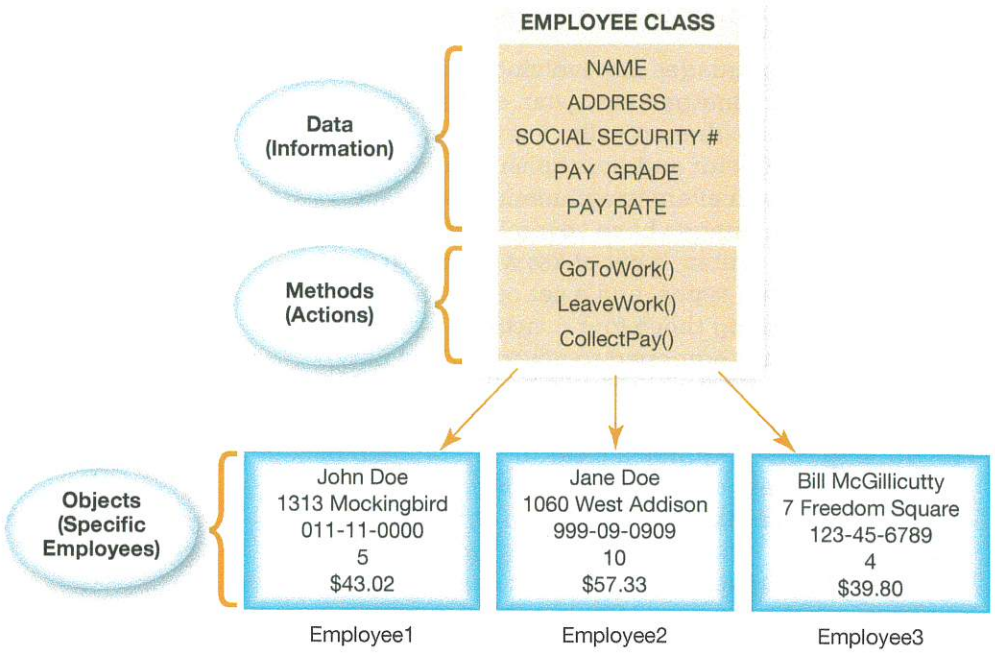


FIGURE 10.11 The Employee class includes the complete set of information (data) and actions (methods or behaviors) that describe an Employee.

Why would a developer select the object-oriented approach over top-down design? An important aspect of object-oriented design is that it leads to **reusability**. Object-oriented analysis forces programmers to think in general terms about their problem, which tends to lead to more general and reusable solutions. Because object-oriented design generates a family of classes for each project, programmers can easily reuse existing classes from other projects, enabling them to produce new code quickly.

How does a programmer take advantage of reusability? Programmers must study the relationships between objects. Hierarchies of objects can be built quickly in object-oriented languages using the mechanism of inheritance. **Inheritance** means that a new class can automatically pick up all the data and methods of an existing class and then can extend and customize those to fit its own specific needs.

The original class is called the **base class**, and the new, modified class is called the **derived class**, as illustrated in Figure 10.12. You can compare this with making cookies.

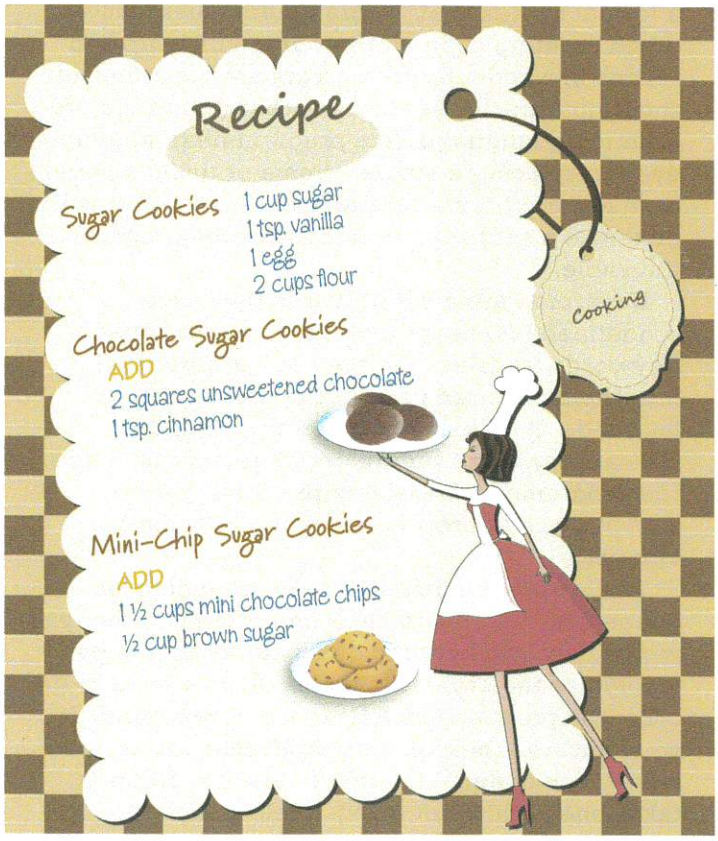


FIGURE 10.12 In object-oriented programming, a single base class—for example, Sugar Cookies—helps you quickly create many additional derived classes, such as Chocolate and Mini-Chip Sugar Cookies. *(Nataliya Dolotko/Fotolia)*

DIG DEEPER

The Building Blocks of Programming Languages: Syntax, Keywords, Data Types, and Operators

Programming languages are evolving constantly. New languages emerge every year, and existing languages change dramatically. Therefore, it would be extremely difficult and time consuming for programmers to learn every programming language. However, all languages have several common elements: rules of syntax, a set of keywords, a group of supported data types, and a set of allowed operators. By learning these four concepts, a programmer will be better equipped to approach any new language.

