CHAPTER 8

Recursive Thinking

LEARNING OBJECTIVES

When you complete Chapter 8, you will be able to...

- recognize situations in which a subtask is nothing more than a simpler version of the larger problem and design recursive solutions for these problems.
- trace recursive calls by drawing pictures of the runtime stack.
- prove that a recursive method has no infinite recursion by finding a valid variant expression and threshold.
- use induction to prove that a recursive method meets its precondition/postcondition contract.

CHAPTER CONTENTS

8.1 Recursive Methods
8.2 Studies of Recursion: Fractals and Mazes
8.3 Reasoning About Recursion
Chapter Summary
Solutions to Self-Test Exercises
Programming Projects

CHAPTER

8 Recursive Thinking

"Well," said Frog, "I don't suppose anyone ever is completely self-winding. That's what friends are for." He reached for the father's key to wind him up again.

RUSSELL HOBAN
The Mouse and His Child

one of the subtasks is a simpler version of the same problem you are trying to solve in the first place

Often, during top-down design, you'll meet a remarkable situation: One of the subtasks to be solved is nothing more than a simpler version of the same problem you are trying to solve in the first place. In fact, this situation occurs so frequently that experienced programmers start *expecting* to find simpler versions of a large problem during the design process. This expectation is called **recursive thinking**. Programming languages, such as Java, support recursive thinking by permitting a method implementation to actually activate itself. In such cases, the method is said to use **recursion**.

In this chapter, we start encouraging you to recognize situations in which recursive thinking is appropriate. We also discuss recursion in Java, both how it is used to implement recursive designs and the mechanisms that occur during the execution of a recursive method.

8.1 RECURSIVE METHODS

We'll start with an example. Consider the task of writing a non-negative integer to the screen with its decimal digits stacked vertically. For example, the number `1234` should be written as:

```
1
2
3
4
```

a case with an easy solution . . . and a case that needs more work

One version of this problem is quite easy: If the integer has only one digit, then we can just print that digit. But if the number has more than one digit, the solution is not immediately obvious, so we might break the problem into two subtasks: (1) print all digits except the last digit, stacked vertically; (2) print the last digit. For example, if the number is `1234`, then the first step will write:

```
1
2
3
```

The second step will output the last digit, `4`.

The main factor that influenced our selection of these two steps is the ease of providing the necessary data. For example, with our input number 1234, the first step needs the digits of 123, which is easily expressed as 1234/10 (since dividing an integer by 10 results in the quotient, with any remainder discarded). In general, if the integer is called number, and number has more than one digit, then the first step prints the digits of number/10, stacked vertically. The second step is equally easy: It requires us to print the last digit of number, which is easily expressed as number % 10. (This is the remainder upon dividing number by 10.) Simple expressions, such as number/10 and number % 10, are not so easy to find for other ways of breaking down the problem (such as printing only the first digit in the first step).

The pseudocode for our solution is as follows:

```
// Printing the digits of a non-negative number, stacked vertically
if (the number has only one digit)
     Write that digit.
else
{
     Write the digits of number/10 stacked vertically.
     Write the single digit of number % 10.
}
```

one of the steps is a simpler instance of the same task

At this point, your recursive thinking cap should be glowing: One of the steps—*write the digits of* number/10 *stacked vertically*—is a simpler instance of the same task of writing a number's digits vertically. It is simpler because number/10 has one fewer digit than number. This step can be implemented by activating writeVertical itself. This is an example of a method activating itself to solve a smaller problem, which is a **recursive call**. The implementation, with a recursive call, is shown in Figure 8.1. This implementation is a static method that could be part of any class. Within this class itself, the static method can be activated by the simple name writeVertical(...). From outside the class, the method can be activated by giving the class name followed by ".writeVertical(...)".

In a moment, we'll look at the exact mechanism that occurs during the activation of a recursive method, but first there are two notions to explain.

1. The Stopping Case. If the problem is simple enough, it is solved without recursive calls. In writeVertical, this occurs when number has only one digit. The case without any recursion is called the **stopping case** or **base case**. In Figure 8.1, the stopping case of writeVertical is implemented with the two lines:

```
if (number < 10)
   System.out.println(number); // Write the one digit.
```

2. The Recursive Call. In Figure 8.1, the method writeVertical makes a recursive call. The recursive call is the highlighted statement at the top of the next page.

```
else
{
   writeVertical(number/10);   // Write all but the last digit.
   System.out.println(number % 10); // Write the last digit.
}
```

This is an instance of the `writeVertical` method activating itself to solve the simpler problem of writing all but the last digit.

FIGURE 8.1 The `writeVertical` Method

Specification

- **writeVertical**

  `public static void writeVertical(int number)`

  Print the digits of a non-negative integer vertically.

  Parameters:

  `number` - the number to be printed

  Precondition:

  `number >= 0`

  The method does not check the precondition, but the behavior is wrong for negative numbers.

  Postcondition:

  The digits of `number` have been written, stacked vertically.

Implementation

```
public static void writeVertical(int number)
{
   if (number < 10)
      System.out.println(number);          // Write the one digit.
   else
   {
       writeVertical(number/10);           // Write all but the last digit.
       System.out.println(number % 10);    // Write the last digit.
   }
}
```

Sample Results of `writeVertical(1234)`

```
1
2
3
4
```

Tracing Recursive Calls

During a computation such as `writeVertical(3)`, what actually occurs? The first step is that the argument `3` is copied to the method's formal parameter, `number`. This is the way that all primitive types are handled as a parameter: The argument provides an initial value for the formal parameter.

Once the formal parameter has its initial value, the method's code is executed. Since `3` is less than `10`, the boolean expression in the if-statement is `true`. So, in this case, it is easy to see that the method just prints `3` and does no more work.

Next, let's try an argument that causes us to enter the else-part, for example:

```
writeVertical(37);
```

When the method is activated, the value of `number` is set equal to `37`, and the code is executed. Since `37` is not less than `10`, the two statements of the else-part are executed. Here is the first statement:

```
writeVertical(number/10);   // Write all but the last digit.
```

example of a recursive call

In this statement, `(number/10)` is `(37/10)`, which is `3`. So this is an activation of `writeVertical(3)`. We already know the action of `writeVertical(3)`: Print `3` on a single line of output. After this activation of `writeVertical` is completely finished, the second line in the else-part executes:

```
System.out.println(number % 10);
```

This just writes `number % 10`. In our example, `number` is `37`, so the statement prints the digit `7`. The total output of the two lines in the else-part is:

```
3
7
```

The method `writeVertical` uses recursion. Yet we did nothing new or different in carrying out the computation of `writeVertical(37)`. We treated it just like any other method. We simply substituted the actual argument for `number` and then executed the code. When the computation reached the recursive call of `writeVertical(3)`, we simply activated `writeVertical` with the argument 3.

PROGRAMMING EXAMPLE: An Extension of writeVertical

Figure 8.2 shows an extension of `writeVertical` to a more powerful method called `superWriteVertical`, which handles all integers including negative integers. With a negative input, the new method prints the negative sign on the first line of output, above any of the digits. For example:

```
superWriteVertical(-361)
```

FIGURE 8.2 The `superWriteVertical` Method

Specification

- **superWriteVertical**
  `public static void superWriteVertical(int number)`
  Print the digits of an integer vertically.
  Parameters:
  `number` – the number to be printed
  Postcondition:
  The digits of `number` have been written, stacked vertically. If `number` is negative, then a negative sign appears on top.

Implementation

```
public static void superWriteVertical(int number)
{
   if (number < 0)
   {
      System.out.println("-");                  // Print a negative sign.
      superWriteVertical(-number);              // -1*number is positive.
      | This is Spot #1 referred to in the text.
   }
   else if (number < 10)
      System.out.println(number);               // Write the one digit.
   else
   {
      superWriteVertical(number/10);            // Write all but the last digit.
      | This is Spot #2 referred to in the text.
      System.out.println(number % 10);          // Write the last digit.
   }
}
```

This produces this output with a minus sign on the first line:

```
-
3
6
1
```

How do we handle a negative number? The first step seems clear enough: Print the negative sign. After this, we must print the digits of `number`, which are the same as the digits of `-1*number`, which is *positive* (because `number` itself is negative). So the pseudocode for `superWriteVertical` is an extension of our original pseudocode:

```
if (the number is negative)
{
     Write a negative sign.
     Write the digits of -1*number stacked vertically.
}
else if (the number has only one digit)
     Write that digit.
else
{
     Write the digits of number/10 stacked vertically.
     Write the single digit of number % 10.
}
```

If you think recursively, you will recognize that the step *write the digits of* `-1*number` *stacked vertically* is a simpler version of our original problem (simpler because the negative sign does not need to be written). This suggests the implementation in Figure 8.2 with two recursive calls: one for the new case that writes the digits of `-1*number` and a second call for the original case that writes the digits of `number/10`. The implementation has one simplification using "`-number`" instead of "`-1*number`". We also have added comments in the code, identifying two particular locations "Spot #1" and "Spot #2" to aid in taking a closer look at recursion.

A Closer Look at Recursion

The computer keeps track of method activations in the following way: When a method is activated, the computer temporarily stops the execution. Before actually executing the method, some information is saved that will allow the computation to return to the correct location after the method's work is completed. The computer also provides memory for the method's parameters and any local variables that the method uses. Next, the actual arguments are plugged in for the parameters, and the code of the activated method begins to execute.

how methods are executed

If the execution should encounter an activation of *another* method—recursive or otherwise—then the first method's computation is temporarily stopped. This is because the second method must be executed before the first method can continue. Information is saved that indicates precisely where the first method should resume when the second method is completed. The second method is given memory for its own parameters and local variables. The execution then proceeds to the second method. When the method is completed, the execution returns to the correct location within the first method, and the first method resumes its computation.

This mechanism is used for both recursive and nonrecursive methods. As an example of the mechanism in a recursive method, let's completely trace the work of `superWriteVertical(-36)`. Initially, we activate `superWriteVertical` with `number` set to `-36`. The actual argument, `-36`, is copied to the parameter, `number`, and we start executing the code. At the moment when the method's execution begins, all of the important information that the method needs to work is

Activation record for first call to superWriteVertical

number: –36

When the method returns, the computation should continue at line 57 of the main program.

stored in a memory block called the method's **activation record**. The activation record contains information as to where the method should return when it is done with its computation, and it also contains the values of the method's local variables and parameters. For example, if our `superWriteVertical` method was called from a main program, then the activation record might contain the information shown to the left.

