

Chapter 13 Recursion

CHAPTER GOALS

- To learn about the method of recursion
- To understand the relationship between recursion and iteration
- To analyze problems that are much easier to solve by recursion than by iteration
- To learn to “think recursively”
- To be able to use recursive helper methods
- To understand when the use of recursion affects the efficiency of an algorithm

The method of recursion is a powerful technique to break up complex computational problems into simpler ones. The term “recursion” refers to the fact that the same computation recurs, or occurs repeatedly, as the problem is solved. Recursion is often the most natural way of thinking about a problem, and there are some computations that are very difficult to perform without recursion. This chapter shows you simple and complex examples of recursion and teaches you how to “think recursively”.

587

588

13.1 Triangle Numbers

We begin this chapter with a very simple example that demonstrates the power of thinking recursively. In this example, we will look at triangle shapes such as the ones from [Section 6.3](#). We'd like to compute the area of a triangle of width n , assuming that each [] square has area 1. This value is sometimes called the n^{th} *triangle number*. For example, as you can tell from looking at

```
[]  
[] []  
[] [] []
```

the third triangle number is 6.

Java Concepts, 5th Edition

You may know that there is a very simple formula to compute these numbers, but you should pretend for now that you don't know about it. The ultimate purpose of this section is not to compute triangle numbers, but to learn about the concept of recursion in a simple situation.

Here is the the class that we will develop:

```
public class Triangle
{
    public Triangle(int aWidth)
    {
        width = aWidth;
    }
    public int getArea()
    {
        ...
    }
    private int width;
}
```

If the width of the triangle is 1, then the triangle consists of a single square, and its area is 1. Let's take care of this case first.

588

```
public int getArea()
{
    if (width == 1) return 1;
    ...
}
```

589

To deal with the general case, consider this picture.

```

[]
[] []
[] [] []
[] [] [] []
```

Suppose we knew the area of the smaller, colored triangle. Then we could easily compute the area of the larger triangle as

```
smallerArea + width
```

How can we get the smaller area? Let's make a smaller triangle and ask it!

```
Triangle smallerTriangle = new Triangle(width - 1);
```

Java Concepts, 5th Edition

```
int smallerArea = smallerTriangle.getArea();
```

Now we can complete the `getArea` method:

```
public int getArea()
{
    if (width == 1) return 1;
    Triangle smallerTriangle = new Triangle(width
- 1);
    int smallerArea = smallerTriangle.getArea();
    return smallerArea + width;
}
```

A recursive computation solves a problem by using the solution of the same problem with simpler values.

Here is an illustration of what happens when we compute the area of a triangle of width 4.

- The `getArea` method makes a smaller triangle of width 3.
 - It calls `getArea` on that triangle.
 - That method makes a smaller triangle of width 2.
 - It calls `getArea` on that triangle.
 - That method makes a smaller triangle of width 1.
 - It calls `getArea` on that triangle.
 - That method returns 1.
 - The method returns $\text{smallerArea} + \text{width} = 1 + 2 = 3$.
 - The method returns $\text{smallerArea} + \text{width} = 3 + 3 = 6$.
- The method returns $\text{smallerArea} + \text{width} = 6 + 4 = 10$.

This solution has one remarkable aspect. To solve the area problem for a triangle of a given width, we use the fact that we can solve the same problem for a lesser width. This is called a *recursive* solution.

The call pattern of a *recursive method* looks complicated, and the key to the successful design of a recursive method is *not to think about it*. Instead, look at the `getArea` method one more time and notice how utterly reasonable it is. If the width is 1, then, of course, the area is 1. The next part is just as reasonable. Compute the area of the smaller triangle *and don't think about why that works*. Then the area of the larger triangle is clearly the sum of the smaller area and the width. 589
590

There are two key requirements to make sure that the recursion is successful:

- Every recursive call must simplify the computation in some way.
- There must be special cases to handle the simplest computations directly.

The `getArea` method calls itself again with smaller and smaller width values. Eventually the width must reach 1, and there is a special case for computing the area of a triangle with width 1. Thus, the `getArea` method always succeeds.

For a recursion to terminate, there must be special cases for the simplest values.

Actually, you have to be careful. What happens when you call the area of a triangle with width -1 ? It computes the area of a triangle with width -2 , which computes the area of a triangle with width -3 , and so on. To avoid this, the `getArea` method should return 0 if the width is ≤ 0 .