The transition from a well-designed algorithm to working code requires a clear understanding of the rules (syntax) of the programming language being used. **Syntax** is an agreed-on set of rules defining how a language must be structured. The English language has a syntax that defines which symbols are words (for example, *gorilla* is a word but *allirog* is not) and what order words and symbols (such as semicolons and commas) must follow.

Likewise, all programming languages have a formal syntax that programmers must follow when creating code **statements**, which are sentences in a code. **Syntax errors** are violations of the strict, precise set of rules that define the programming language. In a programming language, even misplacing a single comma or using a lowercase letter where a capital letter is required will generate a syntax error and make the program unusable.

Keywords are a set of words that have predefined meanings for a particular language. Keywords translate the flow of the algorithm into the structured code of the programming language. For example, when the algorithm indicates where a binary decision must be made, the programmer translates that binary decision into the appropriate keyword(s) from the language.

To illustrate further, in the programming language C++, the binary decision asking whether you worked enough hours to qualify for overtime pay would use the keywords **if else**. At this point in the code, the program can follow one of two paths: If you indicated through your input that you worked fewer than or equal to eight hours, the program takes one path; if not (else), it follows another. Figure 10.13 shows this binary decision in the algorithm and the corresponding lines of C++ code that use the "if else" keywords.

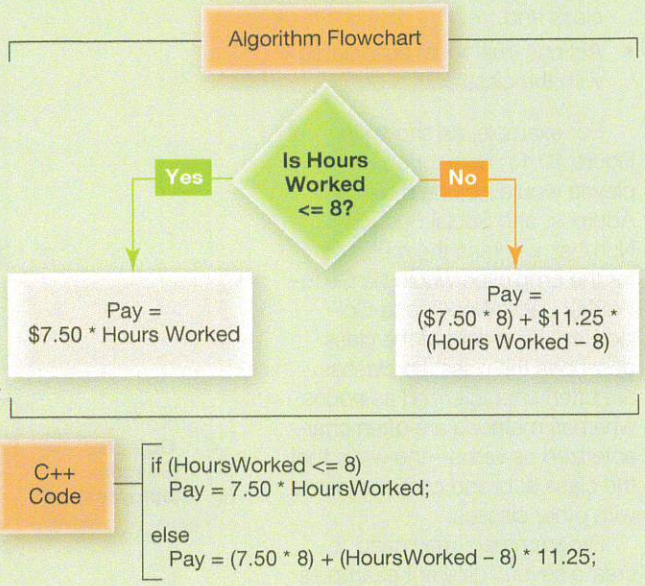


FIGURE 10.13 The binary decision in the algorithm has been converted into C++ code.

Loops are likewise translated from algorithm to code by using the appropriate keyword from the language. For example, in the programming language Visual Basic, programmers use the keywords **For** and **Next** to implement a loop. After the keyword For, an input or output item is given a starting value. Then the statements, or "sentences," in the body of the loop are executed. When the command Next is run, the program returns to the For statement and increments the value of the input or output item by 1. It then tests that the value is still inside the range given. If it is, the body of the loop is executed again. This continues until the value of the input or output item is outside the range listed. The loop is then ended, and the statement that follows the loop is run.

In the parking garage example, the following lines of Visual Basic code loop to sum up the total pay for the entire week:

```
For Day = 1 to 7
    TotalPay = TotalPay + Pay
Next Day
```

In this statement, the starting value of the input item Day is 1, and the program loops through until Day equals 7. Then, when Day equals 8, we move out of the loop, to the line of code that comes after the loop.

Often, a quick overview of a language's keywords can reveal the unique focus of that language. For example, the language C++ includes the keywords *public, private,* and *protected,* which indicate that the language includes a mechanism for controlling security.

Each time programmers want to store data in their program, they must ask the OS for storage space at a RAM location. **Data types** describe the kind of data being stored at the memory location. Each programming language has its own data types (although there is some degree of overlap). For example, C++ includes data types that represent integers, real numbers, characters, and Boolean (true–false) values. These C++ data types show up in code statements as *int* for integer, *float* for real numbers, *char* for characters, and *bool* for Boolean values.

Because it takes more room to store a real number such as 18,743.23 than it does to store the integer 1, programmers use data types in their code to indicate to the OS how much memory it needs to allocate. Programmers must be familiar with all the data types available in the language so that they can assign the most appropriate one for each input and output value, without wasting memory space.

Operators are the coding symbols that represent the fundamental actions of the language. Each programming language has its own set of operators. Many languages include common algebraic operators such as +, –, *, and / to represent the mathematical operations of addition, subtraction, multiplication, and division, respectively. Some languages, however, introduce new symbols as operators. The language APL (short for A Programmer's Language) was designed to solve multidimensional mathematical problems. APL includes less familiar operators such as rho, sigma, and iota, each representing a complex mathematical operation. Because it contains many specialized operators, APL requires programmers to use a special keyboard (see Figure 10.14).

