4. Consider a method with this heading:

   ```
   public static void printSqrt(double x)
   ```

   The method prints the square root of x to the standard output. Write a reasonable specification for this method and compare your answer to the solution at the end of the chapter. The specification should forbid a negative argument.

5. Consider a method with this heading:

   ```
   public static double sphereVolume(double radius)
   ```

   The method computes and returns the volume of a sphere with the given radius. Write a reasonable specification for this method and compare your answer to the solution at the end of the chapter. The specification should require a positive radius.

6. Write an if-statement to throw an `IllegalArgumentException` when `x` is less than zero. (Assume that `x` is a `double` variable.)
7. When can a `final` variable be used?
8. How was the specification produced at the start of Figure 1.1?
9. What are the components of a Java signature?
10. Write a Java statement that will print two variables called age and height. The age should be printed as a whole number using ten output spaces; the height should be printed using twelve output spaces and three decimal digits. Label each part of the output as "Age" and "Height." Use Appendix B if necessary.

1.2 RUNNING TIME ANALYSIS

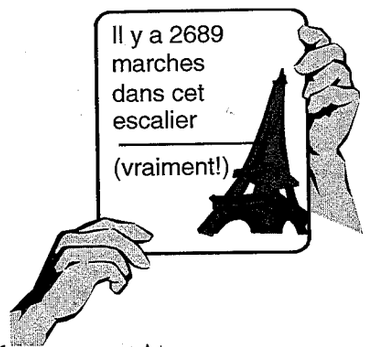
Time analysis consists of reasoning about an algorithm's speed. *Does the algorithm work fast enough for my needs? How much longer does the algorithm take when the input gets larger? Which of several different algorithms is fastest?* These questions can be asked at any stage of software development. Some time analysis is useful to analyze an algorithm before any implementation is done to avoid wasted work of implementing inappropriately slow solutions. Further analysis can be carried out during or after an implementation. This section discusses time analysis, starting with an example that involves no implementation in the usual sense.

The Stair-Counting Problem

Suppose you and your friend Judy are standing at the top of the Eiffel Tower. As you gaze out over the French landscape, Judy turns to you and says, "I wonder how many steps there are to the bottom?" You, of course, are the ever-accommodating host, so you reply, "I'm not sure...but I'll find out." We'll look at three techniques that you could use and analyze the time requirements of each.

Technique 1: Walk Down and Keep a Tally. In the first technique, Judy gives you a pen and a sheet of paper. "I'll be back in a minute," you say as you dash down the stairs. Each time you take a step down, you make a mark on the sheet of paper. When you reach the bottom, you run back up, show Judy the piece of paper, and say, "There are this many steps."

Technique 2: Walk Down, but Let Judy Keep the Tally. In the second technique, Judy is unwilling to let her pen or paper out of her sight. But you are undaunted. Once more you say, "I'll be back in a minute," and you set off down the stairs. But this time you stop after one step, lay your hat on the step, and run back to Judy. "Make a mark on the piece of paper!" you exclaim. Then you run back to your hat, pick it up, take one more step, and lay the hat down on the second step. Then back up to Judy: "Make another mark on the piece of paper!" you say. You run back down the two stairs, pick up your hat, move to the third step, and lay down the hat. Then back up the stairs to Judy: "Make another mark!" you tell her. This continues until your hat reaches the bottom, and you speed back up the steps one more time. "One more mark, please." At this point, you grab Judy's piece of paper and say, "There are this many steps."



Technique 3: Jervis to the Rescue. In the third technique, you don't walk down the stairs at all. Instead, you spot your friend Jervis by the staircase, holding the sign shown here. The translation is *There are 2689 steps in this stairway (really!)*. So you take the paper and pen from Judy, write the number 2689, and hand the paper back to her, saying, "There are this many steps."

This is a silly example, but even so, it does illustrate the issues that arise when performing a time analysis for an algorithm or program. The first issue is deciding exactly how you will measure the time spent carrying out the work or executing the program. At first glance, the answer seems easy: For each of the three stair-counting techniques, just measure the actual time it takes to carry out the work. You could do this with a stopwatch. But there are some drawbacks to measuring actual time. Actual time can depend on various irrelevant details such as whether you or somebody else carried out the work. The actual elapsed time may vary from person to person, depending on how fast each person can run the stairs. Even if we decide that *you* are the runner, the time may vary depending on other factors such as the weather, what you had for breakfast, and what other things are on your mind.