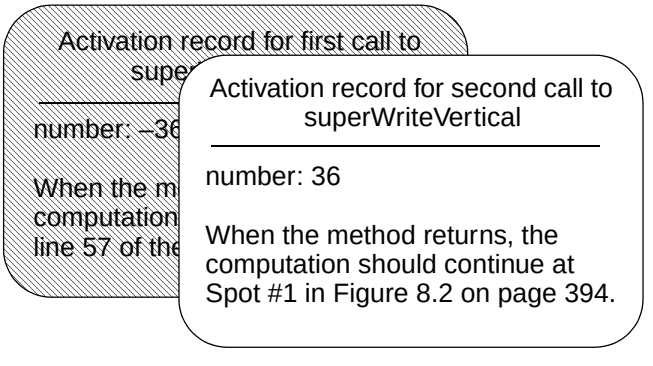
The return location specified in a real activation record does not actually refer to lines of code in the main program, but when you're imagining an activation record, you can think of a return location in this manner.

With the activation record in place, the method starts executing. Because the `number` is negative, the boolean test of the if-statement is `true`, and the negative sign is printed. At this point, the computation is about to make a recursive call, indicated here:

A recursive call is made in the superWriteVertical method.

```
if (number < 0)
{
   System.out.println("-");
   superWriteVertical(-number);
   || This is Spot #1 referred to in the text.
}
```

This method generates its own activation record with its own value of `number` (which will be `36`) and its own return location. The new activation record is placed on top of the other activation record, like this:



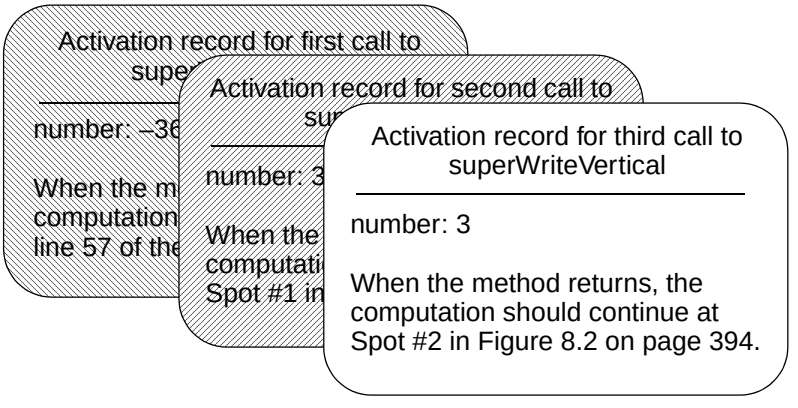
In fact, the collection of activation records is stored in a *stack* data structure called the **execution stack.** Each activation of a method pushes the next activation record on top of the execution stack.

In our example, the second call of `superWriteVertical` executes with its own value of `number` equal to `36`. The method's code executes, taking the last branch of the if-else control structure, arriving at another recursive call:

```
else
{
    superWriteVertical(number/10);
    | This is Spot #2 referred to in the text.
    System.out.println(number % 10);
}
```

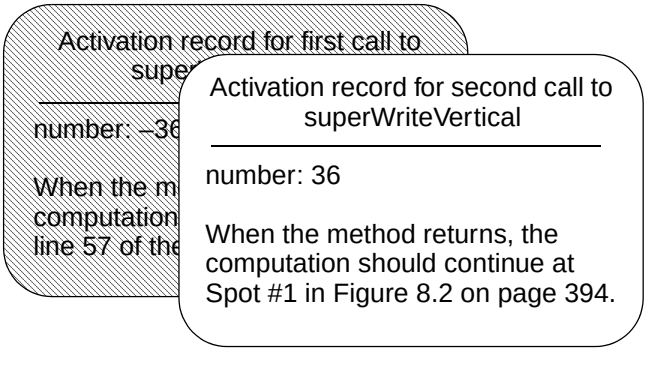
Another recursive call is made in the superWriteVertical method.

To execute this recursive call, another activation record is created (with `number` now set to 3), and this activation record is pushed onto the execution stack, as shown here:



The `superWriteVertical` method begins executing once more. With `number` set to 3, the method enters the section to handle a one-digit number. At this point, the digit 3 is printed, and no other work is done during this activation.

When the third method activation ends, its activation record is popped off the stack. But just before it is popped, the activation record provides one last piece of information—telling where the computation should continue. In our example, the third activation record is popped off the stack, and the computation continues at Spot #2 in Figure 8.2 on page 394. At this point, we have the two remaining activation records shown here:



As we said, the computation is now at Spot #2 in Figure 8.2 on page 394. This is the highlighted location shown here:

```
else
{
    superWriteVertical(number/10);
    || This is Spot #2 referred to in the text.
    System.out.println(number % 10);
}
```

The next statement is an output statement. What does it print? From the activation record on top of the stack, we see that `number` is `36`, so the statement prints 6 (which is `36 % 10`). The second method activation then finishes, returning to Spot #1 in Figure 8.2 on page 394. But there is no more work to do after Spot #1, so the first method also returns. The total effect of the original method activation was to print three characters: a minus sign, then `3`, and finally `6`. The tracing was all accomplished with the usual mechanism for activating a method—no special treatment was needed to trace recursive calls. In the example, there are two levels of recursive calls:

1. `superWriteVertical(-36)` made a recursive call to `superWriteVertical(36)`.
2. `superWriteVertical(36)` made a recursive call to `superWriteVertical(3)`.

In general, recursive calls may be much deeper than this, but even at the deepest levels, the activation mechanism remains the same as the example that we have traced.

General Form of a Successful Recursive Method

Key Design Concept

Find smaller versions of a problem within the larger problem itself.

Java places no restrictions on how recursive calls are used within a method. However, for recursive methods to be useful, any sequence of recursive calls must ultimately terminate with some piece of code that does not depend on recursion—in other words, there must be a *stopping case*. The method may call itself, and that recursive call may call the method again. The process may be repeated any number of times. However, the process will not terminate unless eventually one of the recursive calls does not itself make a recursive call. The general outline of a recursive method definition is as follows.

1. Suppose a problem has one or more cases in which some of the subtasks are simpler versions of the same problem you are trying to solve in the first place. These subtasks are solved by recursive calls.
2. A method that makes recursive calls must also have one or more cases in which the entire computation is accomplished without recursion. These cases without recursion are called **stopping cases** or **base cases**.

Often, a series of if-else statements determines which of the cases will be executed. A typical scenario is for the first method activation to execute a case that includes a recursive call. That recursive call may in turn execute a case that requires another recursive call. For some number of times, each recursive call produces another recursive call, but eventually one of the stopping cases applies. Every call of the method must eventually lead to a stopping case; otherwise, the execution will never end because of an infinite sequence of recursive calls. In practice, infinite recursion will terminate with a Java exception called `StackOverflowError`, indicating that the execution stack has run out of memory for more activation records.

Self-Test Exercises for Section 8.1

1. What is the output produced by `f(3)` for each of the following?

```
public static void f(int n)
{
   System.out.println(n);
   if (n > 1)
      f(n-1);
}
```

```
public static void f(int n)
{
   if (n > 1)
      f(n-1);
   System.out.println(n);
}
```

```
public static void f(int n)
{
    System.out.println(n);
    if (n > 1)
       f(n-1);
    System.out.println(n);
}
```

2. What is the output of the following method with an argument of 3?

```
public static void cheers(int n)
{
   if (n <= 1)
      System.out.println("Hurrah");
   else
   {
      System.out.println("Hip");
      cheers(n-1);
   }
}
```

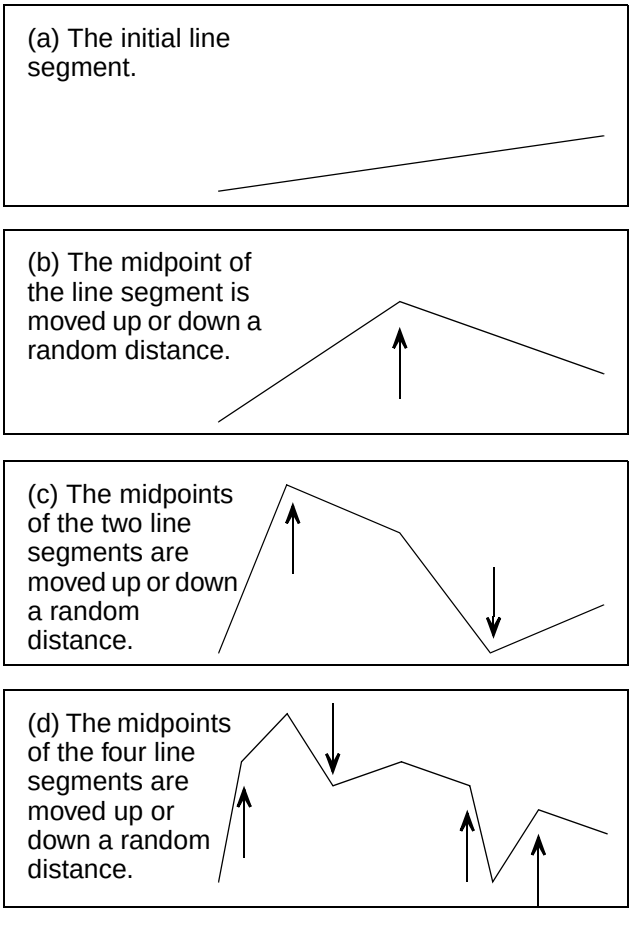
3. Modify the `cheers` method from the preceding exercise so that it first prints `"Hurrah"` followed by `n-1` `"Hip"`s. Make a further modification so that `n-1` `"Hip"`s occur both before and after the `"Hurrah"`. Make another modification so that approximately half of the `"Hip"`s occur before the `"Hurrah"` and half appear after.

4. Write a recursive method with a parameter that is a non-negative integer. The method writes that number of asterisks to the screen, followed by that number of exclamation points. Use no loops or local variables.

8.2 STUDIES OF RECURSION: FRACTALS AND MAZES

Recursive thinking makes its biggest impact on problems in which one of the *subtasks* is a simpler version of the *same* problem you are working on. For example, when we write the digits of a long `number`, our first step is to write the digits of the smaller number, `number/10`. This section provides additional examples of recursive thinking and the methods that recursion leads to.

FIGURE 8.3
The First Few Steps in Generating a Random Fractal

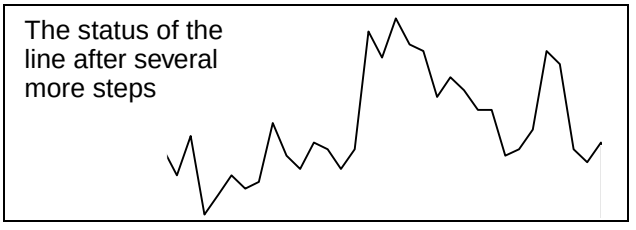


PROGRAMMING EXAMPLE: Generating Random Fractals

Fractals are one of the techniques that graphics programmers use to artificially produce remarkably natural scenes of mountains, clouds, trees, and other objects. We'll explain fractals in a simple setting and develop a recursive method to produce a certain kind of fractal.