Recursion is not really necessary to compute the triangle numbers. The area of a triangle equals the sum

$$1 + 2 + 3 + \dots + \text{width}$$

Of course, we can program a simple loop:

```
double area = 0;
for (int i = 1; i <= width; i++)
    area = area + i;
```

Many simple recursions can be computed as loops. However, loop equivalents for more complex recursions—such as the one in our next example—can be complex.

Actually, in this case, you don't even need a loop to compute the answer. The sum of the first n integers can be computed as

$$1 + 2 + \dots + n = n \times (n + 1) / 2$$

Thus, the area equals

$$\text{width} * (\text{width} + 1) / 2$$

Therefore, neither recursion nor a loop is required to solve this problem. The recursive solution is intended as a “warm-up” to introduce you to the concept of recursion.

ch13/triangle/Triangle.java

```
1  /**
2   A triangular shape composed of stacked unit squares like this:
3       []
4       [] []
5       [] [] []
6       ...
7   */
8   public class Triangle
9   {
10      /**
11       Constructs a triangular shape.
12       @param aWidth the width (and height) of the triangle
13       */
14      public Triangle(int aWidth)
15      {
16          width = aWidth;
17      }
18
19      /**
20       Computes the area of the triangle.
21       @return the area
22       */
23      public int getArea()
24      {
25          if (width <= 0) return 0;
26          if (width == 1) return 1;
27          Triangle smallerTriangle = new
Triangle(width - 1);
```

590

591

Java Concepts, 5th Edition

```
28         int smallerArea =
smallerTriangle.getArea();
29         return smallerArea + width;
30     }
31
32     private int width;
33 }
```

ch13/triangle/TriangleTester.java

```
1  public class TriangleTester
2  {
3      public static void main(String[] args)
4      {
5          Triangle t = new Triangle(10);
6          int area = t.getArea();
7          System.out.println("Area: " + area);
8          System.out.println("Expected: 55");
9      }
10 }
```

Output

```
Enter width: 10
Area: 55
Expected: 55
```

SELF CHECK

1. Why is the statement `if (width == 1) return 1;` in the `getArea` method unnecessary?
2. How would you modify the program to recursively compute the area of a square?

591

592

COMMON ERROR 13.1: Infinite Recursion

A common programming error is an infinite recursion: a method calling itself over and over with no end in sight. The computer needs some amount of memory for bookkeeping for each call. After some number of calls, all memory that is

available for this purpose is exhausted. Your program shuts down and reports a “stack fault”.

Infinite recursion happens either because the parameter values don't get simpler or because a special terminating case is missing. For example, suppose the `getArea` method computes the area of a triangle with width 0. If it wasn't for the special test, the method would have constructed triangles with width -1 , -2 , -3 , and so on.

13.2 Permutations

We will now turn to a more complex example of recursion that would be difficult to program with a simple loop. We will design a class that lists all permutations of a string. A *permutation* is simply a rearrangement of the letters. For example, the string “eat” has six permutations (including the original string itself):

```
"eat"  
"eta"  
"aet"  
"ate"  
"tea"  
"tae"
```

As in the preceding section, we will define a class that is in charge of computing the answer. In this case, the answer is not a single number but a collection of permuted strings. Here is our class:

```
public class PermutationGenerator  
{  
    public PermutationGenerator(String aWord) { ... }  
    ArrayList<String> getPermutations() { ... }  
}
```

Here is the test program that prints out all permutations of the string “eat”:

ch13/permute/PermutationGeneratorDemo.java

```
1  import java.util.ArrayList;  
2  
3  /**  
4   This program demonstrates the permutation generator.  
5   */
```

Java Concepts, 5th Edition

6	<code>public class</code> PermutationGeneratorDemo	592
7	{	
8	<code>public static void</code> main(String[] args)	593
9	{	
10	PermutationGenerator generator	
11	= new	
	PermutationGenerator("eat");	
12	ArrayList<String> permutations =	
	generator.getPermutations();	
13	for (String s : permutations)	
14	{	
15	System.out.println(s);	
16	}	
17	}	
18	}	