So, instead of measuring the actual elapsed time, we count certain operations that occur while carrying out the work. In this example, we will count two kinds of operations:

1. Each time you walk up or down one step, that is one operation.
2. Each time you or Judy marks a symbol on the paper, that is also one operation.

decide what operations to count

Of course, each of these operations takes a certain amount of time, and making a mark may take a different amount of time than taking a step. But this doesn't concern us because we won't measure the actual time taken by the operations. Instead, we will ask: *How many operations are needed for each of the three techniques?*

For the first technique, you take 2689 steps down, another 2689 steps up, and you also make 2689 marks on the paper, for a total of 3×2689 operations—that is 8067 total operations.

For the second technique, there are also 2689 marks made on Judy's paper, but the total number of operations is much more. You start by going down one step and back up one step. Then down two and up two. Then down three and up three, and so forth. The total number of operations taken is shown below.

Downward steps	= 3,616,705 (which is $1+2+\ldots+2689$)
Upward steps	= 3,616,705
Upward steps	= 3,616,705
Total operations	= Downward steps + Upward steps + Marks made = 7,236,099

The third technique is the quickest of all: Only four marks are made on the paper (that is, we're counting one "mark" for each digit of 2689), and there is no going up and down stairs. The number of operations used by each of the techniques is summarized here:

Technique 1	8067	operations
Technique 2	7,236,099	operations
Technique 3	4	operations

Doing a time analysis for a program is similar to the analysis of the stair-counting techniques. For a time analysis of a program, we do not usually measure the actual time taken to run the program because the number of seconds can depend on too many extraneous factors—such as the speed of the processor and whether the processor is busy with other tasks. Instead, the analysis counts the number of operations required. There is no precise definition of what constitutes an **operation**, although an operation should satisfy your intuition of a "small step." An operation can be as simple as the execution of a single program statement. Or we could use a finer notion of operation that counts each arithmetic operation (addition, multiplication, etc.) and each assignment to a variable as a separate operation.

dependence on input size

For most programs, the number of operations depends on the program's input. For example, a program that sorts a list of numbers is quicker with a short list than with a long list. In the stairway example, we can view the Eiffel Tower as the input to the problem. In other words, the three different techniques all work on the Eiffel Tower, but the techniques also work on Toronto's CN Tower or on the stairway to the top of the Statue of Liberty or on any other stairway.

When a time analysis depends on the size of the input, then the time can be given as an expression, where part of the expression is the input's size. The time expressions for our three stair-counting techniques are:

Technique 1	$3n$
Technique 2	$n + 2\ (1+2+\ldots+n)$
Technique 3	The number of digits in the number n

The expressions in this list give the number of operations performed by each technique when the stairway has n steps.

The expression for the second technique is not easy to interpret. It needs to be simplified to become a formula that we can easily compare to other formulas. So let's simplify it. We start with the following subexpression:

$$(1+2+\ldots+n)$$

simplification of the time analysis for Technique 2

There is a trick that will enable us to find a simplified form for this expression. The trick is to *compute twice the amount of the expression and then divide the result by 2*. Unless you've seen this trick before, it sounds crazy. But it works fine. The trick is illustrated in Figure 1.2. Let's go through the computation of that figure step by step.

We write the expression $(1+2+\ldots+n)$ twice and add the two expressions. But as you can see in Figure 1.2, we also use another trick: When we write the expression twice, *we write the second expression backward.* After we write down the expression twice, we see the following:

$$\begin{aligned}&(1+2+\ldots+n)\\ &+(n+\ldots+2+1)\end{aligned}$$

We want the sum of the numbers on these two lines. That will give us twice the value of $(1+2+\ldots+n)$, and we can then divide by 2 to get the correct value of the subexpression $(1+2+\ldots+n)$.

Now, rather than proceeding in the most obvious way, we instead add pairs of numbers from the first and second lines. We add the 1 and the n to get $n+1$. Then we add the 2 and the $n-1$ to again get $n+1$. We continue until we reach the last pair consisting of an n from the top line and a 1 from the bottom line. All the pairs add up to the same amount, namely $n+1$. Now that is handy! We get n numbers, and all the numbers are the same, namely $n+1$. So the total of all the numbers on the preceding two lines is:

$$n(n+1)$$

The value of twice the expression is n multiplied by $n+1$. We are now essentially done. The number we computed is twice the quantity we want. So, to obtain our simplified formula, we only need to divide by 2. The final simplification is thus:

$$(1+2+\ldots+n) = \frac{n(n+1)}{2}$$

We will use this simplified formula to rewrite the Technique 2 time expression, but you'll also find that the formula occurs in many other situations. The simplification for the Technique 2 expression is as follows:

Number of operations for Technique 2

$= n + 2(1+2+\ldots+n)$

$= n + 2\left(\frac{n(n+1)}{2}\right)$ *Plug in the formula for* $(1+2+\ldots+n)$

$= n + n(n+1)$ *Cancel the* 2*s*

$= n + n^2 + n$ *Multiply out*

$= n^2 + 2n$ *Combine terms*

So, Technique 2 requires $n^2 + 2n$ operations.