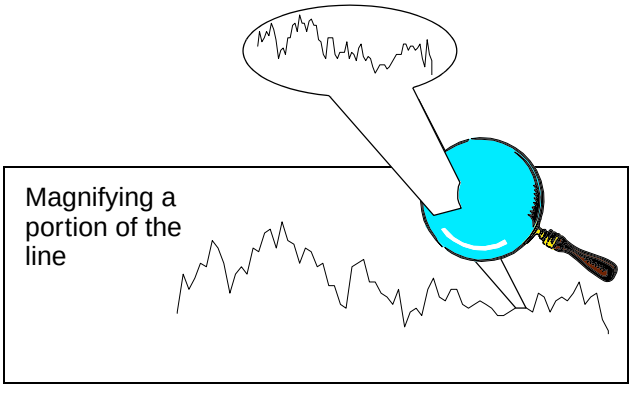
To understand fractals, think about a short line segment, as shown in Figure 8.3(a). Imagine grabbing the middle of the line and moving it vertically a random distance. The two endpoints of the line stay fixed, so the result might look like Figure 8.3(b). This movement has created two smaller line segments: the left half of the original segment and the right half. For each of these smaller line segments, we'll grab the midpoint and move it up or down a random distance. Once again, the endpoints of the line segments remain fixed, so the result of this second step might look like Figure 8.3(c). One more step might produce Figure 8.3(d). The process continues as long as you like, with each step creating a slightly more jagged line.

After several more steps, the line could appear as shown here:



Perhaps you can imagine that this jagged line is the silhouette of a mountain skyline.

The line that we're generating has an interesting property. Suppose that we carry out thousands of steps to create our line and then magnify a small portion of the result, as shown here:



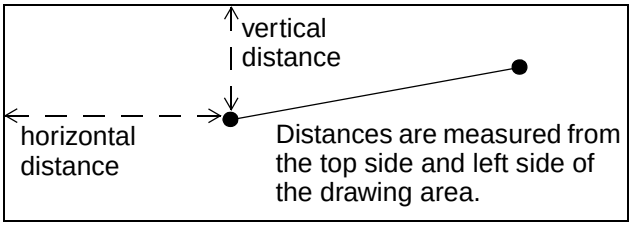
fractals, nature's fractals, and random fractals

The magnified portion is not identical to the entire line, but there is a lot of similarity. **Fractal** is the term coined by the mathematician Benoit Mandelbrot to describe objects, such as our line, that exhibit some kind of similarity under magnification. In fact, the first fractals studied were mathematically defined sets that remained *completely unchanged* when magnified to certain powers. In nature, objects don't generally remain completely unchanged under magnification, but magnified views commonly exhibit similarities with the larger object (such as our line, or a cloud, or a fern). Also, in nature (and our line), the powers of magnification where the similarities occur are limited, so nature's fractals are really rough approximations to the more formal mathematical fractals. Even so, the term *fractal* is often applied to any object that exhibits similarities under some magnification. The jagged line we have described is called a **random fractal** because of the randomness in its generation.

A Method for Generating Random Fractals—Specification

We wish to write a method that can draw a random fractal in the way we have described. The input to this method includes the locations of the two endpoints

of the original line, measured from the top side and the left side of the drawing area. For example, consider the line segment shown here:



In computer graphics, these distances are measured in **pixels**, which are the individual dots on a computer screen. Today's typical screens have about 100 pixels per inch, so the endpoint just shown might have an x-coordinate of 100 (the distance from the left side) and a y-coordinate of 55 (the distance below the top side). These y-coordinates are probably different than what you're used to because positive coordinates occur below the top side, whereas normal mathematics coordinates have positive numbers going upward—but that's the way computer graphics are usually measured, from the top down.

The x- and y-coordinates of both endpoints will be four of the parameters to the fractal-generating method. The method also has a fifth parameter called `drawingArea`. The `drawingArea` is a Java `Graphics` object, which is a class from the `java.awt` package. A `Graphics` object allows a program to draw images based on pixels. For example, one of the `Graphics` methods is specified here:

- **drawLine** (from the `Graphics` class)

  `public void writeVertical(int x1, int y1, int x2, int y2)`

  Draw a line on this `Graphics` object.

  Parameters:

  `x1 and y1` – the `x` and `y` pixel coordinates of one endpoint of the line

  `x2 and y2` – the `x` and `y` pixel coordinates of the other endpoint of the line

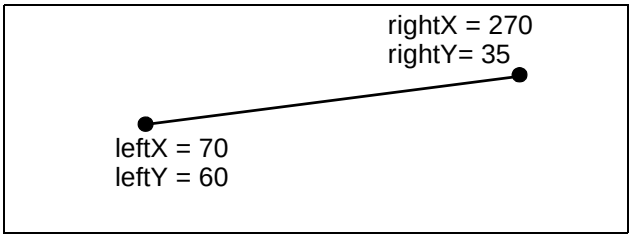
  Postcondition:

  A line has been drawn on this `Graphics` object from (`x1, y1`) to (`x2, y2`).

Java programs have several ways to create `Graphics` objects, but we don't need to know about that yet. Instead, we'll provide the specification of the fractal method without worrying about exactly how the program generates a `Graphics` object. Here is a listing of the method with its five parameters:

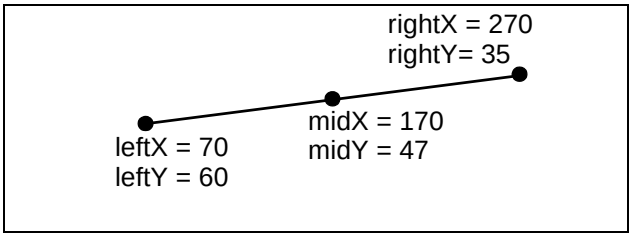
```
public static void randomFractal(
    int leftX,
    int leftY,
    int rightX,
    int rightY,
    Graphics drawingArea
)
```

Now we must consider the method's behavior. Let's look at an example using an initial line segment shown here with its x- and y-coordinates:



Notice that we're using pixel coordinates, so the higher point has the lower y-coordinate.

Normally, the first step of `randomFractal` is to find the approximate x- and y-coordinates of the segment's midpoint. We'll call these coordinates `midX` and `midY`, as shown here:



We are keeping track of the coordinates with integers, so the y-midpoint is a bit inaccurate—the real midpoint is $47\frac{1}{2}$. The statements to compute the approximate midpoints use integer division (throwing away any remainder), as shown here:

```
midX = (leftX + rightX) / 2;
midY = (leftY + rightY) / 2;
```

Next, the y-midpoint must be shifted up or down a random amount. To limit the rise and fall of the fractal, the complete shift will be no more than half of the distance from `leftX` to `rightX`. The amount of the shift is stored in a variable called `delta`, computed randomly with this statement:

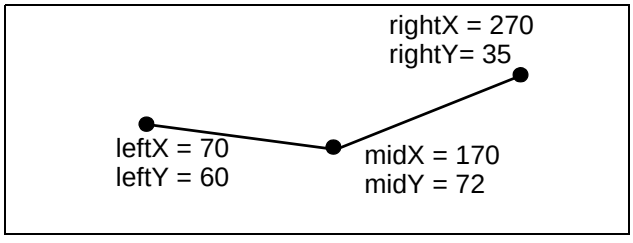
```
delta = (int)((Math.random( ) - 0.5) * (rightX - leftX));
```

The random part of this expression, `Math.random( )`, provides a random double number from 0 up to but not including 1. So (`Math.random( ) - 0.5`) is from `-0.5` to `0.5` (not including `0.5`). The assignment to `delta` is this random number times the distance between the x-coordinates and rounded to an integer.

Once we have `delta`, this value is added to `midY` with the statement:

```
midY += delta;
```

For example, suppose that `delta` is 25. Then the `midY` shifts down 25 pixels, as shown here:



After computing the location of the displaced midpoint, we have two segments: from the left endpoint of the original segment to the displaced midpoint, and from the displaced midpoint to the right endpoint of the original segment. And what work do we have left to do? *We must generate a random fractal for each of these two smaller line segments.* This work can be accomplished with two recursive calls:

```
randomFractal(leftX,  leftY,  midX,   midY,   drawingArea);
randomFractal(midX,   midY,   rightX, rightY, drawingArea);
```

two recursive calls solve two smaller versions of the original problem

The first of these recursive calls generates a random fractal for the leftmost smaller line segment. Let's examine the arguments of the first recursive call: The first two arguments are the left coordinates of the leftmost segment (`leftX` and `leftY`), the next two arguments are the right coordinates of the leftmost segment (`midX` and `midY`), and the final argument is the `Graphics` object that we are drawing on. The second recursive call handles the rightmost line segment in a similar way.

To summarize, the `randomFractal` method normally carries out these six statements:

```
midX = (leftX + rightX) / 2;
midY = (leftY + rightY) / 2;
delta = (int)((Math.random( ) - 0.5) * (rightX - leftX));
midY += delta;
randomFractal(leftX,  leftY,  midX,   midY,   drawingArea);
randomFractal(midX,   midY,   rightX, rightY, drawingArea);
```

stopping case for the random fractal method

These six statements cannot be all of `randomFractal`. For one thing, no drawing takes place in these statements. For another, these statements on their own will lead to infinite recursion—there is no stopping case. The solution is a stopping case that's triggered when `leftX` and `rightX` get close to each other. In this case, the horizontal extent of the segment is small and further subdividing of the segment has little effect. So this will be our stopping case, and we will simply draw a line from the left endpoint to the right endpoint. This approach is taken in Figure 8.4 on the next page.

FIGURE 8.4 A Method to Generate a Random Fractal

Specification

- **randomFractal**

```
public static void randomFractal(
   int leftX,
   int leftY,
   int rightX,
   int rightY,
   Graphics drawingArea
)
```

Draws a random fractal on a `Graphics` object.

Parameters:

`leftX` and `leftY` – the `x` and `y` pixel coordinates of one endpoint of a line segment
`rightX` and `rightY` – the `x` and `y` pixel coordinates of the other endpoint of a line segment
`drawingArea` – a `Graphics` object on which to draw a random fractal

Postcondition:

A random fractal has been drawn on the `Graphics` object. The fractal extends from (`leftX`, `leftY`) to (`rightX`, `rightY`).