Programming languages sometimes include other unique operators. The C++ operator >>, for example, is used to tell the computer to read data from the keyboard or from a file. The C++ operator && is used to tell the computer to check whether two statements are both true. "Is your age greater than 20 AND less than 30?" is a question that requires the use of the && operator.

In the following C++ code, several operators are being used. The > operator checks whether the number of hours worked is greater than 0. The && operator checks that the number of hours worked is simultaneously positive AND less than or equal to 8. If that happens, then the = operator sets the output Pay to equal the number of hours multiplied by $7.50:

```
if (Hours > 0 && Hours <= 8)
    Pay = Hours * 7.50;
```

Knowing operators such as these, as well as the other common elements described earlier, helps programmers learn new programming languages.

FIGURE 10.14 APL requires programmers to use an APL keyboard that includes the many specialized operators in the language.

For example, you have a basic recipe for sugar cookies (base class: Sugar Cookies). However, in your family, some people like chocolate-flavored sugar cookies (derived class: Chocolate Sugar Cookies), and others like mini-chip sugar cookies (derived class: Mini-Chip Sugar Cookies). All the cookies share the attributes of the basic sugar cookie. However, instead of creating two entirely new recipes—one for chocolate cookies and one for mini-chip cookies—the two varieties inherit the basic sugar cookie (base class) recipe; the recipe is then customized to make the chocolate and mini-chip sugar cookies (derived classes).

Coding: Speaking the Language of the Computer

How is a person's idea translated into CPU instructions? Once programmers create an algorithm, they select the best programming language for the problem and then translate the algorithm into that language. Translating an algorithm into a programming language is the act of **coding**. Programming languages are somewhat readable by humans but then are translated into patterns of *1*s and *0*s to be understood by the CPU.

Although programming languages free programmers from having to think in binary language—the *1*s and *0*s that computers understand—they still force programmers to translate the ideas of the algorithm into a highly precise format. Programming languages are quite limited, allowing programmers to use only a few specific keywords, while demanding a consistent structure.

How exactly do programmers move from algorithm to code? Once programmers have an algorithm, in the form of either a flowchart or a series of pseudocode statements, they identify the key pieces of information the algorithm uses to make decisions:

- What steps are required for the calculation of new information?
- What is the exact sequence of the steps?
- Are there points where decisions have to be made?
- What kinds of decisions are made?
- Are there places where the same steps are repeated several times?

Once programmers identify the required information and the flow of how it will be changed by each step of the algorithm, they can begin converting the algorithm into computer code in a specific programming language.

What exactly is a programming language? A **programming language** is a kind of "code" for the set of instructions the CPU knows how to perform. Computer programming languages use special words and strict rules so that programmers can control the CPU without having to know all its hardware details.

What kinds of programming languages are there? Programming languages are classified into several major groupings, sometimes referred to as *generations*. With each generation of language development, programmers have been relieved of more of the burden of keeping track of what the hardware requires. The earliest languages—assembly language and machine languages—required the programmer to know a great deal about how the computer was constructed internally and how it stored data. Programming is becoming easier as languages continue to become more closely matched to how humans think about problems.

How have modern programming languages evolved? There are five major categories of languages:

1. A **first-generation language (1GL)** is the actual **machine language** of a CPU, the sequence of bits (*1*s and *0*s) that the CPU understands.
2. A **second-generation language (2GL)** is also known as an **assembly language**. Assembly languages allow programmers to write their programs using a set of short, English-like commands that speak directly to the CPU and that give the programmer direct control of hardware resources.
3. A **third-generation language (3GL)** uses symbols and commands to help programmers tell the computer what to do. This makes 3GL languages easier for humans to read and remember. Most programming languages today, including BASIC, FORTRAN, COBOL, C/C++, and Java, are considered third generation.
4. **Fourth-generation languages (4GLs)** include database query languages and report generators. **Structured Query Language (SQL)** is a database programming language that is an example of a 4GL. The following SQL command would check a huge table of data on the employees and build a new table showing all those employees who worked overtime:

```
SELECT EmployeeName, TotalHours FROM
EMPLOYEES WHERE "TotalHours" Total
More Than 8
```

FIGURE 10.15

Sample Code for Different Language Generations

GENERATION	EXAMPLE	SAMPLE CODE
1GL	Machine	**Bits** describe the commands to the CPU. 1110 0101 1001 1111 0000 1011 1110 0110
2GL	Assembly	**Words** describe the commands to the CPU. ADD Register 3, Register 4, Register 5
3GL	FORTRAN, BASIC, C, Java	**Symbols** describe the commands to the CPU. TotalPay = Pay + OvertimePay
4GL	SQL	**More powerful commands** allow complex work to be done in a single sentence. SELECT isbn, title, price, price*0.06 AS sales_tax FROM books WHERE price>100.00 ORDER BY title;
5GL	PROLOG	Programmers can build applications **without specifying an algorithm**. Find all the people who are Mike's cousins as: ?-cousin (Mike, family)

5. **Fifth-generation languages (5GLs)** are considered the most "natural" of languages. In a 5GL, a problem is presented as a series of facts or constraints instead of as a specific algorithm. The system of facts can then be queried (asked) questions. PROLOG (PROgrammingLOGic) is an example of a 5GL. A PROLOG program could be a list of family relationships and rules such as "Mike is Sally's brother. A brother and a sister have the same mother and father." Once a user has amassed a huge collection of facts and rules, he or she can ask for a list of all Mike's cousins, for example. PROLOG would find the answers by repeatedly applying the principles of logic instead of following a systematic algorithm that the programmer provided.