Output

```
eat
eta
aet
ate
tea
tae
```

Now we need a way to generate the permutations recursively. Consider the string "eat". Let's simplify the problem. First, we'll generate all permutations that start with the letter 'e', then those that start with 'a', and finally those that start with 't'. How do we generate the permutations that start with 'e'? We need to know the permutations of the substring "at". But that's the same problem—to generate all permutations—with a simpler input, namely the shorter string "at". Thus, we can use recursion. Generate the permutations of the substring "at". They are

```
"at"
"ta"
```

For each permutation of that substring, prepend the letter 'e' to get the permutations of "eat" that start with 'e', namely

```
"eat"
"eta"
```


Java Concepts, 5th Edition

Now let's turn our attention to the permutations of "eat" that start with 'a'. We need to produce the permutations of the remaining letters, "et". They are:

```
"et"  
"te"
```

We add the letter 'a' to the front of the strings and obtain

```
"aet"  
"ate"
```

We generate the permutations that start with 't' in the same way.

That's the idea. The implementation is fairly straightforward. In the `get-Permutations` method, we loop through all positions in the word to be permuted. For each of them, we compute the shorter word that is obtained by removing the *i*th letter: 593

```
String shorterWord = word.substring(0, i) +  
word.substring(i + 1);
```

We construct a permutation generator to get the permutations of the shorter word, and ask it to give us all permutations of the shorter word.

```
PermutationGenerator shorterPermutationGenerator  
    = new PermutationGenerator(shorterWord);  
ArrayList<String> shorterWordPermutations  
    = shorterPermutationGenerator.getPermutations();
```

Finally, we add the removed letter to the front of all permutations of the shorter word.

```
for (String s : shorterWordPermutations)  
{  
    result.add(word.charAt(i) + s);  
}
```

As always, we have to provide a special case for the simplest strings. The simplest possible string is the empty string, which has a single permutation—itsself.

Here is the complete `PermutationGenerator` class.

ch13/permute/PermutationGenerator.java

```
1  import java.util.ArrayList;
2
3  /**
4      This class generates permutations of a
5      word.
6      */
7  public class PermutationGenerator
8  {
9      /**
10         Constructs a permutation generator.
11         @param aWord the word to permute
12         */
13     public PermutationGenerator(String aWord)
14     {
15         word = aWord;
16     }
17
18     /**
19         Gets all permutations of a given word.
20         */
21     public ArrayList<String> getPermutations()
22     {
23         ArrayList<String> result = new
24         ArrayList<String>();
25
26         // The empty string has a single permutation: itself
27         if (word.length() == 0)
28         {
29             result.add(word);
30             return result;
31         }
32
33         // Loop through all character positions
34         for (int i = 0; i < word.length(); i++)
35         {
36             // Form a simpler word by removing the ith character
37             String shorterWord =
38             word.substring(0, i)
39             + word.substring(i + 1);
```

594

595

```
37
38     // Generate all permutations of the simpler word
39     PermutationGenerator
shorterPermutationGenerator
40     = new
PermutationGenerator(shorterWord);
41     ArrayList<String>
shorterWordPermutations
42     =
shorterPermutationGenerator.getPermutations();
43
44     // Add the removed character to the front of
45     // each permutation of the simpler word
46     for (String s :
shorterWordPermutations)
47     {
48         result.add(word.charAt(i) +
s);
49     }
50 }
51 // Return all permutations
52     return result;
53 }
54
55 private String word;
56 }
```

Compare the `PermutationGenerator` and `Triangle` classes. Both of them work on the same principle. When they work on a more complex input, they first solve the problem for a simpler input. Then they combine the result for the simpler input with additional work to deliver the results for the more complex input. There really is no particular complexity behind that process as long as you think about the solution on that level only. However, behind the scenes, the simpler input creates even simpler input, which creates yet another simplification, and so on, until one input is so simple that the result can be obtained without further help. It is interesting to think about this process, but it can also be confusing. What's important is that you can focus on the one level that matters—putting a solution together from the slightly simpler problem, ignoring the fact that it also uses recursion to get its results.

SELF CHECK

3. What are all permutations of the four-letter word beat?
4. Our recursion for the permutation generator stops at the empty string. What simple modification would make the recursion stop at strings of length 0 or 1?