Implementation

```
public static void randomFractal
(
   int leftX,
   int leftY,
   int rightX,
   int rightY,
   Graphics drawingArea
)
{
   final int STOP = 4;
   int midX, midY;
   int delta;

   if ((rightX - leftX) <= STOP)
      drawingArea.drawLine(leftX, leftY, rightX, rightY);
   else
   {
      midX = (leftX + rightX) / 2;
      midY = (leftY + rightY) / 2;
      delta = (int)((Math.random( ) - 0.5) * (rightX - leftX));
      midY += delta;
      randomFractal(leftX,  leftY,  midX,   midY,   drawingArea);
      randomFractal(midX,   midY,   rightX, rightY, drawingArea);
   }
}
```

The Stopping Case for Generating a Random Fractal

The `randomFractal` method from Figure 8.4 reaches a stopping case when the distance from `leftX` to `rightX` gets small enough, as shown here:

```
if ((rightX - leftX) <= STOP)
   drawingArea.drawLine(leftX, leftY, rightX, rightY);
```

The constant, `STOP`, is defined as 4 at the top of the `randomFractal` method. In the stopping case, we use the `Graphic drawLine` method to draw a line segment from the left endpoint to the right endpoint.

Putting the Random Fractal Method in an Applet

There's one item you're probably still wondering about. How does a program get a `Graphics` object to draw on? One approach can be used by any applet that draws a single fixed image with no user interaction. This approach has five easy steps, though you must also be familiar with the general format of applets as described in Appendix I.

The fixed-image approach for an applet follows these steps:

1. Import `java.awt.*`, which provides the `Graphics` class and another class called `Image`.

2. The applet declares two private instance variables, like this:

   ```
   private Image display;
   private Graphics drawingArea;
   ```

   An `Image` is an area that can store a drawing. Sometimes these drawings are obtained from external files, but the drawings can also be created by the applet itself by attaching a `Graphics` object to the `Image`. This is what we plan to do; the `drawingArea` (a `Graphics` object) will be attached to the `display` (an `Image`).

3. The applet's `init` method initializes the `Image` and sets the `Graphics` object to draw on that image. This takes four statements:

   ```
   int height = getSize( ).height;
   int width = getSize( ).width;
   display = createImage(width, height);
   drawingArea = display.getGraphics( );
   ```

   The `width` and `height` variables are initialized with `getSize( ).width` and `getSize( ).height`, which provide the current size of a running applet. The `createImage` method is also part of any applet, and it creates an `Image` with a given width and height in pixels. The fourth statement sets `drawingArea` to a `Graphics` object that can be used to draw on the `Image`.

4. The rest of the applet's `init` method draws any items that you want to appear in the image. These are drawn using the `drawingArea Graphics` object. For example, we want an applet that draws a random fractal, so our `init` method will have this complete implementation:

```
public void init( )
{
  int height = getSize( ).height;
  int width = getSize( ).width;
  display = createImage(width, height);
  drawingArea = display.getGraphics( );

  randomFractal(0, height/2, width, height/2, drawingArea);
}
```

   The last line of the `init` method does the actual drawing by calling the `randomFractal` method. The fractal will be drawn in `drawingArea`, which is the `Graphics` object that we set to draw in the `display Image`.

   As with any applet, the `init` method is automatically called when the applet starts. This will put the `randomFractal` image in the `display Image`.

5. To actually show the image, we need to provide one more applet method called `paint`. The `paint` method is called when the applet is started and whenever the applet's location or shape changes, such as moving from behind some other window. For the applet to draw a single fixed image, the paint method is simply this:

```
public void paint(Graphics g)
{
   g.drawImage(display, 0, 0, null);
}
```

   The argument, `g`, is the underlying `Graphics` object where all applet drawing takes place. It is different than `drawingArea` (which is the `Graphics` object attached to the image that we drew). The one statement that we placed in the `paint` method will draw our `display` image in the applet's drawing area, aligned in the upper-left corner. (The first argument is the image we are drawing; the second and third arguments indicate the coordinates where the upper-left corner of the image will be placed in the applet; the fourth argument is a Java object called an observer, but we can set it to `null` if we are not using this feature.)

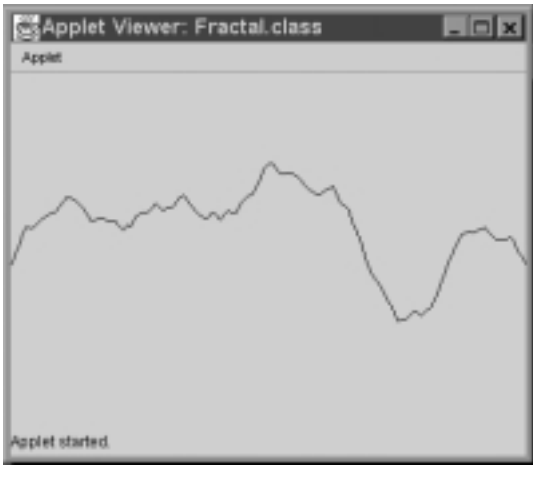
As we said, any applet that draws a single fixed image can follow these five steps. A fractal-drawing applet following these steps is shown in Figure 8.5 on page 408, along with a picture of how the applet looks when it runs.

FIGURE 8.5 Implementation of an Applet to Draw a Random Fractal

Java Applet Implementation

```
// File: Fractal.java
// This applet is a small example to illustrate the randomFractal method.

import java.applet.Applet;
import java.awt.*;           // Provides Graphics, Image

public class Fractal extends Applet
{
   private Image display;
   private Graphics drawingArea;

   public void init( )
   {
     int height = getSize( ).height;
     int width = getSize( ).width;
     display = createImage(width, height);
     drawingArea = display.getGraphics( );

     randomFractal(0, height/2, width, height/2, drawingArea);
   }

   public void paint(Graphics g)
   {
      g.drawImage(display, 0, 0, null);
   }

   public static void randomFractal
   (
      int leftX,
      int leftY,
      int rightX,
      int rightY,
      Graphics drawingArea
   )
   {
      || See Figure 8.4 on page 405.
   }
}
```

Sample of the Applet Running



PROGRAMMING EXAMPLE: Traversing a Maze

Suppose your friend Jervis has a maze in his backyard. One day, Jervis mentions two facts about the maze:

1. Somewhere in the maze is a magic tapestry that contains the secret of the universe.
2. You can keep this tapestry (and its secret) if you can enter the maze, find the tapestry, and return to the maze's entrance. (So far, many have entered, but none has returned.)

The maze is built on a rectangular grid. At each point of the grid, there are four directions to move: north, east, south, or west. Some directions, however, may be blocked by an impenetrable wall. You decide to accept Jervis's challenge and enter the maze—but only with the help of your portable computer and a method that we'll write to guide you into the maze *and back out*.

Traversing a Maze—Specification

We plan to write a method, `traverseMaze`, which you can execute on a portable computer that you will carry through the maze. The method will give you directions and ask you questions to take you to the magic tapestry and back out. Here's the complete specification:

- **traverseMaze**

  ```
  public static boolean traverseMaze( )
  ```

  Provide interactive help to guide a user through a maze and back out.

  Precondition:
  The user of the program is facing an unblocked spot in the maze, and this spot has not previously been visited by the user.

  Postcondition:
  The method has asked a series of queries and provided various directions to the user. The queries and directions have led the user through the maze and back to the exact same position where the user started. If there was a magic tapestry that could be reached in the maze, then the user has picked up this tapestry, and the method returns `true`; otherwise, the method returns `false`.

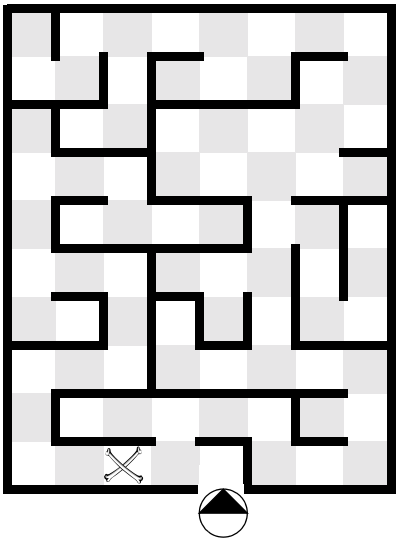
Figure 8.6 shows a drawing of what the maze might look like, along with a sample dialogue. A sample dialogue written at this point—before we've written the program—is called a **script**, and it can help clarify a loose specification. As you might suspect, Jervis's actual maze is more complex than this sample dialogue might suggest, but simplicity often results in the best scripts.

FIGURE 8.6 Script for the Maze Traversal Method

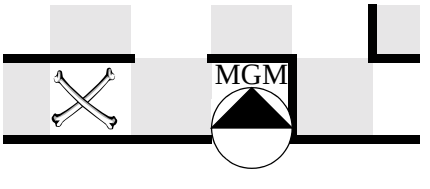
A Sample Dialogue

```
Step forward & write your name on the ground.
Have you found the tapestry? [Y or N]
N
Please turn left 90 degrees.
Are you facing a wall? [Y or N]
N
Is your name written ahead of you? [Y or N]
N
Step forward & write your name on the ground.
Have you found the tapestry? [Y or N]
N
Please turn left 90 degrees.
Are you facing a wall? [Y or N]
Y
Please turn right 90 degrees.
Are you facing a wall? [Y or N]
N
Is your name written ahead of you? [Y or N]
N
Step forward & write your name on the ground.
Have you found the tapestry? [Y or N]
Y
Pick up the tapestry, and take a step backward.
Please turn right 90 degrees.
Please turn right 90 degrees.
Please step forward, then turn 180 degrees.
Please turn right 90 degrees.
Please turn right 90 degrees.
Please turn right 90 degrees.
Please step forward, then turn 180 degrees.
```

This dialogue supposes that the user starts at the entrance to the maze drawn below. The starting point of the user is represented by the arrow in a circle, with the arrow pointing in the direction that the user is facing. The large X is the magic tapestry.



Let's trace through this script to make sure it's correct. I'll pretend that I am the user and I'm standing, facing the entrance to the maze in Figure 8.6. The program asks me to step forward and write my name on the ground, which I do. I am now standing on the white square just inside the entrance, still facing in the same direction that I started, and my name is written in the dirt underneath me. Here's the bottom part of the maze at that point (the "MGM" is my name):



Next, the script asks me whether I have found the tapestry, and I answer `N`. I am then told to turn left 90 degress, which I do, placing me in this situation:

Am I facing a wall? (I answer `N`.) Is my name written ahead of me? (Again, I answer `N` because my name is not written in the square in front of me, which is the dark square just to the left of the entrance.) So, again, I am told to step forward and write my name on the ground, which I do:

My name is now written on the first two squares of the maze, and the program can use that information to determine which squares I have already visited. Once again, I am asked whether I have found the tapestry (I answer `N`) and then I'm told to turn left 90 degrees, like this:

Am I facing a wall? Yes, I am, so I am told to turn *right* 90 degrees. Kind of seems like I'm not getting anywhere because I've gone back to where I was a moment ago:

But even though I am back in a previous situation, the program has gained some information about one of the walls. If you continue following this particular script, you'll see that I pick up the tapestry and find my way back out of the maze. The exact instructions aren't that important to understanding the script. In fact, at this point, we might not even have a complete working script, just an idea that the script will ask me questions and give me orders that will take me to the tapestry and safely back out of the maze. So let's move on to apply recursive thinking to this problem of asking the questions and giving the orders to traverse the maze.