Figure 10.15 shows small code samples of examples of each generation of language.

Do programmers have to use a higher-level programming language to solve a problem with a computer? No, experienced programmers sometimes write a program directly in the CPU's assembly language. However, the main advantage of higher-level programming languages—C or Java, for example—is that they allow programmers to think in terms of the problem they're solving rather than having to worry about the internal design and specific instructions available for a given CPU. In addition, higher-level programming languages have the capability to easily produce a program that will run on differently configured CPUs. For example, if programmers wrote directly in the assembly language for an Intel i7 CPU, they would have to rewrite the program completely if they wanted it to run on a Sun workstation with the SPARC T5CPU. Thus, higher-level programming languages offer **portability**—the capability to move a completed solution easily from one type of computer to another.

What happens first when you write a program? All of the inputs a program receives and all of the outputs the program produces need to be stored in the computer's RAM while the program is running. Each input and each output item that the program manipulates, also known as a **variable**, needs to be announced early in the program so that memory space can be set aside. A **variable declaration** tells the operating system that the program needs to allocate storage space in RAM. The following line of code is a variable declaration in the language Java:

```
int Day;
```

The one simple line makes a lot of things happen behind the scenes:

- The variable's name is *Day*. The "int" that precedes the Day variable indicates that this variable will always be an integer (a whole number such as 15 or –4).
- The statement "int Day;" asks for enough RAM storage space to hold an integer. After the RAM space is found, it is reserved. As long as the program is running, these RAM cells will be saved for the Day variable, and no other program can use that memory until the program ends. From that point on, when the program encounters the symbol Day, it will access the memory it reserved as Day and find the integer stored there.

Can programmers leave notes to themselves inside a program? Programmers often insert a **comment** into program code to explain the purpose of a section of code, to indicate the date they wrote the program, or to include other important information about the code so that fellow programmers can more easily understand and update it.

Comments are written into the code in plain English. The *compiler*, a program that translates codes into binary *1*s and *0*s, just ignores comments. Comments are intended to be read only by human programmers. Languages provide a special symbol or keyword to indicate the beginning of a comment. In C++, the symbol // at the beginning of a line indicates that the rest of the line is a comment.

What would completed code for a program look like? Figure 10.16 presents a completed C++ program for our parking garage example. We'll discuss later how you would actually enter this program on your system. This program, which is written in a top-down style, does not make use of objects. Each statement in the program is executed sequentially (that is, in order from the first statement to the last) unless the program encounters a keyword that changes the flow. In the figure:

1. The program begins by declaring the variables needed to store the program's inputs and outputs in RAM.
2. Next, the *for* keyword begins a looping pattern. All of the steps between the very beginning bracket ({) and the end one (}) will be repeated seven times to gather the total pay for each day of the week.
3. The next section collects the input data from the user.
4. The program then checks that the user entered a reasonable value. In this case, a positive number for hours worked must be entered.
5. Next, the program processes the data. If the user worked eight or fewer hours, he or she is paid at the rate of $7.50, whereas hours exceeding eight are paid at $11.25.
6. The final statement updates the value of the TotalPay variable. The end bracket indicates that the program has reached the end of a loop. The program will repeat the loop

```
#include <iostream>
using namespace std;

void
main()
{
    //Begin by asking for some variables to be stored in RAM

    float NumberHoursWorkedToday = 0.0;
    float Pay=0.0, TotalPay=0.0;
    int Day;

    //Set up a loop to ask the user how many hours they worked each day

    for( Day = 1; Day <= 7; Day++)
    {
        //Read the input data from the screen
        cout << " Enter the number of hours you worked on day " << Day << ": ";
        cin >> NumberHoursWorkedToday;

        //Check the input makes sense
        if ( NumberHoursWorkedToday < 0 )
        {
            //Wait a minute! We need to handle possible errors here.
            //Print a warning message to the user if they enter a negative value.

            cout << "You can't work negative hours! Try again :";
            cin >> NumberHoursWorkedToday;

        }

        if ( NumberHoursWorkedToday <= 8 )

            //Compute pay earned today at normal base rate
            Pay = NumberHoursWorkedToday * 7.50;

        else

            //Compute the pay earned using the overtime rule
            Pay = ( 8 * 7.50 ) + ( 11.25 * ( NumberHoursWorkedToday - 8) );

        // Update the total pay you have earned so far this week
        TotalPay = TotalPay + Pay;

    } // when we hit this brace, we bounce back up to the beginning of the loop

    //Now we're free from the loop! Let's print then go spend our paycheck!
    cout << " Your totalpay for this week comes to : " << TotalPay;

} // The program is done .
```

Comments (green)

STEP 1: Declare variables

STEP 2: Create loop

STEP 3: Collect input data

STEP 4: Check for errors

Keywords (blue)

Commands (black)

Loop

STEP 5: Process data

STEP 6: Update variables

FIGURE 10.16 This complete C++ program solves the parking garage pay problem.

to collect and process the information for the next day. When the seventh day of data has been processed, the Day variable will be bumped up to the next value, 8. This causes the program to fail the test (Day <= 7?). At that point, the program exits the loop, prints the results, and quits.

