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

Recursive Thinking

CHAPTER 8

"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

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**.

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

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
```

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:

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

```
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

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.

Activation record for first call to superWriteVertical

number: –36

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

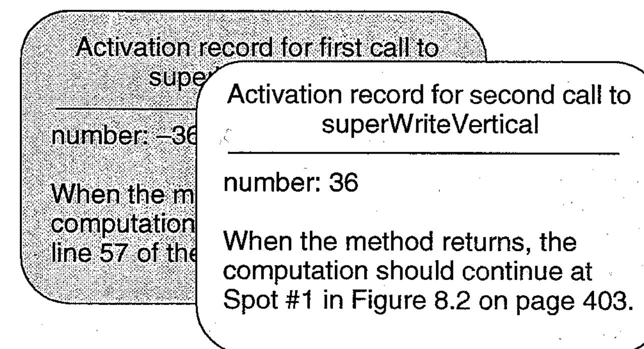
The return location specified in a real activation record does not actually refer to lines of code in the main program, but when you're imagining an activation record, you can think of a return location in this manner.

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:



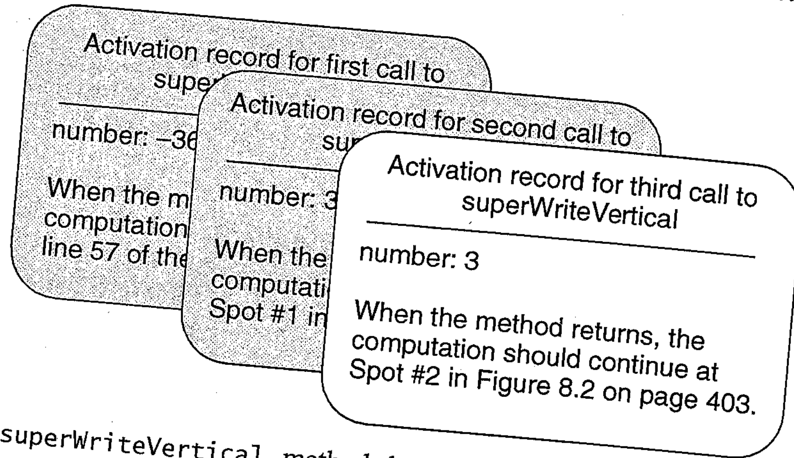
In fact, the collection of activation records is stored in a *stack* data structure called the **execution stack.** Each activation of a method pushes the next activation record on top of the execution stack.

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);
}
```

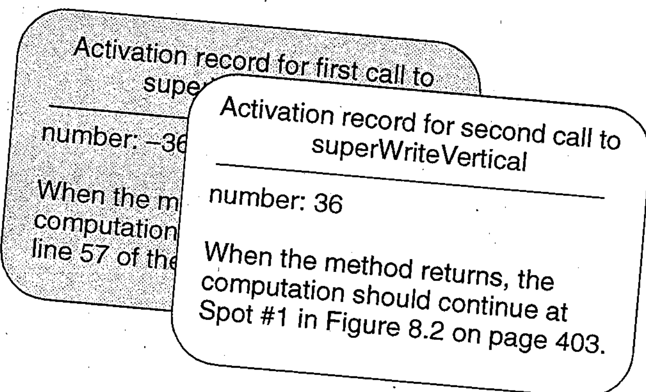
Another recursive call is made in the superWriteVertical method.

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 403. 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 403. 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 403. 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

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.

> **Key Design Concept**
> Find smaller versions of a problem within the larger problem itself.

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.

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.

FIGURE 8.3
The First Few Steps in Generating a Random Fractal