Traversing a Maze—Design

The `traverseMaze` method appears to perform a pretty complex task. When the method starts, all that's really known is that the user of the program is facing some spot in the maze, and that spot has not been previously visited. The method must take the user into the maze, eventually leading the user back to the exact starting spot. Hopefully, along the way, the user will find the magic tapestry. Recursive thinking can make the task easier.

We'll call the method's user Judy. We'll always start by asking Judy to take a step forward onto the spot she hasn't visited before. We'll also ask her to write her name on the ground so that later we can tell whether she's been to this spot before. After these two steps, we will ask her whether she has found the tapestry at this new spot and will place her `true` or `false` answer in a local variable named `found`.

the stopping case

Now there is one easy case: If the tapestry is at this spot, then we will have Judy pick up the tapestry and ask her to take a step backward to her starting spot. This is the stopping case, and the method returns `true` to indicate that the tapestry was found.

But what if Judy does not find the tapestry at this spot? In this case, there are three possible directions for Judy to explore: forward, left, or right. We won't worry about exploring backward because that is the direction that Judy just came from. Also, sometimes we do not need to explore all three directions (forward, left, and right). A direction is a "dead end" if (a) there is a wall blocking that direction, or (b) Judy's name is written on the ground in that direction (indicating she has already been there)—and there is no need to explore dead ends. We also don't need to explore a direction if Judy already found the tapestry in one of the other directions.

This description suggests the following steps if the user has not found the tapestry at this spot:

1. Have the user face left (the first direction to explore).
2. `for` each of the three directions
   {
   `if (!found` && the direction that is now being faced isn't a dead end)
   {
   Explore the direction that is now being faced, returning to this exact spot after the exploration and setting `found` to `true` if the tapestry was found.
   }
   Have the user turn 90 degrees to the right (the next direction).
   }
3. The user is now facing the direction she came from, so ask her to step forward and turn around (so that she is facing the same spot that she was before this method was activated).

Are you thinking recursively? The highlighted step "Explore the direction that is now being faced. . ." is a simpler instance of the problem we are trying to solve. The instance is simpler because one spot in the maze has been eliminated from consideration—namely, the spot that the user is at does not contain the tapestry. In our implementation, we'll solve this simpler problem with a recursive call.

Traversing a Maze—Implementation

Our implementation will benefit from two methods to carry out various subtasks. Figure 8.7 shows the methods we have in mind. The first method provides a simple way to ask a question and get a yes/no answer. The other method determines whether the user has reached a dead end.

A short discussion of these two methods appears after the figure.

FIGURE 8.7 Two Methods to Simplify the Maze Traversal

Specification

- **inquire**

  `public static boolean inquire(String query)`

  Ask a yes/no question and read the answer.

  Parameters:

  `query` – a question to ask

  Postcondition:

  The `query` has been printed to `System.out` and a one-character response read from `System.in` (skipping any whitespace characters). The method returns `true` if the user's response was `'Y'` or `'y'` and returns `false` if the user's response was `'N'` or `'n'`. (If the response is some other character, then the query is repeated and a new answer read.)

- **deadend**

  `public static boolean deadend( )`

  Determine whether the person traversing the maze has reached a dead end.

  Postcondition:

  The return value is `true` if the direction directly in front of the user is a dead end (that is, a direction that cannot contain the tapestry).

(continued)

(FIGURE 8.7 continued)

Implementation

```
public static boolean inquire(String query)
{
   char answer = 'N';

   do
   {
      System.out.println(query + " [Y or N]");
      try
      {
         do
            answer = (char) System.in.read( );
         while (Character.isWhitespace(answer));
      }
      catch (IOException e)
      {
         System.err.println("Standard input error: " + e);
         System.exit(0);
      }
      answer = Character.toUpperCase(answer);
   }  while ((answer != 'Y') && (answer != 'N'));

   return (answer == 'Y');
public static boolean deadend( )
{
   return inquire("Are you facing a wall?")
          ||
          inquire("Is your name written in front of you?");
}
```

The two methods in Figure 8.7 are:

- `inquire`: This method asks the user a yes/no question; it returns `true` if the user answers `'Y'`or `'y'` and returns `false` if the user answers `'N'` or `'n'`.
- `deadend`: This method determines whether the direction in front of the user is a dead end. Remember that dead ends are caused by a wall or by a direction that the user has already explored. The method returns `true` (for a dead end) or `false` (for no dead end).

The implementation of the `deadend` method is short, as shown here:

```
return inquire("Are you facing a wall?")
       ||
       inquire("Is your name written in front of you?");
```

This implementation of `deadend` makes two activations of the `inquire` method and returns the "or" of the results (using the `||` operation). When the "or" expression is evaluated, the `inquire` method is called to ask the user, "Are you facing a wall?" If the user answers `'Y'` or `'y'`, then the `inquire` method returns `true`, and the rest of the "or" expression will not be executed. This follows the general rule of **short-circuit evaluation** that we have seen before—meaning that the evaluation of a logical expression stops when there is enough information to determine whether the expression is `true` or `false`. On the other hand, if the user answers `'N'` or `'n'` to the first query, then the `inquire` method returns `false`, and the rest of the "or" expression will be executed (asking the second question, "Is your name written in front of you?").

In all, the method returns `true` if the user indicates that he or she is facing a wall or if the user says there is no wall and then answers `'Y'` or `'y'` to the second question (the user's name is written in front of him or her).

The actual `traverseMaze` method is given in Figure 8.8. It follows our pseudocode quite closely. You might enjoy knowing that the magic tapestry really does exist in a book called *Castle Roogna* (Ballantine Books) by Piers Anthony. The hero of the book actually becomes part of the tapestry, whereupon his quest leads him to a *smaller* version of the *same* tapestry.

The Recursive Pattern of Exhaustive Search with Backtracking

The `traverseMaze` method of Figure 8.8 follows a recursive pattern that you may find useful elsewhere. The pattern is useful when a program is searching for a goal within a space of individual points that have connections between them. In the maze problem, the "points" are the individual squares of the maze, and the "connections" are the possible steps that the explorer can take in the four compass directions. Later you'll run into many data structures that have the form of "points and connections"—in fact, our very next chapter introduces trees that are such a structure.

The task of searching such a structure can often follow this pattern:

- Start by marking the current point in some way. In the maze, the mark was obtained by asking the explorer to write her name on the ground. The purpose of the mark is to ensure that we don't mistakenly return to this point and end up going around in circles, continually returning to this same spot.

  This "marking" step is not always necessary; sometimes there are other mechanisms to prevent unbounded repetitions of searching the same direction.

- Check whether the current point satisfies the goal. If so, return some value that indicates the success of the search.

- On the other hand, if the current point does not satisfy the goal, then one by one examine the other points that are connected to the current point. For each such point, check to see whether the point is marked; if so, we can ignore the point because we have already been there. On the other hand, if a connected point is not marked, then make a recursive call to continue the search from the connected point onward. If the recursive call finds the goal, then we won't bother checking any other points, but if the recursive call fails, then we will check each of the other unmarked connected points by further recursive calls.

This pattern is called **exhaustive search with backtracking**. The term *exhaustive search* means that all possibilities are tried. *Backtracking* is the process of a recursive call returning without finding the goal. When such a recursive call returns, we are "back where we started," and we can explore other directions with further recursive calls.

Exhaustive search with backtracking is most useful when the known search space doesn't get too large. But even with huge search spaces, programmers often use variants that try to cut down the search space in an intelligent manner. You'll find successful variants in programs that play games such as chess that have an enormous number of possibilities to search. As one more recursive example, we'll write a method to play a game with a definite small search space, so we won't need to cut down this space at all.

FIGURE 8.8 A Method to Traverse a Maze

Specification

- **traverseMaze**

  `public static boolean traverseMaze( )`

  Provide interactive help to guide a user through a maze and back out.

  Precondition:

  The program's user is facing an unblocked spot that has not been previously visited.

  Postcondition:

  The method has asked a series of questions and provided various directions to the user. The questions and directions have led the user through the maze and back to the exact same position where the user started. If there was a magic tapestry that could be reached in the maze, then the user has picked up this tapestry, and the method returns `true`; otherwise, the method returns `false`.

Implementation

```
public static boolean traverseMaze( )
{
   int direction;  // Counts 1, 2, 3 for the three directions to explore
   boolean found;  // Will be set to true if we find the tapestry

   System.out.println("Step forward & write your name on the ground.");
   found = inquire("Have you found the tapestry?");

   if (found)
   {  // Pick up the tapestry and step back from whence you came.
      System.out.println("Pick up the tapestry and take a step backward.");
   }
   else
   {  // Explore the three directions (not counting the one that you just came from). Start
      // with the direction on your left, and then turn through each of the other directions.
      System.out.println("Please turn left 90 degrees.");
      for (direction = 1; direction <= 3; direction++)
      {
         if ( !found && !deadend( ) )
            found = traverseMaze( );
         System.out.println("Please turn right 90 degrees.");
      }
      // You're now facing the direction from whence you came, so step forward and turn
      // around. This will put you in the same spot when the method was activated.
      System.out.println("Please step forward, then turn 180 degrees.");
    }
    return found;
}
```

PROGRAMMING EXAMPLE: The Teddy Bear Game

Here are the rules of the Teddy Bear game: Your friend is going to give you a certain number of bears. The number of bears is called `initial`, and your goal is to end up with a particular number of bears, called the `goal` number.

There are two other integer parameters to the game: `increment` and `n`. At any point in the game, you have two choices: (a) You can ask for (and receive) `increment` more bears, or (b) If you have an even number of bears, then you can give half of them back to your friend. Each time you do (a) or (b), that is called a *step* in the game, and the goal must be reached in `n` steps or fewer. For example, if `initial` is 99, `increment` is 53, and `n` is at least 4, then the following sequence of steps will reach the goal of 91:

$$99 \underset{\text{step a}}{\rightarrow} 152 \underset{\text{step b}}{\rightarrow} 76 \underset{\text{step b}}{\rightarrow} 38 \underset{\text{step a}}{\rightarrow} 91$$

We want to write a recursive method, `bears`, that determines whether it is possible to reach a `goal` starting with some `initial` and `increment` numbers (allowing no more than `n` steps). The implementation follows the pattern from the previous page, although the first marking step of the pattern is not needed since we can prevent going around in circles by stopping when the parameter `n` reaches zero. So the pattern has only these two steps:

- Check whether the `initial` value is equal to the `goal`. If so, return `true` to indicate that the goal can be reached.
- On the other hand, if the `initial` value does not equal the `goal`, then we'll check that `n` is positive (otherwise we have no more moves to make and must return `false` because the goal cannot be reached). When `n` is positive, we'll solve the problem by making one or two recursive calls. One call starts by taking an (a)-step and the other starts by taking a (b)-step—although this second call is made only if `initial` is an even number.