Are there ways in which programmers can make their code more useful for the future? One aspect of converting an algorithm into good code is the programmer's ability to design general code that can easily be adapted to new settings. Sections of code that will be used repeatedly, with only slight modification, can be packaged into reusable "containers" or components. Depending on the language, these reusable components are referred to as *functions, methods, procedures, subroutines, modules,* or *packages*.

In our program, we could create a function that implements the overtime pay rule. As it stands in Figure 10.16, the code works only in situations in which the hourly pay is exactly $7.50 and the bonus pay is exactly $11.25. However, if we rewrote this part of the processing rules as a function, we could have code that would work for any base pay rate and any overtime rate. If the base pay rate or overtime rate changed, the function would use whichever values it was given as input to compute the output pay variable. Such a function, as shown in Figure 10.17, may be reused in many settings without changing any of the code.

Compilation

How does a programmer move from code in a programming language to the 1s and 0s the CPU can understand? Compilation is the process by which code is converted into machine language—the language the CPU can understand. A **compiler** is a program that understands both the syntax of the programming language and the exact structure of the CPU and its machine language. It can "read" the **source code**, which comprises the instructions programmers have written in the higher-level language and can translate the source code directly into machine language—the binary patterns that will execute commands on the CPU. You can learn more about the details of the binary number system in the Technology in Focus section, "Under the Hood" (see page 298).

Each programming language has its own compiler. It's a program that you purchase and install just like any other software on your system. Separate versions of the compiler are required if you want to compile code that will run on separate processor types. One version of a compiler would create finished programs for a Sun SPARC T5 processor, for example, and another version of the compiler would create programs for an Intel i7 CPU.

At this stage, once the compiler has generated the executable program, the programmers have a program to distribute. An **executable program**, the binary sequence that instructs

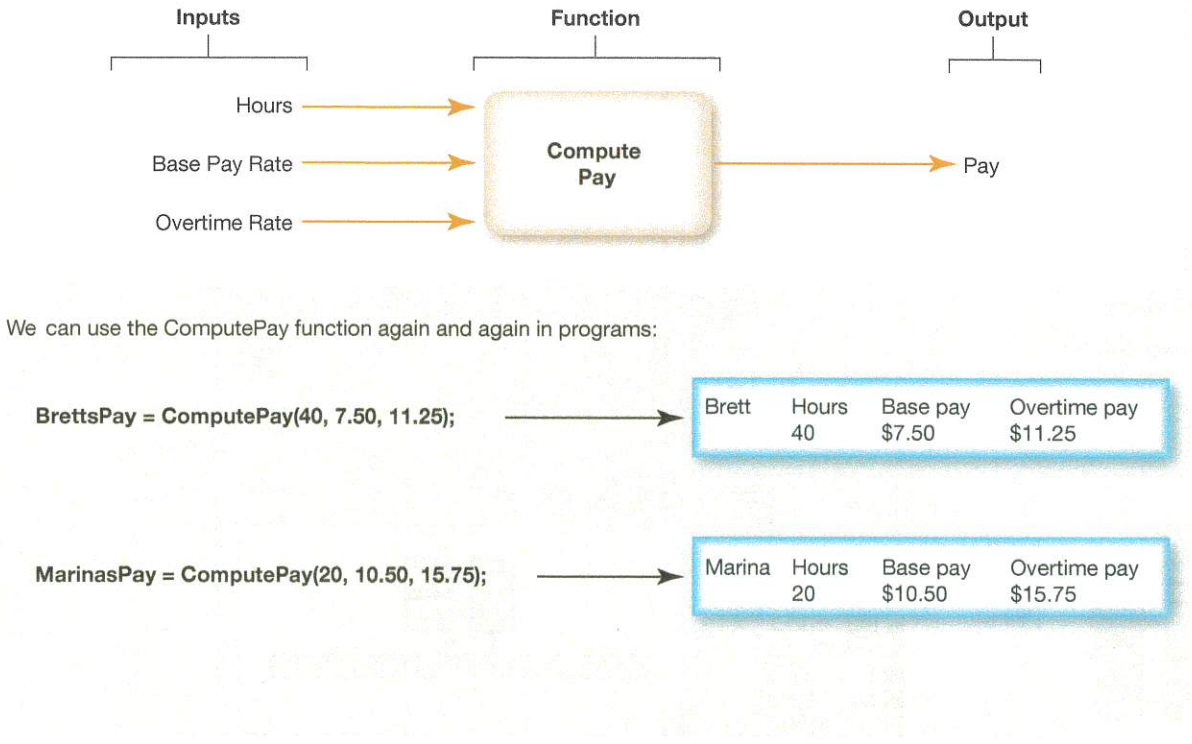


FIGURE 10.17 A function is a reusable component that can be used in different settings.

the CPU to run their code, can't be read by human eyes because they're pure binary codes. They're stored as *.exe or *.com files on Windows systems.