595

596

COMMON ERROR 13.2: Tracing Through Recursive Methods

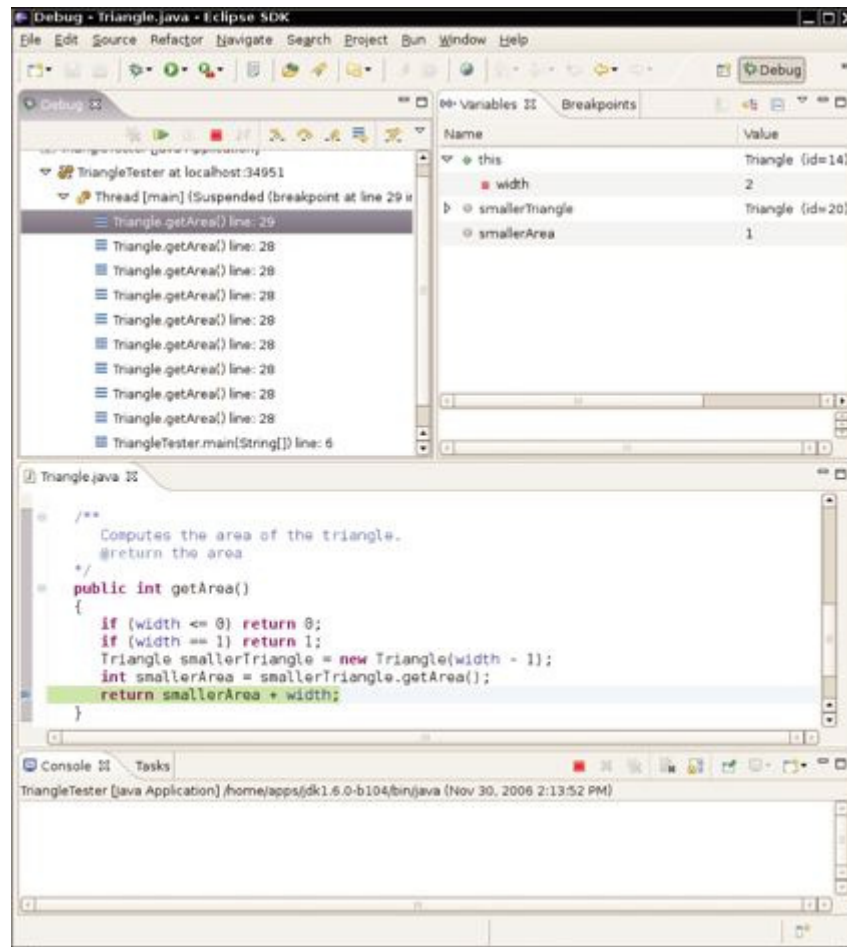
Debugging a recursive method can be somewhat challenging. When you set a breakpoint in a recursive method, the program stops as soon as that program line is encountered in *any call to the recursive method*. Suppose you want to debug the recursive `getArea` method of the `Triangle` class. Debug the `TriangleTester` program and run until the beginning of the `getArea` method. Inspect the `width` instance variable. It is 10.

Remove the breakpoint and now run until the statement `return smallerArea + width;` (see [Figure 1](#)). When you inspect `width` again, its value is 2! That makes no sense. There was no instruction that changed the value of `width`. Is that a bug with the debugger?

No. The program stopped in the first recursive call to `getArea` that reached the `return` statement. If you are confused, look at the *call stack* (top left in the figure). You will see that nine calls to `getArea` are pending.

You can debug recursive methods with the debugger. You just need to be particularly careful, and watch the call stack to understand which nested call you currently are in.

Figure 1



Debugging a Recursive Method

596

597

How To 13.1: Thinking Recursively

To solve a problem recursively requires a different mindset than to solve it by programming loops. In fact, it helps if you are, or pretend to be, a bit lazy and like others to do most of the work for you. If you need to solve a complex problem, pretend that “someone else” will do most of the heavy lifting and solve the

problem for all simpler inputs. Then you only need to figure out how you can turn the solutions with simpler inputs into a solution for the whole problem.

To illustrate the method of recursion, let us consider the following problem. We want to test whether a sentence is a *palindrome*—a string that is equal to itself when you reverse all characters. Typical examples of palindromes are

- A man, a plan, a canal—Panama!
- Go hang a salami, I'm a lasagna hog

and, of course, the oldest palindrome of all:

- Madam, I'm Adam

When testing for a palindrome, we match upper- and lowercase letters, and ignore all spaces and punctuation marks.