The implementation of the bear method appears in Figure 8.9. Notice the expression `(inital % 2 == 0)` to determine whether the initial number of bears is even.

Self-Test Exercises for Section 8.2

5. Suppose you activate `randomFractal` (Figure 8.4 on page 405) with the distance between the x-coordinates equal to 24 and a stopping case when this distance is less than or equal to 4. Then `randomFractal` will make two recursive calls, and each of those will make two more calls, and so on until `width` is less than or equal to 4. How many total calls will be made of `randomFractal`, including the original activation?
6. Draw a copy of the maze from Figure 8.6 on page 410, moving the magic tapestry to a more difficult location. Run the `traverseMaze` method (Figure 8.8 on page 417), pretending that you are in this maze and following the method's directions. (Do not peek over the walls.)

7. Revise `randomFractal` (Figure 8.4 on page 405) so that the movements of the midpoints are not random. Instead, the first midpoint is moved upward by half the distance between the x-coordinates; the midpoints at the next level of recursion are moved down by half the distance between the x-coordinates; the next level moves up again, then down, and so on.
8. Suppose you are exploring a rectangular maze containing 10 rows and 20 columns. What is the maximum number of recursive calls generated if you start at the entrance of the maze and activate `traverseMaze` (Figure 8.8 on page 417)? Include the initial activation as part of your count.

FIGURE 8.9 A Method to Play the Teddy Bear Game

Specification

- **bears**

  `public static boolean bears(int initial, int goal, int increment, int n)`

  Determine whether the goal can be reached in the Teddy Bear game.

  Precondition:
  All parameters should be non-negative.

  Postcondition:
  The method has determined whether it is possible to reach the goal in the following Teddy Bear game. In the game, your friend gives you a certain number of bears. The number of bears is called `initial`, and your goal is to end up with a particular number of bears, called the `goal` number. At any point in the game, you have two choices: (a) You can ask for (and receive) `increment` more bears, or (b) If you have an even number of bears, then you can give half of them back to your friend. Each time you do (a) or (b), that is called a *step* in the game.

  The return value is `true` if and only if the goal can be reached in `n` steps or fewer.

Implementation

```
public static boolean bears(int initial, int goal, int increment, int n)
{
   if (initial == goal)
      return true;
   else if (n == 0)
      return false;
   else if (bears(initial+increment, goal, increment, n-1))
      return true;
   else if ((initial % 2 == 0) && bears(initial/2, goal, increment, n-1))
      return true;
   else
      return false;
}
```

8.3 REASONING ABOUT RECURSION

After a lecture on cosmology and the structure of the solar system, William James was accosted by a little old lady.

"Your theory that the sun is the center of the solar system, and the earth is a ball which rotates around it has a very convincing ring to it, Mr. James, but it's wrong. I've got a better theory," said the little old lady.

"And what is that, madam?" inquired James politely.

"That we live on a crust of earth which is on the back of a giant turtle."

Not wishing to demolish this absurd little theory by bringing to bear the masses of scientific evidence he had at his command, James decided to gently dissuade his opponent by making her see some of the inadequacies of her position.

"If your theory is correct, madam," he asked, "what does this turtle stand on?"

"You're a very clever man, Mr. James, and that's a very good question," replied the little old lady, "but I have an answer to it. And it is this: the first turtle stands on the back of a second, far larger, turtle, who stands directly under him."

"But what does this second turtle stand on?" persisted James patiently.

To this the little old lady crowed triumphantly. "It's no use, Mr. James—it's turtles all the way down."

J. R. ROSS
Constraints on Variables in Syntax

In all our examples of recursive thinking, the series of recursive calls eventually reaches a call that does not involve further recursion (that is, it reaches a *stopping case*). If, on the other hand, every recursive call produces another recursive call, then a recursive call will, in theory, run forever. This is called **infinite recursion**. In practice, such a method will run until the computer runs out of memory for the activation records and throws a `StackOverflowError`, indicating that the execution stack has run out of memory for activation records. Phrased another way, a recursive declaration should not be "recursive all the way down." Otherwise, like the lady's explanation of the solar system, a recursive call will never end, except perhaps in frustration.

In this section, we will show you how to reason about recursion, both to show that there is no infinite recursion and to show that a recursive method's results are correct. To illustrate how to do this reasoning, we will go through a series of methods, each of which incorporates a bit more recursion. The method is called `power`, and the activation of `power(x,n)` computes x^n so that `power(3.0, 2)` is 3.0^2 (which is 9.0), and `power(4.2, 3)` is 4.2^3 (which is 74.088). For any nonzero value of x, the value of x^0 is defined to be 1, so for example, `power(9.1, 0)` is 1. For a negative exponent, $-n$, the value returned is defined by:

$$x^{-n} = 1/x^n \quad \{\ x \text{ is any real number, and } -n \text{ is a negative integer } \}$$

For example, `power(3.0, -2)` is $1/3.0^2$ (which is $\frac{1}{9}$). The only forbidden power is taking 0^n when n is not positive. The complete specification of the `power` method follows.

- **power**

  `public static double power(double x, int n)`

  Compute the value of x^n.

  Precondition:

  If `x` is zero, then `n` must be positive.

  Returns:

  `x` raised to the `n` power

  Throws: `IllegalArgumentException`

  Indicates that `x` is zero and `n` is not positive.

Our implementation begins by checking the precondition and then deals with several cases. The first case, when `n` is non-negative, is easy. This case is computed by setting a local variable, `product`, to 1 and then repeatedly multiplying `product` by `x`. The repeated multiplication occurs `n` times, in this loop:

```
if (n >= 0)
{
   product = 1;
   for (count = 1, count <= n; count++)
      product = product * x;
   return product;
}
```

To understand what is needed for a negative exponent, let's consider a concrete case. Suppose we are computing `power(3.0, -2)`, which must return the value 3.0^{-2}. But this value is equal to $\frac{1}{3^2}$. Negative powers are the same as positive powers in the denominator. This means that if we know that the method returns the correct answer when `n` is positive, then we can calculate the correct value for `power(3.0, -2)` by the expression `1/power(3.0, 2)`. By thinking recursively, whenever `n` is negative, `power` can compute its answer with a recursive call, like this:

```
return 1/power(x, -n); // When n is negative (and so -n is positive)
```

Remember, in this statement, `n` is negative (such as -2), so `-n` is positive, and therefore, the recursive call in the expression `1/power(x, -n)` has a positive second argument. With a positive second argument, our `power` method makes no further recursive calls (i.e., a stopping case), and so the recursion ends.

This brings us to our first general technique for reasoning about recursion, which can be applied to the complete `power` method of Figure 8.10.

One-Level Recursion

Suppose every case is either a stopping case or makes a recursive call that is a stopping case. Then the deepest recursive call is only one level deep, and therefore no infinite recursion occurs.

FIGURE 8.10 Implementation of the `power` Method with Only One Level of Recursion

Specification

- **power**

  `public static double power(double x, int n)`

  Compute the value of x^n.

  Precondition:
  If `x` is zero, then `n` must be positive.

  Returns:
  `x` raised to the `n` power

  Throws: `IllegalArgumentException`
  Indicates that `x` is zero and `n` is not positive.

Implementation

```
public static double power(double x, int n)
{
   double product; // The product of x with itself n times
   int count;

    if (x == 0 && n <= 0)
        throw new IllegalArgumentException("x is zero and n=" + n);

    if (n >= 0)
    {
        product = 1;
        for (count = 1; count <= n; count++)
            product = product * x;
        return product;
    }
    else
        return 1/power(x, -n);
}
```

Sample Results of the Method

Call with these arguments		Return value of the method
x	n	
3.0	2	9.0
2.0	-3	0.125
4.1	0	1.0
-2.0	3	-8.0

FIGURE 8.11 Alternate Implementation of a Method to Compute Powers

Implementation

```
public static double pow(double x, int n)
{
   if (x == 0 && n <= 0)
      throw new IllegalArgumentException("x is zero and n=" + n);
   else if (x == 0)
      return 0;
   else if (n == 0)
      return 1;
   else if (n > 0)
      return x * pow(x, n-1);
   else // x is nonzero, and n is negative.
      return 1/pow(x, -n);
}
```

How to Ensure That There Is No Infinite Recursion in the More General Case

Recursive calls don't often stop just one level deep, but we needed to understand that case to form a base for deeper reasoning. Programmers have developed methods to reason about deeper recursive calls based on the principles of *mathematical induction.* The reasoning can increase your confidence that a recursive method avoids infinite recursion. As an example to show that there is no infinite recursion, let's rewrite the method `power` so that it uses more recursion.

The revision is based on the observation that for any number x and any positive integer n, the following relation holds: $x^n = x(x^{n-1})$. This formula means that an alternative way to define x^n is as follows:

- The value of x^n is undefined when $n \leq 0$ and $x = 0$.
- Otherwise, the value is 0 when $x = 0$.
- Otherwise, the value is 1 when n = 0.
- Otherwise, the value is x times x^{n-1} when $n > 0$.
- Otherwise, the value is $1/x^{-n}$ when $n < 0$.

an alternative algorithm for computing powers

The Java version of a recursive method that computes in this way is given in Figure 8.11. To avoid confusion, we have used a slightly different name, `pow`, for this version of the method.

Tracing a recursive method such as pow can quickly overwhelm you, but there are relatively simple ways of showing that there is no infinite recursion without actually tracing through the execution. The most common way to ensure that a stopping case is *eventually* reached is to define a numeric quantity called the *variant expression*. This quantity must associate each legal recursive call to a single number. In a moment, we'll discuss the properties that the variant expression should have, but first let's look at the kind of quantity we have in mind for the variant expression of the pow method. The variant expression for pow depends on whether n is negative or not. For a negative n, the variant expression is abs(n) + 1, which is one more than the absolute value of n. For a non-negative n, the variant expression is just the value of n itself.

With this definition, we can examine a sequence of recursive pow calls, beginning with pow(2.0, -3), as follows:

A Sequence of Recursive Calls

1. pow(2.0, -3) has a variant expression abs(n) + 1, which is 4; it makes a recursive call of pow(2.0, 3).
2. pow(2.0, 3) has a variant expression n, which is 3; it makes a recursive call of pow(2.0, 2).
3. pow(2.0, 2) has a variant expression n, which is 2; it makes a recursive call of pow(2.0, 1).
4. pow(2.0, 1) has a variant expression n, which is 1; it makes a recursive call of pow(2.0, 0).
5. pow(2.0, 0) has a variant expression n, which is 0; this is the stopping case.