Does every programming language have a compiler? Some programming languages don't have a compiler but use an interpreter instead. An **interpreter** translates the source code into an intermediate form, line by line. Each line is then executed as it's translated. The compilation process takes longer than the interpretation process because in compilation, all the lines of source code are translated into machine language before any lines are executed. However, the finished compiled program runs faster than an interpreted program because the interpreter is constantly translating and executing as it goes.

If producing the fastest executable program is important, programmers will choose a language that uses a compiler instead of an interpreter. For development environments in which many changes are still being made to the code, interpreters have an advantage because programmers don't have to wait for the entire program to be recompiled each time they make a change. With interpreters, programmers can immediately see the results of their program changes as they're making them.

Coding Tools: Integrated Development Environments

What tools make the coding process easier? Modern programming is supported by a collection of tools that make the writing and testing of software easier. An **integrated development environment (IDE)** is a developmental tool that helps programmers write and test their programs. One IDE can often be configured to support many different languages. Figure 10.18 shows the IDE jGRASP working with Java code.

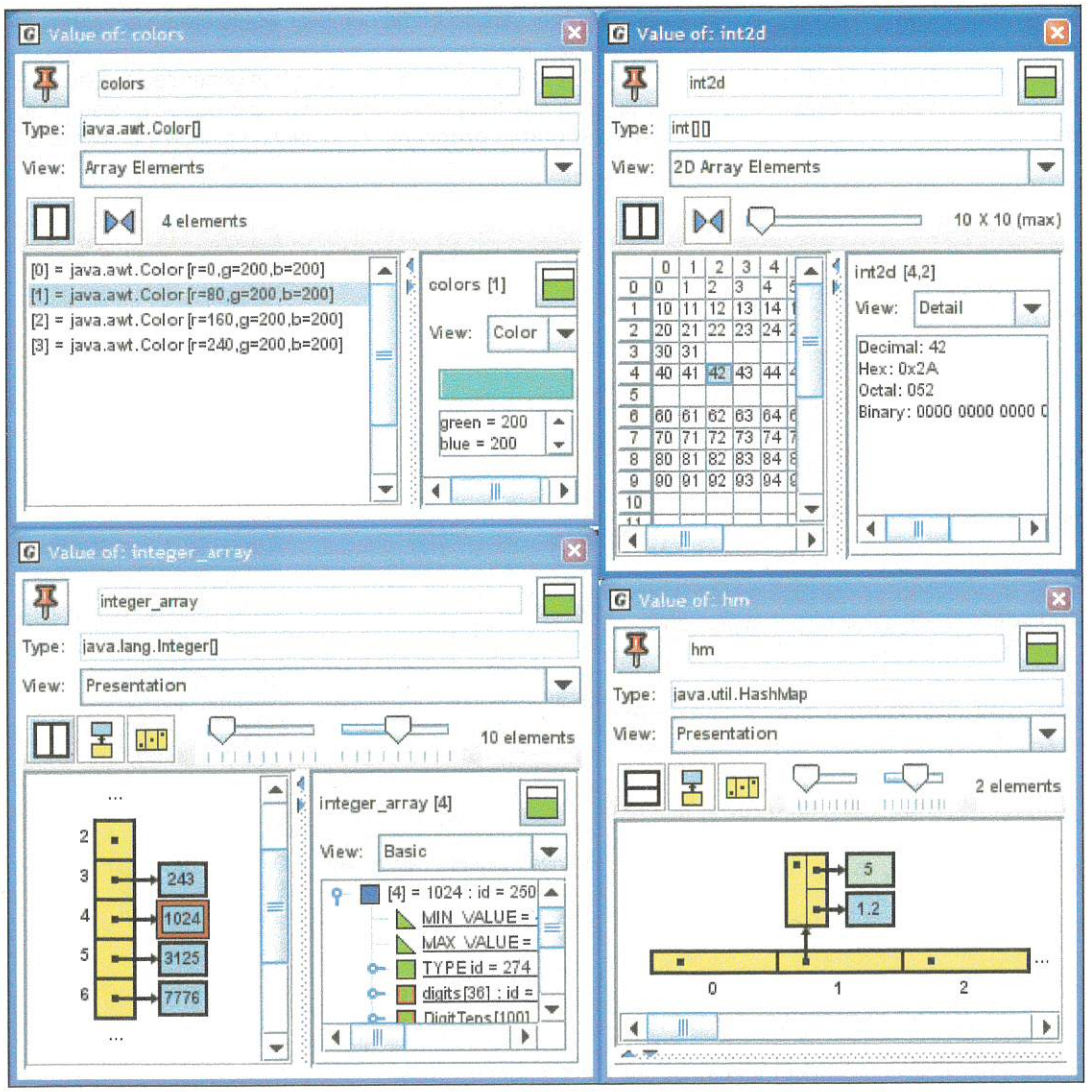


FIGURE 10.18 The jGRASP IDE is a free tool that helps the programmer visualize the code as it runs to help eliminate logical errors. *(Courtesy of Auburn University)*

How does an IDE help programmers when they're typing the code? Code editing is the step in which a programmer physically types the code into the computer. An IDE includes an **editor**, a special tool that helps programmers as they enter the code, highlighting keywords and alerting the programmers to typos. Modern IDE editors also automatically indent the code correctly, align sections of code appropriately, and apply color to code comments to remind programmers that these lines won't be executed as code. In addition, IDEs provide auto-completion of code, suggest solutions to common errors, and more.