FIGURE 1.2 Deriving a Handy Formula

$(1+2+\ldots+n)$ can be computed by first computing the sum of twice $(1+2+\ldots+n)$, as shown here:

$$\begin{array}{rcccccc} & 1 & + & 2 & +\ldots+ & (n-1) & + & n \\ + & n & + & (n-1) & +\ldots+ & 2 & + & 1 \\ \hline & (n+1) & + & (n+1) & +\ldots+ & (n+1) & + & (n+1) \end{array}$$

The sum is $n(n+1)$, so $(1+2+\ldots+n)$ is half this amount:

$$(1+2+\ldots+n) = \frac{n(n+1)}{2}$$

simplification of time analysis for Technique 3

The number of operations for Technique 3 is just the number of digits in the integer n when written down in the usual way. The usual way of writing down numbers is called **base 10 notation**. As it turns out, the number of digits in a number n, when written in base 10 notation, is approximately equal to another mathematical quantity known as the **base 10 logarithm** of n. The notation for the base 10 logarithm of n is written:

$$\log_{10} n$$

base 10 notation and base 10 logarithms

The base 10 logarithm does not always give a whole number. For example, the actual base 10 logarithm of 2689 is about 3.43 rather than 4. If we want the actual number of digits in an integer n, we need to carry out some rounding. In particular, the exact number of digits in a positive integer n is obtained by rounding $\log_{10} n$ downward to the next whole number and then adding 1. The notation for rounding down and adding 1 is obtained by adding some marks to the logarithm notation as follows:

$$\lfloor \log_{10} n \rfloor + 1$$

This is all fine if you already know about logarithms, but what if some of this is new to you? For now, you can simply define the notation to mean *the number of digits in the base 10 numeral for n*. You can do this because if others use any of the other accepted definitions for this formula, they will get the same answers that you do. You will be right! (And they will also be right.) In Section 10.3 of this book, we will show that the various definitions of the logarithm function are all equivalent. For now, we will not worry about all that detail. We have larger issues to discuss first. The table of the number of operations for each technique can now be expressed more concisely as shown here:

Technique 1	$3n$
Technique 2	$n^2 + 2n$
Technique 3	$\lfloor \log_{10} n \rfloor + 1$

Big-*O* Notation

The time analyses we gave for the three stair-counting techniques were very precise. They computed the exact number of operations for each technique. But such precision is sometimes not needed. Often it is enough to know in a rough manner how the number of operations is affected by the input size. In the stair example, we started by thinking about a particular tower, the Eiffel Tower, with a particular number of steps. We expressed our formulas for the operations in terms of n, which stood for the number of steps in the tower. Now suppose we apply our various stair-counting techniques to a tower with 10 times as many steps as the Eiffel Tower. If n is the number of steps in the Eiffel Tower, then

this taller tower will have $10n$ steps. The number of operations needed for Technique 1 on the taller tower increases tenfold (from $3n$ to $3 \times (10n) = 30n$); the time for Technique 2 increases approximately 100-fold (from about n^2 to about $(10n)^2 = 100n^2$); and Technique 3 increases by only one operation (from the number of digits in n to the number of digits in $10n$, or to be very concrete, from the four digits in 2689 to the five digits in 26,890). We can express this kind of information in a format called **big-*O* notation**. The symbol O in this notation is the letter O, so big-O is pronounced "big Oh."

We will describe three common examples of the big-O notation. In these examples, we use the notion of "the largest term in a formula." Intuitively, this is the term with the largest exponent on n or the term that grows the fastest as n itself becomes larger. For now, this intuitive notion of "largest term" is enough.

quadratic time $O(n^2)$

Quadratic Time. If the largest term in a formula is no more than a constant times n^2, then the algorithm is said to be "**big-*O* of n^2**," written $O(n^2)$, and the algorithm is called **quadratic**. In a quadratic algorithm, doubling the input size makes the number of operations increase approximately fourfold (or less). For a concrete example, consider Technique 2, requiring $n^2 + 2n$ operations. A 100-step tower requires 10,200 operations (that is, $100^2 + 2 \times 100$). Doubling the tower to 200 steps increases the time approximately fourfold, to 40,400 operations (that is, $200^2 + 2 \times 200$).