We want to implement the `isPalindrome` method in the following class:

```
public class Sentence
{
    /**
     * Constructs a sentence.
     * @param aText a string containing all characters of the
     * sentence
     */
    public Sentence(String aText)
    {
        text = aText;
    }
    /**
     * Tests whether this sentence is a palindrome.
     * @return true if this sentence is a palindrome, false
     * otherwise
     */
    public boolean isPalindrome()
    {
        ...
    }
    private String text;
}
```

Step 1 Consider various ways to simplify inputs.

In your mind, fix a particular input or set of inputs for the problem that you want to solve.

Think how you can simplify the inputs in such a way that the same problem can be applied to the simpler input.

When you consider simpler inputs, you may want to remove just a little bit from the original input—maybe remove one or two characters from a string, or remove a small portion of a geometric shape. But sometimes it is more useful to cut the input in half and then see what it means to solve the problem for both halves.

597

598

In the palindrome test problem, the input is the string that we need to test. How can you simplify the input? Here are several possibilities:

- Remove the first character.
- Remove the last character.
- Remove both the first and last characters.
- Remove a character from the middle.
- Cut the string into two halves.

These simpler inputs are all potential inputs for the palindrome test.

Step 2 Combine solutions with simpler inputs into a solution of the original problem.

In your mind, consider the solutions of your problem for the simpler inputs that you discovered in Step 1. Don't worry *how* those solutions are obtained. Simply have faith that the solutions are readily available. Just say to yourself: These are simpler inputs, so someone else will solve the problem for me.

Now think how you can turn the solution for the simpler inputs into a solution for the input that you are currently thinking about. Maybe you need to add a small quantity, related to the quantity that you lopped off to arrive at the simpler input.

Java Concepts, 5th Edition

Maybe you cut the original input in half and have solutions for each half. Then you may need to add both solutions to arrive at a solution for the whole.

Consider the methods for simplifying the inputs for the palindrome test. Cutting the string in half doesn't seem a good idea. If you cut

```
"Madam, I 'm Adam"
```

in half, you get two strings:

```
"Madam, I "
```

and

```
" 'm Adam"
```

Neither of them is a palindrome. Cutting the input in half and testing whether the halves are palindromes seems a dead end.

The most promising simplification is to remove the first *and* last characters.

Removing the M at the front and the m at the back yields

```
"adam, I 'm Ada"
```

Suppose you can verify that the shorter string is a palindrome. Then *of course* the original string is a palindrome—we put the same letter in the front and the back. That's extremely promising. A word is a palindrome if

- The first and last letters match (ignoring letter case)

and

- The word obtained by removing the first and last letters is a palindrome.

Again, don't worry how the test works for the shorter string. It just works.

There is one other case to consider. What if the first or last letter of the word is not a letter? For example, the string

```
"A man, a plan, a canal, Panama!"
```

598

ends in a ! character, which does not match the A in the front. But we should ignore non-letters when testing for palindromes. Thus, when the last character is

599

not a letter but the first character is a letter, it doesn't make sense to remove both the first and the last characters. That's not a problem. Remove only the last character. If the shorter string is a palindrome, then it stays a palindrome when you attach a nonletter.

The same argument applies if the first character is not a letter. Now we have a complete set of cases.

- If the first and last characters are both letters, then check whether they match. If so, remove both and test the shorter string.
- Otherwise, if the last character isn't a letter, remove it and test the shorter string.
- Otherwise, the first character isn't a letter. Remove it and test the shorter string.

In all three cases, you can use the solution to the simpler problem to arrive at a solution to your problem.

Step 3 Find solutions to the simplest inputs.

A recursive computation keeps simplifying its inputs. Eventually it arrives at very simple inputs. To make sure that the recursion comes to a stop, you must deal with the simplest inputs separately. Come up with special solutions for them, which is usually very easy.

However, sometimes you get into philosophical questions dealing with *degenerate* inputs: empty strings, shapes with no area, and so on. Then you may want to investigate a slightly larger input that gets reduced to such a trivial input and see what value you should attach to the degenerate inputs so that the simpler value, when used according to the rules you discovered in Step 2, yields the correct answer.