How does the IDE help programmers after code editing is finished? Editing is complete when the entire program has been keyed into the editor. At that time, the programmer clicks a button in the IDE, and the compilation process begins. The IDE shows how the compilation is progressing, which line is currently being compiled, how many syntax errors have been identified, and how many warnings have been generated. A warning is a suggestion from the compiler that the code might not work in the way the programmer intended even though there's no formal syntax error on the line.

A syntax error is a violation of the strict, precise set of rules that define the language. Programmers create syntax errors when they misspell keywords (such as typing BEEGIN instead of BEGIN) or use an operator incorrectly (such as typing x = , y + 2 instead of x = y + 2). Once compilation is finished, the IDE presents all of the syntax errors in one list. The programmer can then click any item in the list to see a detailed explanation of the type of error. When the programmer double-clicks an item in the list, the editor jumps to the line of code that contains the error, enabling the programmer to repair syntax errors quickly.

FIGURE 10.19 Debugging—the process of correcting errors in a program—combines logic with an understanding of the problem to be solved. *(Pictafolio/Getty Images)*

Debugging: Getting Rid of Errors

What is debugging? Once the program has compiled without syntax errors, it has met all of the syntax rules of the language. However, this doesn't mean that the program behaves in a logical way or that it appropriately addresses the task the algorithm described. If programmers made errors in the strategy used in the algorithm or in how they translated the algorithm to code, problems will occur. The process of running the program over and over to find and repair errors and to make sure the program behaves in the way it should is termed **debugging** (see Figure 10.19).

How do programmers know the program has solved the problem? At this point in the process, the testing plan that was documented as part of the problem statement becomes critically important to programmers. The testing plan clearly lists input and output values, showing how the users expect the program to behave in each input situation. It's important that the testing plan contain enough specific examples to test every part of the program.

In the parking garage problem, we want to make sure the program calculates the correct pay for a day when you worked eight or fewer hours and for a day when you worked more than eight hours. Each of these input values forces the program to make different decisions in its processing path. To be certain the program works as intended, programmers try every possible path.

For example, once we can successfully compile the example code for the parking garage problem, we can begin to use our testing plan. The testing plan indicates that an input of 3 for NumberHoursWorkedToday must produce an output of Pay = 3 * $7.50 = $22.50. In testing, we run the program and make sure that an input value of 3 yields an output value of $22.50. To check that the processing path involving overtime is correct, we input a value of 12 hours. That input must produce Pay = 8 * $7.50 + (12 – 8) * $11.25 = $105.00.

A complete testing plan includes sample inputs that exercise all the error handling required as well as all the processing paths. Therefore, we would also want to check how the program behaves when NumberHoursWorkedToday is entered as –6.

If the testing plan reveals errors, why does the program compile? The compiler can't think through code or decide whether what the programmer wrote is logical. The compiler can only make sure that the specific rules of the language are followed.

For example, if in the parking garage problem we happened to type the if statement as

```
if (NumberHoursWorkedToday > 88)
   //Use the Overtime Pay rule
```

instead of

```
if (NumberHoursWorkedToday > 8)
   //Use the Overtime Pay rule
```

the compiler wouldn't see a problem. It doesn't seem strange to the compiler that you only get overtime after working 88 hours a day. These **logical errors** in the problem are caught only when the program executes.

Another kind of error caught when the program executes is a **runtime error**. For example, it's easy for a programmer to accidentally write code that occasionally divides by zero, a big "no-no" mathematically! That kind of forbidden operation generates a runtime error message.

BITS&BYTES

Many Languages on Display

At the 99 Bottles of Beer site **99-bottles-of-beer.net**, you can find a simple program that displays the lyrics to the song "99 Bottles of Beer on the Wall." If you've ever sat through round after round of this song on a long school bus trip, you know how repetitive it is. That means the code to write this song can take advantage of looping statements. This site presents the program in more than 1,500 different languages. Also check out the Rosetta Code wiki (**rosettacode.org/wiki/Rosetta_Code**) to see solutions to many other programming problems in a wide variety of languages.

Are there tools that help programmers find logic errors? Most IDEs include a tool called a **debugger** that helps programmers dissect a program as it runs. The debugger pauses the program while it's executing and allows the programmer to examine the values of all the variables. The programmer can then run the program in slow motion, moving it forward just one line at a time. This lets the programmer see the exact sequence of steps being executed and the outcome of each calculation. He or she can then isolate the precise place in which a logical error occurs, correct the error, and recompile the program.

Testing and Documentation: Finishing the Project

What is the first round of testing for a program? Once debugging has detected all the runtime errors in the code, it's time for users to test the program. This process is called *internal testing*. In internal testing, a group within the software company uses the program in every way it can imagine—including how the program was intended to be used and in ways only new users might think up. The internal testing group makes sure the program behaves as described in the original testing plan. Any differences in how the program responds are reported back to the programming team, which makes the final revisions and updates to the code.

The next round of testing is *external testing*. In this testing round, people like the ones who eventually will purchase and use the software must work with it to determine whether it matches their original vision.