linear time $O(n)$

Linear Time. If the largest term in the formula is a constant times n, then the algorithm is said to be "**big-*O* of n**," written $O(n)$, and the algorithm is called **linear**. In a linear algorithm, doubling the input size makes the time increase approximately twofold (or less). For example, a formula of $3n + 7$ is linear, so $3 \times 200 + 7$ is about twice $3 \times 100 + 7$.

logarithmic time $O(\log n)$

Logarithmic Time. If the largest term in the formula is a constant times a logarithm of n, then the algorithm is "**big-*O* of the logarithm of n**," written $O(\log n)$, and the algorithm is called **logarithmic**. (The base of the logarithm may be base 10 or possibly another base. We'll talk about the other bases in Section 10.3.) In a logarithmic algorithm, doubling the input size will make the time increase by no more than a fixed number of new operations, such as one more operation or two more operations—or in general by c more operations, where c is a fixed constant. For example, Technique 3 for stair counting has a logarithmic time formula. And doubling the size of a tower (perhaps from 500 stairs to 1000 stairs) never requires more than one extra operation.

Using big-O notation, we can express the time requirements of our three stair-counting techniques as follows:

Technique 1	$O(n)$
Technique 2	$O(n^2)$
Technique 3	$O(\log n)$

FIGURE 1.3 Number of Operations for Three Techniques

Number of stairs (n)	Logarithmic $O(\log n)$ Technique 3, with $\lfloor \log_{10} n \rfloor + 1$ operations	Linear $O(n)$ Technique 1, with $3n$ operations	Quadratic $O(n^2)$ Technique 2, with $n^2 + 2n$ operations
10	2	30	120
100	3	300	10,200
1000	4	3000	1,002,000
10,000	5	30,000	100,020,000

order of an algorithm

When a time analysis is expressed with big-O, the result is called the **order** of the algorithm. We want to reinforce one important point: Multiplicative constants are ignored in the big-O notation. For example, both $2n$ and $42n$ are linear formulas, so both are expressed as $O(n)$, ignoring the multiplicative constants 2 and 42. As you can see, this means that a big-O analysis loses some information about relative times. Nevertheless, a big-O analysis does provide some useful information for comparing algorithms. The stair example illustrates the most important kind of information provided by the order of an algorithm.

> The order of an algorithm generally is more important than the speed of the processor.

For example, using the quadratic technique (Technique 2), the fastest stair climber in the world is still unlikely to do better than a slowpoke—provided that the slowpoke uses one of the faster techniques. In an application such as sorting a list, a quadratic algorithm can be impractically slow on even moderately sized lists, regardless of the processor speed. To see this, notice the comparisons showing actual numbers for our three stair-counting techniques, which are shown in Figure 1.3.

Time Analysis of Java Methods

The principles of the stair-climbing example can be applied to counting the number of operations required by code written in a high-level language such as Java. As an example, consider the method implemented in Figure 1.4 on page 24. The method searches through an array of numbers to determine whether a particular number occurs.

FIGURE 1.4 Specification and Implementation of a `search` Method

Specification

♦ **search**

`public static boolean search(double[ ] data, double target)`

Notice that there is no precondition.

Search an array for a specified number.

Parameters:

`data` – an array of double numbers in no particular order
`target` – a specific number that we are searching for

Returns:

`true` (to indicate that `target` occurs somewhere in the array)
or `false` (to indicate that `target` does not occur in the array)

Implementation

```
public static boolean search(double[ ] data, double target)
{
   int i;

   for (i = 0; i < data.length; i++)
   {  // Check whether the target is at data[i].
      if (data[i] == target)
         return true;
   }

   // The loop finished without finding the target.
   return false;
}
```

Examples:

Suppose that the data array has the five numbers {2, 14, 6, 8, 10}. Then search(data, 10) *returns true, but* search(data, 42) *returns false.*

for our first analysis, the number that we are searching for does not occur in the array

As with the stair-climbing example, the first step of the time analysis is to decide precisely what we will count as a single operation. For Java, a good choice is to count the total number of Java operations (such as an assignment, an arithmetic operation, or the < comparison). If a method calls other methods, we would also need to count the operations that are carried out in the other methods.