There are two important points to this example: (a) Each time a recursive call is made, the variant expression is reduced by at least one; and (b) When the variant expression reaches zero, there is a stopping case that terminates with no further recursive calls.

variant expression and threshold

In general, a **variant expression** is a numeric quantity that is decreased by at least some fixed amount on each recursive call. Once the variant expression reaches a small enough value, a stopping case occurs. The "small enough value" that guarantees a stopping case is called the **threshold**.

Variant Expression and Threshold

A **variant expression** is a numeric quantity that is decreased by at least some fixed amount on each recursive call. Once the variant expression reaches a small enough value (called the **threshold**), then the stopping case occurs.

In the pow example, the threshold is zero, and each recursive call reduces the variant expression by one. A summary of the general technique for proving that a recursive call terminates is shown here:

Ensuring That There Is No Infinite Recursion

To prove that a recursive call does not lead to infinite recursion, it is enough to find a *variant expression* and a *threshold* with the following properties:

1. Between one call of the method and any succeeding recursive call of that method, the value of the variant expression decreases by at least some fixed amount.
2. If the method is activated and the value of the variant expression is less than or equal to the threshold, then the method terminates without making any recursive calls.

It is important that the reduction is at least a fixed amount. Otherwise, the variant expression might start at 1, then decrease to one-half, then to one-quarter, then to one-eighth, and so on, decreasing by ever-smaller amounts and never reach the threshold. In the most common case, such as pow, the variant expression always decreases by at least one, and the threshold is zero.

To see that these two conditions guarantee no infinite recursion, reason as follows. Suppose the two conditions hold. Since Condition 1 is true, every recursive call will decrease the variant expression. This means that either the method will terminate, which is fine, or the variant expression will decrease until it reaches the threshold. But if Condition 2 holds, then once the variant expression reaches the threshold, the method will terminate. That covers all the cases.

Inductive Reasoning About the Correctness of a Recursive Method

induction

In addition to checking that a recursive method terminates, you should also check that it always behaves correctly—in other words, that it meets its precondition/postcondition contract. The usual method for showing correctness of a recursive method is called **induction**. (And, in fact, the technique is the same as *mathematical induction*, which you may have used in math classes.) The induction approach requires a programmer to demonstrate the following facts about the method's behavior:

Induction Method to Show That a Recursive Method Is Correct

To show that a recursive method meets its precondition/postcondition contract, first show that there is no infinite recursion (by showing Conditions 1 and 2) and then show that the following two conditions are also valid:

3. Whenever the method makes no recursive calls, then it meets its precondition/postcondition contract. (This is called the **base step**.)
4. Whenever the method is activated *and* all the recursive calls it makes meet their precondition/postcondition contract, then the original call will also meet its precondition/postcondition contract. (This is called the **induction step**.)

The conditions are numbered 3 and 4 to emphasize that they ensure correctness only if you know that there is no infinite recursion. You must also ensure that Conditions 1 and 2 hold for an appropriate variant expression and threshold.

Let's return to the method pow defined in Figure 8.11 on page 423. To complete our demonstration that it performs as desired, we must show that 3 and 4 hold.

It is easy to see that Condition 3 holds. The only way that the method can terminate without a recursive call is if the value of x is zero or n is zero. If x is zero, the method returns 0, which is the correct answer; if n is zero (and x is not zero), the method returns 1, which is also correct.

To see that Condition 4 holds, we need only recall the algebraic identities:

$$x^n = x(x^{n-1}) \quad \text{and} \quad x^n = 1/x^{-n}$$

To summarize how to reason about recursion: First check that the method always terminates (no infinite recursion); next make sure the stopping cases work correctly; finally, for each recursive case, pretend that you know the recursive calls will work correctly and use this to show that each recursive case works correctly.

Self-Test Exercises for Section 8.3

9. Write a recursive method that computes the number of digits in an integer *n*. (You might recall from page 23 that this is $\lfloor \log_{10} n \rfloor + 1$ for positive numbers.) Do not use any local variables in your method declaration. Find a variant expression and threshold to show that your method has no infinite recursion.

10. Use inductive reasoning to show that your method from the preceding exercise is always correct.
11. Find variant expressions and thresholds to show that the methods `randomFractal` and `traverseMaze` (Section 8.2) never result in infinite recursion.
12. Use induction to show that `randomFractal` meets its precondition/post-condition contract.
13. Rewrite the `pow` method using these two facts for x^n:
    - If n is positive and even, then $x^n = x^{n/2} \times x^{n/2}$.
    - If n is positive and odd, then $x^n = x \times x^{n-1}$.
14. Find a variant expression and threshold for the `bears` method from Figure 8.9 on page 419.
15. What kind of error is likely to occur if you write a method that results in infinite recursion?
16. Are there any methods that require recursion for their implementation?

CHAPTER SUMMARY

- If a problem can be reduced to smaller instances of the same problem, then a recursive solution is likely to be easy to find and implement.
- A recursive algorithm for a method implementation contains two kinds of cases: one or more cases that include a *recursive call* and one or more *stopping cases* in which the problem is solved without the use of any recursive calls.
- When writing recursive methods, always check to see that the method will not produce infinite recursion. This can be done by finding an appropriate *variant expression* and *threshold*.
- *Inductive reasoning* can be used to show that a recursive method meets its precondition/postcondition contract.

Solutions to Self-Test Exercises

1. The top-left method prints 3, then 2, then 1. The top-right method prints 1, then 2, then 3. The bottom method prints 3, then 2, then 1, then 1 again, then 2 again, then 3 again.

2. The output is `Hip`, then `Hip`, then `Hurrah`, on three separate lines.

3. For the first modification, change the two lines in the else-block to:

```
cheers(n-1);
System.out.println("Hip");
```

For the second modification, change the lines to:

```
System.out.println("Hip");
cheers(n-1);
System.out.println("Hip");
```

For the third modification, change the lines to:

```
if (n % 2 == 0)
   System.out.println("Hip");
cheers(n-1);
if (n % 2 == 1)
   System.out.println("Hip");
```

4. The method's implementation is:

```
public static void exercise4(int n)
{
    if (n > 0)
    {
        System.out.print('*');
        exercise4(n-1);
        System.out.print('!');
    }
}
```

5. The original activation makes two calls with an x-distance of 12. Each of those calls makes two calls with an x-distance of 6, so there are four calls with an x-distance of 6. Each of those four calls makes two more calls, again cutting the width in half, so there are eight calls with an x-distance of 3. These eight calls do not make further calls since `width` has reached the stopping point. The total number of calls, including the original call, is 1 + 2 + 4 + 8, which is 15 calls.

6. Did you peek?

7. The easiest solution requires an extra parameter called `level`, which indicates how deep the recursion has proceeded. When the revised method is called from a program, the value of `level` should be given as zero. Each recursive call increases the level by one. When the level is an even number, the midpoint is moved upward; when the level is odd, the midpoint is moved downward. The code to do the movement is as follows:

```
if (level % 2 == 0)
    midY += (rightX - leftX)/2;
else
    midY -= (rightX - leftX)/2;
```

8. Each recursive call steps forward into a location that has not previously been visited. Therefore, the number of calls can be no more than the number of locations in the maze, which is 200.

9. The method's implementation is:

```
int digits(int n)
{
    if (n < 10) && (n > -10))
        return 1;
    else
        return 1 + digits(n/10);
}
```

A good variant expression is "the number of digits in `n`," with the threshold of 1.

10. The stopping case includes numbers that are less than 10 and more than negative 10. All these numbers have one digit, and the method correctly returns the answer 1. For the induction case, we have a number *n* with more than one digit. The number of digits will always be one more than *n*/10 (using integer division), so if we assume that the recursive call of `digits(n/10)` returns the right answer, then `1 + digits(n/10)` is the correct number of digits in *n*.

11. For `randomFractal`, a good variant expression is `(rightX-leftX)`. If we make a recursive call, then this expression is greater than 4, and the recursive call cuts the expression in half. Therefore, each recursive call subtracts at least 2 from the expression. When the expression reaches 4 (or less), the recursion stops. Therefore, 4 is the threshold.

The method `traverseMaze` has a variant expression that is expressed in English as "the number of locations in the maze that do not yet have your name written on the ground." This value is reduced by at least one during each recursive call, and when this value reaches zero, there can be no further recursive calls. Therefore, 0 is the threshold.

12. We have already found a variant expression and threshold for Conditions 1 and 2, showing that `randomFractal` does not result in infinite recursion. For Condition 3, we must show that the method has correct behavior for the stopping case. In this case, `(rightX-leftX)` is no more than `STOP`, and therefore the line segment does not need further dividing. We only need to draw the current line segment, which is what the method does. In Condition 4 of the

inductive reasoning, we assume that the two recursive calls correctly generate a random fractal for the two smaller line segments we have created. Putting these two smaller random fractals together correctly gives us the larger random fractal.

13. In Figure 8.11 on page 423, we change the `(n > 0)` block to this:

```
if ((n > 0) && (n % 2 == 1))
   return x * pow(x, n-1);
else if ((n > 0) && (n % 2 == 0))
{
   double partial = pow(x, n/2);
   return partial * partial;
}
```

14. The variant expression is the number `n`, with a threshold of zero.

15. `StackOverflowError`

16. Explicit recursion can always be removed by using a stack to simulate the recursive calls.

PROGRAMMING PROJECTS

1 Write a method that produces the output shown below. This output was written by call number 1. In the example, the recursion stopped when it reached four levels deep, but your method should be capable of continuing to any specified level.

```
This was written by call number 2.
 This was written by call number 3.
  This was written by call number 4.
  This ALSO written by call number 4.
 This ALSO written by call number 3.
This ALSO written by call number 2.
This ALSO written by call number 1.
```

2 Write a method with two parameters, `prefix` (a string) and `levels` (a nonnegative integer). The method prints the string `prefix` followed by "section numbers" of the form 1.1., 1.2., 1.3., and so on. The `levels` argument determines how many levels the section numbers have. For example, if `levels` is 2, then the section numbers have the form `x.y`. If `levels` is 3, then the section numbers have the form `x.y.z`. The digits permitted in each level are always `'1'` through `'9'`. As an example, if `prefix` is the string `"BOX:"` and `levels` is 2, then the method would start by printing this:

```
BOX:1.1.
BOX:1.2.
BOX:1.3.
```

And finish by printing this:

```
BOX:9.7.
BOX:9.8.
BOX:9.9.
```

The stopping case occurs when `levels` reaches zero. The primary string manipulation technique that you will need is the ability to create a new string that consists of `prefix` followed by a digit and a period. If `s` is the string you want to create and `i` is the digit (an integer in the range `1` to `9`), then the following statement will perform this task:

```
s = prefix + '.' + i;
```

The last part of the expression puts the character that corresponds to the integer `i` onto the end of the string. This new string, `s`, can be passed as a parameter to recursive calls.