What other testing does the code undergo? Before its final commercial release, software is often provided free or at a reduced cost in a **beta version** to certain test sites or to interested users. By providing users with a beta version of software, programmers can collect information about remaining errors in the code and make a final round of revisions before officially releasing the program. Often, popular software packages like Microsoft Windows and Microsoft Office are available for free beta download for months before the official public release.

What happens if problems are found after beta testing? The manufacturer will make changes before releasing the product to other manufacturers, for installation on new machines, for example. That point in the release cycle is called **release to manufacturers** (or **RTM**). After the RTM is issued, the product is in **general availability** (or **GA**) and can be purchased by the public.

Users often uncover problems in a program even after its commercial release to the public. These problems are addressed with the publication of software updates or **service packs**. Users can download these software modules to repair errors identified in the program code. To make sure you have the latest service pack for your Windows OS, visit the Windows Service Pack Center **(windows.microsoft.com/en-US/windows/downloads/service-packs).**

After testing, is the project finished? Once testing is complete, but before the product is officially released, the work of **documentation** is still ahead. At this point, technical writers create internal documentation for the program that describes the development and technical details of the software, how the code works, and how the user interacts with the program. In addition, the technical publishing department produces all the necessary user documentation that will be distributed to the program's users. User training begins once the software is distributed. Software trainers take the software to the user community and teach others how to use it efficiently. ■

Before moving on to Part 2:

- **Watch Replay Video 10.1** .
- Then check your understanding of what you've learned so far.

check your understanding// review & practice

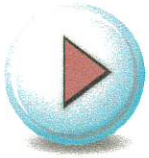
For a quick review to see what you've learned so far, answer the following questions. Visit **pearsonhighered.com /techinaction** to check your answers.

multiple choice

1. Compiling a program turns
   - **a.** top-down design into object-oriented design.
   - **b.** a first-generation language into a fourth-generation language.
   - **c.** source code into an executable program.
   - **d.** none of the above.
2. The step of the SDLC in which we document the transfer of data from the point where it enters the system to its final storage or output is the
   - **a.** problem and opportunity identification phase.
   - **b.** analysis phase.
   - **c.** design phase.
   - **d.** development phase.
3. The life cycle of a program begins with describing a problem and making a plan. Then the PDLC requires
   - **a.** coding, debugging, and testing.
   - **b.** process, input, and output.
   - **c.** data, information, and method.
   - **d.** an algorithm.
4. A visual representation of an algorithm is represented by
   - **a.** an executable.
   - **b.** a compiler.
   - **c.** a debugger.
   - **d.** a flowchart.
5. In object-oriented analysis, classes are defined by their
   - **a.** objects and data.
   - **b.** data and methods.
   - **c.** operators and objects.
   - **d.** behaviors and keywords.

To take an autograded version of this review, please go to the companion website at **pearsonhighered.com/techinaction**, or go your MyITLab course.

Continue

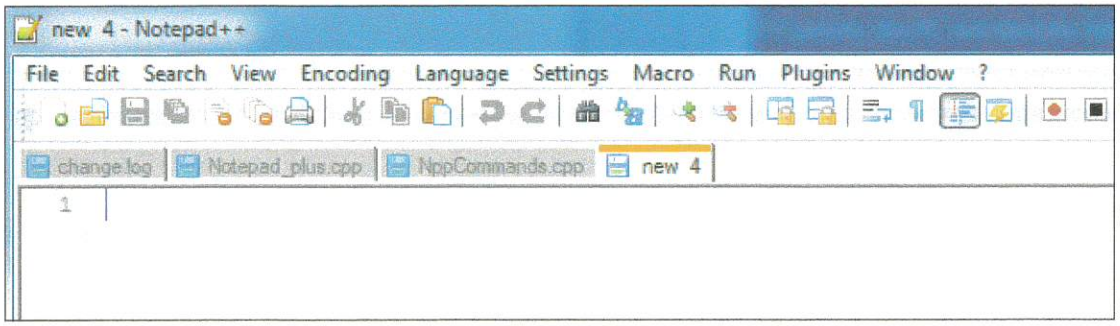
TRY THIS  Programming with Corona

You've already been doing mobile app development in the Make This activities using App Inventor. In this Try This, we'll use the product Corona to create an application that could be run on either an Android or an iOS device.

(Courtesy of Corona Labs, Inc.)

NOTE: You need to be confident in installing and uninstalling demo software to your system to proceed. Refer to Chapter 4, page 149, if you need more support. Also, be aware that you will be asked to provide an e-mail address to qualify for the free trial version of Corona.

Step 1 From **notepad-plus-plus.org**, click **Download** and install the editor Notepad++. Launch Notepad++ and click **File->New**. *(Courtesy of Don Ho)*



(Courtesy of Corona Labs, Inc.)

Step 2 Head to **coronalabs.com/products/corona-sdk**. Select the link for **Download Now** at the bottom of the page, and follow the download instructions.

Step 3 Run the Corona simulator  . Click the **Continue Trial** button.

Step 4 In Notepad++, type the following:

```
myText = display.newText( "TECH IN ACTION", 20, 40, native.systemFont, 36 )
myText:setTextColor( 255,97,3 )
```

Step 5 In Notepad++, now click **File-> Save As**. In the **Save As Type** dropdown, select **LUA source file (*.lua)** and save this file with the name **main**. The file main.lua is now created.