With this in mind, let's do a first analysis of the `search` method for the case in which the array's length is a non-negative integer n and (just to be difficult) the number that we are searching for does not occur in the array. How many operations does the search method carry out in all? Our analysis has three parts:

1. When the for-loop starts, there are two operations: an assignment to initialize the variable `i` to 0, and an execution of the test to determine whether `i` is less than `data.length`.
2. We then execute the body of the loop, and because the number that we are searching for does not occur, we will execute this body n times. How many operations occur during each execution of the loop body? We could count this number, but let's just say that each execution of the loop body requires k operations, where k is some number around 3 or 4 (including the work at the end of the loop where `i` is incremented and the termination test is executed). If necessary, we'll figure out k later, but for now it is enough to know that we execute the loop body n times and each execution takes k operations, for a total of kn operations.
3. After the loop finishes, there is one more operation (a return statement).

The total number of operations is now $kn + 3$. The +3 is from the two operations before the loop and the one operation after the loop. Regardless of how big k is, this formula is always linear time. So, in the case where the sought-after number does not occur, the search method takes linear time. In fact, this is a frequent pattern that we summarize here:

Frequent Linear Pattern

A loop that does a fixed amount of operations n times requires $O(n)$ time.

Later you will see additional patterns, resulting in quadratic, logarithmic, and other times. In fact, in Chapter 11 you will rewrite the `search` method in a way that uses an array that is sorted from smallest to largest but requires only logarithmic time.

Worst-Case, Average-Case and Best-Case Analyses

The `search` method has another important feature: For any particular array size n, the number of required operations can differ depending on the exact parameter values. For example, with n equal to 100, the target could be 27, and the very first array element could also be 27—so the loop body executes just one time. On the other hand, maybe the number 27 doesn't occur until `data[99]` and the loop body executes the maximum number of times (n times). In other words, for any fixed n, different possible parameter values result in a different number of operations. When this occurs, we usually count the *maximum* number of required operations for inputs of a given size. Counting the maximum number of operations is called the **worst-case** analysis. In fact, the worst case for the `search` method occurs when the sought-after number is not in the array, which is the reason why we used the "not in array" situation in our previous analysis.

worst-case analysis

During a worst-case time analysis, you may sometimes find yourself unable to provide an exact count of the number of operations. If the analysis is a worst-case analysis, you may estimate the number of operations, always making your

estimate on the high side. In other words, the actual number of operations must be guaranteed to be less than the estimate that you use in the analysis.

In Chapter 11, when we begin the study of searching and sorting, you'll see two other kinds of time-analysis: **average-case** analysis, which determines the average number of operations required for a given n, and **best-case** analysis, which determines the fewest number of operations required for a given n.

Self-Test Exercises for Section 1.2

11. Write code for a method that computes the sum of all the numbers in an integer array. If the array's length is zero, the sum should also be zero. Do a big-O time analysis of your method.
12. Each of the following are formulas for the number of operations in some algorithm. Express each formula in big-O notation.
    a. $n^2 + 5n$
    b. $3n^2 + 5n$
    c. $(n + 7)(n - 2)$
    d. $100n + 5$
    e. $5n + 3n^2$
    f. The number of digits in $2n$
    g. The number of times that n can be divided by 10 before dropping below 1.0
13. Determine which of the following formulas is $O(n)$:
    a. $16n^3$
    b. $n^2 + n + 2$
    c. $\lfloor n^2/2 \rfloor$
    d. $10n + 25$
14. What is meant by *worst-case analysis*?
15. What is the worst-case big-O analysis of the following code fragment?

```
for (i = 0; i < n; ++i) {
    for (j = i; j < n; ++j) {
        j += n;
    }
}
```

16. List the following formulas in order of running time analysis, from least to greatest time requirements, assuming that n is very large:
    $n^2 + 1$; $50 \log n$; 1,000,000; $10n + 10{,}000$.

1.3 TESTING AND DEBUGGING

Always do right. This will gratify some people, and astonish the rest.

MARK TWAIN
To the Young People's Society, February 16, 1901

program testing

Program testing occurs when you run a program and observe its behavior. Each time you execute a program using some input, you are testing to see how the program works for that particular input, and you are also testing to see how long the program takes to complete. The topic of this section is the construction of test inputs that are likely to discover errors.

Choosing Test Data

To serve as good test data, your test inputs need two properties.

Properties of Good Test Data

1. You must know what output a correct program should produce for each test input.
2. The test inputs should include those inputs that are most likely to cause errors.