3 Write a recursive method that has two parameters, first and second, that are both strings. The method should print all rearrangements of the letters in `first` followed by `second`. For example, if `first` is the string `"CAT"` and `second` is the string `"MAN"`, then the method would print the strings `CATMAN`, `CTAMAN`, `ACTMAN`, `ATCMAN`, `TACMAN`, and `TCAMAN`. The stopping case of the method occurs when the length of `first` has zero characters. We'll leave the recursive thinking up to you, but we should mention three string techniques that will make things go smoother. These techniques are:

(1) The following expression is a string consisting of all of `first` followed by character number `i` from `second`:

```
first + second.charAt(i)
```

(2) The following expression is a string consisting of all of `second` with character `i` removed. The value of `i` must be less than the last index of the string. The first part of the expression is everything from location `0` to location `i-1`, and the second part of the expression is everything after location `i`.

```
second.substring(0, i)
+
second.substring(i+1);
```

(3) The following expression is a string consisting of all of `second` except the last character. For this to work, the string must be nonempty.

```
second.substring(0, s.length()-1)
```

The stopping case occurs when the length of the second string is zero (in which case you just print the first string). For the recursive case, make one recursive call for each character in the second string. During each recursive call, you take one character out of `second` and add it to the end of `first`.

4 Write an interactive program to help you count all of the boxes in a room. The program should begin by asking something like *How many unnumbered boxes can you see?* Then the program will have you number those boxes from 1 to *m*, where *m* is your answer. But remember that each box might have smaller boxes inside, so once the program knows you can see *m* boxes, it should ask you to open box number 1 and take out any boxes you find, numbering those boxes 1.1, 1.2, and so on. It will also ask you to open box number 2 and take out any boxes you find there, numbering those boxes 2.1, 2.2, and so on. This continues for box 3, 4, and so on up to *m*. And, of course, each time you number a box 1.1 or 3.8 or something similar, *that* box might have more boxes inside. Boxes that reside inside of 3.8 would be numbered 3.8.1, 3.8.2, and so on. At the end, the program should print a single number telling you the total number of boxes in the room.

5 Write a recursive method called `sumover` that has one argument *n*, which is a nonnegative integer. The method returns a double value, which is the sum of the reciprocals of the first *n* positive integers. (The reciprocal of *x* is the fraction 1/*x*.) For example, `sumover(1)` returns `1.0` (which is 1/1); `sumover(2)` returns `1.5` (which is 1/1 + 1/2); `sumover(3)` returns approximately `1.833` (which is 1/1 + 1/2 + 1/3). Define `sumover(0)` to be zero. Do not use any local variables in your method.

6 The formula for computing the number of ways of choosing *r* different things from a set of *n* things is the following:

$$C(n, r) = \frac{n!}{r!(n-r)!}$$

In this formula, the factorial function is represented by an exclamation point (!) and defined as the product:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

Discover a recursive version of the $C(n, r)$ formula and write a recursive method that computes the value of the formula. Embed the method in a program and test it.

7 Write a recursive method that has as arguments an array of characters and two bounds on array indexes. The method should reverse the order of those entries in the array whose

indexes are between the two bounds. For example, suppose the array is:

```
a[0] = 'A'  a[1] = 'B'  a[2] = 'C'
a[3] = 'D'  a[4] = 'E'
```

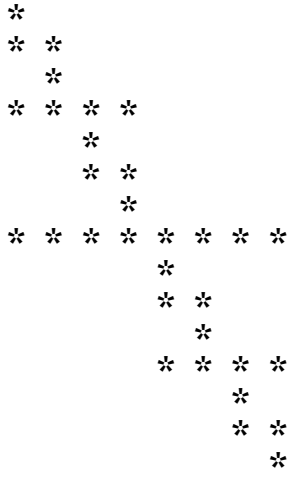
and the bounds are 1 and 4. Then after the method is run the array elements should be:

```
a[0] = 'A'  a[1] = 'D'  a[2] = 'C'
a[3] = 'B'  a[4] = 'e'
```

Embed the method in a program and test it.

8 Write a recursive method to produce a pattern of n lines of asterisks. The first line contains one asterisk, the next line contains two and so on up to the n^{th} line, which contains n asterisks. Line number n+1 again contains n asterisks, the next line has n–1 asterisks, and so on until line number 2n, which has just one asterisk.

9 Examine this pattern of asterisks and blanks and write a recursive method that can generate exactly this pattern:



With recursive thinking, the method needs only seven or eight lines of code (including two recursive calls). How is this pattern a fractal? Your method should also be capable of producing larger or smaller patterns of the same variety. Hint: Have two parameters. One parameter indicates the indentation of the leftmost line in the pattern; the other parameter indicates the number of stars in the longest line.

10 Write a program that asks the user to think of an integer between 1 and 1,000,000, and then guesses the number through a series of yes/no questions. To guess the number, the program calls a recursive method `guess` that has two parameters, `low` and `high`. The precondition for the method requires that the user's number lie in the range `low...high` so that the program's initial call is to `guess(1, 1000000)`. What is a good stopping case for `guess`, when it can guess the user's number with little or no work? Answer: If `(low == high)`, then there is only one possible number, and the method can guess that number. On the other hand, if `(low < high)`, then the method should calculate a point near the middle of the range:

```
midpoint = (low + high) / 2;
```

Then the method asks the user whether the midpoint is the correct number. If so, the method is finished. On the other hand, if the midpoint is not the user's number, then the method asks whether the correct number is larger than `midpoint`. If so, the method knows that the user's number lies in the range `midpoint + 1` to `high`, and a recursive call can be made to solve the smaller problem of finding a user's number in the range `midpoint + 1` to `high`. On the other hand, if the user's number is not larger than `midpoint`, then a recursive call can be made to solve the smaller problem of finding a user's number in the range `low` to `midpoint - 1`. This method of searching is called **binary search**, which we will explore further in Chapter 11.

11 This project uses the `Towers` class from Chapter 3's Programming Project 12 on page 164. For the project, write a recursive method that computes and prints a solution to the Towers of Hanoi game. The method should meet this specification:

```
public static void Hanoi(
    Towers t,
    int n,
    int start,
    int target,
    int spare
);
```

```
// Precondition: start, target, and spare are
// three different peg numbers of the game,
// and n is non-negative.
// Also, the n smallest rings currently are on
// the top of the start peg.
// Postcondition: The method has activated
// a sequence of moves for the tower t, so
// that the total effect of these moves is to
// shift the top n rings from the start peg to
// the target peg, using spare as the extra
// peg. None of the other rings has been
// disturbed. Each time a move is made,
// a message describing the move is
// printed.
```

Your solution should have a simple stopping case: When *n* is zero, there is no work to do. When *n* is more than zero, use a recursive call to move *n*–1 rings from the start peg to the spare peg. Then call the `move` method to move one ring from the start peg to the target peg. Finally, make another recursive call to move *n*–1 rings from the spare peg to the target peg.

12 For this project, you are to write a recursive method that prints all of the objects in a bag. Use the bag specification from Figure 4.12 on page 208, which is a bag of integers. The integers are to be printed in a random order—without using any of the bag private instance variables. For an empty bag, the method has no work to do (that is, the stopping case). For a nonempty bag, the method carries out four steps: (a) grab a random integer from the bag, storing it in a local variable `oneItem`; (b) print `oneItem`; (c) print any items that remain in the bag; (d) put `oneItem` back in the bag. You'll need to identify which of these steps is the "simpler version of the same problem."

13 Let's think about your computer science class for a moment. You might know several students, perhaps Judy, Jervis, Walter, and Michael. Each of those students knows several other students, and each of them knows more students and so on. Now, there is one particular student named Dor that you would like to meet. One way to meet Dor would be if you had a mutual acquaintance. You know Judy, and Judy knows Dor, so Judy could introduce you to Dor. Or there might be a longer path of acquaintances: For example, you know Judy, and Judy knows Harry, and Harry knows Cathy, and Cathy knows Dor. In this case, Judy can introduce you to Harry, Harry can introduce you to Cathy, and Cathy can introduce you to Dor.

Write an interactive program to help you figure out whether there is a path of acquaintances from you to Dor. The program should include a recursive method that has one argument, `person`, which is the name of a person in your class. The method determines whether there is a path of acquaintances from `person` to Dor. Hint: This problem is similar to the maze problem in Section 8.2, but beware of potential infinite recursion! One way to avoid infinite recursion is to include a bag of student names that keeps track of the names of the students you have already visited on your search for a path to Dor.

14 A *pretty print* program takes a program that may not be indented in any particular way and produces a copy with the same program indented so that bracket pairs ({ and }) line up with inner pairs indented more than outer pairs, so that if-else statements are indented as we have been doing, and so that other indenting is as we have been doing. Write a program that reads a Java program from one text file and produces a pretty print version of the program in a second text file. To make it easier, simply do this for each method, ignoring the things outside of methods.

15 Write a Java method that meets the following specification:

```
public static void digits(int c)
// Precondition: c is one of the ints 0 to 9.
// Postcondition: The method has printed a
// pattern of digits as follows:
// 1. If the parameter c is 0, then output is '0'.
// 2. For other values of c, the output consists of
// three parts:
// -- the output for the previous number (c-1);
// -- followed by the digit c itself;
// -- followed by a second copy of the output for
//      the previous number.
// There is no newline printed at the end of the
// output. Example output: digits(3) will print
// this to System.out: 010201030102010
```

16 Write a Java method with the following header:

```
public static void binaryPrint(int n)
```

The number `n` is non-negative. The method prints the value of `n` as a *binary* number. If `n` is zero, then a single zero is printed; otherwise no leading zeros are printed in the output. The `'\n'` character is NOT printed at the end of the output. Your implementation must be recursive and not use any local variables.

EXAMPLES:

`n=0` Output: `0`
`n=4` Output: `100`
`n=27` Output: `11011`

17 Write a Java method with the following header:

```
public static void numbers
(String prefix, int n)
```

The number `k` is non-negative. The argument called `prefix` is a `String` of 0s and 1s. The method prints a sequence of binary numbers. Each output number consists of the prefix followed by a suffix of exactly `k` more binary digits (0s or 1s). All possible combinations of the prefix and some `k`-digit suffix are printed. As an example, if prefix is the string `"00101"` and levels is 2, then the method would print the prefix followed by the 4 possible suffixes shown here:

0010100
0010101
0010110
0010111

The stopping case occurs when `k` reaches zero (in which case the prefix is printed once by itself followed by nothing else).

18 Rewrite the basic calculator program from Figure 6.5 on page 314 so that the method `ReadAndEvaluate` uses recursion instead of stacks.