Do not take the first property lightly—you must choose test data for which you know the correct output. Just because a program compiles, runs, and produces output that looks about right does not mean the program is correct. If the correct answer is 3278 and the program outputs 3277, then something is wrong. How do you know the correct answer is 3278? The most obvious way to find the correct output value is to work it out with pencil and paper using some method other than that used by the program. To aid this, you might choose test data for which it is easy to calculate the correct answer, perhaps by using smaller input values or by using input values for which the answer is well known.

Boundary Values

We will focus on two approaches for finding test data that are most likely to cause errors. The first approach is based on identifying and testing inputs called *boundary values*, which are particularly apt to cause errors. A **boundary value** of a problem is an input that is one step away from a different kind of behavior. For example, recall the `dateCheck` method from the first self-test exercise on page 15. It has the following precondition:

Precondition:
The three arguments are a legal year, month, and day of the month in the years 1900 to 2099.

Two boundary values for `dateCheck` are January 1, 1900 (since one step below this date is illegal) and December 31, 2099 (since one step above this date is illegal). If we expect the method to behave differently for "leap days," we should try days such as February 28, 2000 (just before a leap day); February 29, 2000 (a leap day); and March 1, 2000 (just after a leap day).

test zero as a boundary value

Frequently zero has special behavior, so it is a good idea to consider zero to be a boundary value whenever it is a legal input. For example, consider the `search` method from Figure 1.4 on page 24. This method should be tested with a data array that contains no elements (`data.length` is 0). For example:

```
double[ ] EMPTY = new double[0]; // An array with no elements

// Searching the EMPTY array should always return false.
if (search(EMPTY, 0))
   System.out.println("Wrong answer for an empty array.");
else
   System.out.println("Right answer for an empty array.");
```

test 1 and –1 as boundary values

The numbers 1 and –1 also have special behavior in many situations, so they should be tested as boundary values whenever they are legal input. For example, the `search` method should be tested with an array that contains just one element (`data.length` is 1). In fact, it should be tested twice with a one-element array: once when the target is equal to the element and once when the target is different from the element.

boundary values are "one step away from different behavior"

In general, there is no precise definition of a boundary value, but you should develop an intuitive feel for finding inputs that are "one step away from different behavior."

Test Boundary Values

If you cannot test all possible inputs, at least test the boundary values. For example, if legal inputs range from zero to one million, be sure to test input 0 and input 1000000. It is a good idea also to consider 0, 1, and –1 to be boundary values whenever they are legal input.

Fully Exercising Code

The second widely used testing technique requires intimate knowledge of how a program has been implemented. The technique, called **fully exercising code**, is stated with two rules:.

Fully Exercising Code

1. Make sure that each line of your code is executed at least once by some of your test data. Make sure that this rare situation is included among your set of test data.
2. If there is part of your code that is sometimes skipped altogether, make sure there is at least one test input that actually does skip this part of your code. For example, there might be a loop where the body is sometimes executed zero times. Make sure that there is a test input that causes the loop body to be executed zero times.

profiler

Some Java programming environments have a software tool called a **profiler** to help fully exercise code. A typical profiler will generate a listing indicating how many times each method was called, so you can easily check that each method has been executed at least once. Some profilers offer more complete information, telling how often each individual statement of your program was executed. This can help you spot parts of your program that were not tested.

Keep in mind that fully exercised code may still have bugs. Code that has been fully exercised has had each line of code tested at least once, but one test does not guarantee that there are no errors.

PITFALL

Avoid Impulsive Changes

Finding a test input that causes an error is only half the problem of testing and debugging. After an erroneous test input is found, you still must determine exactly why the "bug" occurs and then "debug the program." When you have found an error, there is an impulse to dive right in and start changing code. It is tempting to look for suspicious parts of your code and change these suspects to something "that might work better."

Avoid the temptation.

Impulsively changing suspicious code almost always makes matters worse. Instead, you must discover exactly why a test case is failing and limit your changes to corrections of known errors. Once you have corrected a known error, all test cases should be rerun.

Using a Debugger

Tracking down the reason why a test case is failing can be difficult. For large programs, tracking down errors is nearly impossible without the help of a software tool called a **debugger**. A debugger executes your code one line at a time, or it may execute your code until a certain condition arises. Using a debugger, you can specify what conditions should cause the program execution to pause. You can also keep a continuous watch on the location of the program execution and on the values of specified variables.

Assert Statements

> *An early advocate of using assertions in programming was none other than Alan Turing himself. On 24 June 1950 at a conference in Cambridge, he gave a short talk entitled* Checking a Large Routine, *which explains the idea with great clarity: "How can one check a large routine in the sense that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."*
>
> C.A.R. HOARE
> 1980 Turing Award Lecture

Assert statements (also called **assertions**) are boolean expressions that can be checked for validity while a program is running. They were introduced in Java 2, Version 1.4. Assertions can assist in debugging and maintaining programs by documenting conditions that the programmer intends to be valid at particular locations in the program. When a program is running, invalid assertions can be automatically flagged by the Java Runtime Environment, allowing a programmer to spot potential problems as early as possible.

The statement uses the new keyword `assert`, usually following the pattern shown here:

```
assert ____________________ : "____________________";
```

This is a boolean expression that we want to make sure is true at this point in the program.

This is an error message that will be generated if the boolean expression is false.

When an assert statement is reached, the boolean expression is tested. If the expression is true, then no action is taken: The program merely continues executing. But if the expression is false, then an exception called `AssertionError` is thrown. When an `AssertionError` occurs, the method where the failure occurred will stop its computation and usually the program will stop, printing the indicated error message along with an indication of the line number where the `AssertionError` occurred.

For example, consider the `maxOf3` method in Figure 1.5. The computation finds the largest of three numbers; it could be done in many different ways such as the if-statements we have written, or perhaps we could make use of the two-argument `max` method from `java.lang.math`. But however the computation is carried out, it's an easy matter to use assertions to check the validity of the answer with the highlighted assertions at the bottom of the figure.

Notice that the error message (which follows the colon in each assertion) does not need to be on the same line as the assertion's boolean expression. Together, the two assertions verify that the answer was computed correctly—or, if there was an error, the program will stop and print an error message to guide the programmer's debugging effort. This particular example uses the "or" operation (`||`) to ensure that the answer is one of the three original parameters, and it uses the "and" operation (&&) to ensure that the answer is not smaller than any of the parameters.

Turning Assert Statements On and Off

By using assertions at key points (particularly for postconditions), a programmer finds programming errors at an early point when the errors are often easier to correct. However, once a program is ready to release for public use, Java environments permit assertion checking to be turned on or off as needed.

When a program is run with the Java 2, Version 1.5 runtime system, assertions are *not* normally checked. During debugging, however, a programmer can turn on assertion checking by using the `-enableassertions` option (or `-ea`) for the Java runtime system. For example:

```
java -ea TemperatureConversion
```

There are other options that permit the programmer to turn on or off assertions in specific locations, but the general `-ea` option is sufficient to start.

FIGURE 1.5 Two Examples of Assertions

Specification

♦ **maxOf3**

`public static int maxOf3(int a, int b, int c)`

Returns the largest of three `int` values.

Parameters:

`a, b, c` – any `int` numbers

Returns:

The return value is the largest of the three arguments `a`, `b` and `c`.

Implementation

```
public static int maxOf3(int a, int b, int c)
{
   int answer;

   // Set answer to the largest of a, b and c:
   answer = a;        // Initially set answer to a
   if (b > answer)    // Maybe change the answer to b
      answer = b;
   if (c > answer)    // Maybe change the answer to c
      answer = c;

   // Check that the computation did what we expected:
   assert (answer == a) || (answer == b) || (answer == c)
      : "maxOf3 answer is not equal to one of the arguments";
   assert (answer >= a) && (answer >= b) && (answer >= c)
      : "maxOf3 answer is not equal to the largest argument";

   return answer;
}
```

PROGRAMMING TIP

USE A SEPARATE METHOD FOR COMPLEX ASSERTIONS

Some assertions can be implemented with a small boolean expression, such as the two assertions in maxOf3. But when the necessary checking becomes more complex, you should write a separate private method (or several methods) that carry out the checking. The method should return a boolean value that can be used in the assertion, as shown in Figure 1.6. Notice that the boolean methods are private (there is no `public` keyword), which means they can be used only within the class that they appear.

FIGURE 1.6 The `maxOfArray` Method Together with Methods to Implement Assertions

Specification

♦ **maxOfArray**

`public static int maxOfArray(int[ ])`

Returns the largest value in an array.

Parameters:

`a` – a non-empty array (the length must not be zero)

Precondition:

`a.length > 0`.

Returns:

The return value is the largest value in the array a.

Throws: `ArrayIndexOutOfBounds`

Indicates that the array length is zero.

Implementation

```
// This private method checks to make sure that the specified value is contained somewhere
// in the array a. The return value is true if the value is found; otherwise, the return value
// is false.
static boolean contains(int[ ] a, int value)
{
   int i;

   for (i = 0; i < a.length; i++)
   {
      if (a[i] == value)
         return true;
   }

   // The loop finished without finding the specified value, so we return false:
   return false;
}
```

(continued)

(FIGURE 1.6 continued)

```
// This private method checks to make sure that the specified value is greater than or equal to
// every element in the array a. In this case, the method returns true. On the other hand, if the
// specified value is less than some element in the array, then the method returns false.
static boolean greaterOrEqual(int[ ] a, int value)
{
   int i;

   for (i = 0; i < a.length; i++)
   {
      if (a[i] > value)
         return false;
   }

   // The loop finished without finding an array element that exceeds the value,
   // so we can return true:
   return true;
}

public static int maxOfArray(int[ ] a)
{
   int answer;
   int i;

   // Set answer to the largest value in the array.
   answer = a[0];        // Initially set answer to the first element.
   for (i = 1; i < a.length; i++)
   {
      if (a[i] > answer)  // Maybe change the answer to a[i]
         answer = a[i];
   }

   // Check that the computation did what we expected:
   assert contains(a, answer)
      : "maxOfArray answer is not equal in the array";
   assert greaterOrEqual(a, answer)
      : "maxOfArray answer is less than an array element";

   return answer;
}
```

PITFALL

AVOID USING ASSERTIONS TO CHECK PRECONDITIONS

Because assertions can be turned on and off, most programmers will not use assertions to check preconditions in public methods. Why not? The problem is that public methods may be used by other programmers. Of course, it is the responsibility of that other programmer to ensure that the precondition is valid. Still, if an invalid precondition is detected, it is best to throw an exception that cannot be turned off. Therefore, the designers of the assert statement suggest that assert statements should not be used to check preconditions in public methods.

Static Checking Tools

Static checking refers to program verification that can be carried out before a program is running. The Java compiler does certain kinds of static checking, such as checking that some data types are correct. (You cannot assign a String value to a char variable.) Other tools are available to do more extensive static checking, including verification of certain kinds of assertions. One such tool is the *Extended Static Checker for Java (ESC/Java)*, developed at the Compaq Systems Research Center and first shown to me by a colleague, Jim Royer, from Syracuse University. The tool is available to download for research and educational use at `http://research.compaq.com/SRC/esc/`

Self-Test Exercises for Section 1.3

17. Suppose you write a program that accepts as input any integer in the range –20 through 20 and then outputs the number of digits in the input integer. What boundary values should you use as test inputs?
18. Suppose you write a program that accepts a single line as input and then outputs a message telling whether or not the line contains the letter A and whether or not it contains more than three A's. What is a good set of test inputs?
19. What does it mean to "fully exercise" code?
20. Suppose you have written a method with two double-number parameters: `x` and `epsilon`. The precondition requires that both parameters be actual numbers (rather than the special values such as "infinity"). The return value from the function is a number `z` so that `z*z*z` is no more than `epsilon` away from `x`. Don't worry about exactly how you've written this method; just assume that you wrote it. For this exercise, write an assertion that can be added at the end of the method to check that the return value is correct. Assume that the method has not altered `x` and `epsilon`, and use `java.math.abs` to compute an absolute value.
21. Why are assertions not usually used to check preconditions of public methods?

CHAPTER SUMMARY

- The first step in producing a program is to write down a precise description of what the program is supposed to do.
- *Pseudocode* is a mixture of Java (or some other programming language) and English (or some other natural language). Pseudocode is used to express algorithms so that you are not distracted by details about Java syntax.
- One good method for specifying what a method is supposed to do is to provide a *precondition* and *postcondition* for the method. These form a contract between the programmer who uses the method and the programmer who writes the method.
- *Time analysis* is an analysis of how many operations an algorithm requires. Often, it is sufficient to express a time analysis in big-O notation, which is the *order* of an algorithm. The order analysis is often enough to compare algorithms and estimate how running time is affected by changing input size.
- Three important examples of big-O analyses are *linear* (i.e., $O(n)$), *quadratic* (i.e., $O(n^2)$), and *logarithmic* (i.e., $O(\log n)$).
- An important testing technique is to identify and test *boundary values*. These are values that lie on a boundary between different kinds of behavior for your program.
- A second testing technique is to ensure that test cases are *fully exercising* the code. A software tool called a *profiler* can aid in fully exercising code.
- During debugging, you should discover exactly why a test case is failing and limit your changes to corrections of known errors. Once you have corrected a known error, all test cases should be rerun. Use a software tool called a *debugger* to help track down exactly why an error occurs.
- Assertions can assist in debugging and maintaining programs by documenting conditions that the programmer intends to be valid at particular locations in the program