Let's look at the simplest strings for the palindrome test:

- Strings with two characters
- Strings with a single character

- The empty string

We don't have to come up with a special solution for strings with two characters. Step 2 still applies to those strings—either or both of the characters are removed. But we do need to worry about strings of length 0 and 1. In those cases, Step 2 can't apply. There aren't two characters to remove.

The empty string is a palindrome—it's the same string when you read it backwards. If you find that too artificial, consider a string "mm". According to the rule discovered in Step 2, this string is a palindrome if the first and last characters of that string match and the remainder—that is, the empty string—is also a palindrome. Therefore, it makes sense to consider the empty string a palindrome.

A string with a single letter, such as "I", is a palindrome. How about the case in which the character is not a letter, such as "!"? Removing the ! yields the empty string, which is a palindrome. Thus, we conclude that all strings of length 0 or 1 are palindromes.

Step 4 Implement the solution by combining the simple cases and the reduction step.

Now you are ready to implement the solution. Make separate cases for the simple inputs that you considered in Step 3. If the input isn't one of the simplest cases, then implement the logic you discovered in Step 2.

Here is the `isPalindrome` method.

```
public boolean isPalindrome()
{
    int length = text.length();
    // Separate case for shortest strings.
    if (length <= 1) return true;
    // Get first and last characters, converted to lowercase.
    char first =
Character.toLowerCase(text.charAt(0));
    char last =
Character.toLowerCase(text.charAt(length - 1));
    if (Character.isLetter(first) &&
Character.isLetter(last))
    {
```

599

600

```
        // Both are letters.
        if (first == last)
        {
            // Remove both first and last character.
            Sentence shorter = new
Sentence(text.substring(1, length - 1));
            return shorter.isPalindrome();
        }
        else
            return false;
    }
    else if (!Character.isLetter(last))
    {
        // Remove last character.
        Sentence shorter = new
Sentence(text.substring(0, length - 1));
        return shorter.isPalindrome();
    }
    else
    {
        // Remove first character.
        Sentence shorter = new
Sentence(text.substring(1));
        return shorter.isPalindrome();
    }
}
```

13.3 Recursive Helper Methods

Sometimes it is easier to find a recursive solution if you change the original problem slightly. Then the original problem can be solved by calling a recursive helper method.

Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Here is a typical example. Consider the palindrome test of How To 13.1. It is a bit inefficient to construct new `Sentence` objects in every step. Now consider the following change in the problem. Rather than testing whether the entire sentence is a palindrome, let's check whether a substring is a palindrome:

600

/**

601

Java Concepts, 5th Edition

Tests whether a substring of the sentence is a palindrome.

@param start the index of the first character of the substring

@param end the index of the last character of the substring

@return true if the substring is a palindrome

```
*/  
public boolean isPalindrome(int start, int end)
```

This method turns out to be even easier to implement than the original test. In the recursive calls, simply adjust the `start` and `end` parameters to skip over matching letter pairs and characters that are not letters. There is no need to construct new `Sentence` objects to represent the shorter strings.

```
public boolean isPalindrome(int start, int end)  
{  
    // Separate case for substrings of length 0 and 1.  
    if (start >= end) return true;  
    // Get first and last characters, converted to lowercase.  
    char first =  
Character.toLowerCase(text.charAt(start));  
    char last =  
Character.toLowerCase(text.charAt(end));  
    if (Character.isLetter(first) &&  
Character.isLetter(last))  
    {  
        if (first == last)  
        {  
            // Test substring that doesn't contain the matching letters.  
            return isPalindrome(start + 1, end - 1);  
        }  
        else  
            return false;  
    }  
    else if (!Character.isLetter(last))  
    {  
        // Test substring that doesn't contain the last character.  
        return isPalindrome(start, end - 1);  
    }  
    else  
    {  
        // Test substring that doesn't contain the first character.  
        return isPalindrome(start + 1, end);  
    }  
}
```

```
}
```

You should still supply a method to solve the whole problem—the user of your method shouldn't have to know about the trick with the substring positions. Simply call the helper method with positions that test the entire string:

```
public boolean isPalindrome()  
{  
    return isPalindrome(0, text.length() - 1);  
}
```

Note that this call is *not* a recursive method. The `isPalindrome()` method calls the helper method `isPalindrome(int, int)`. In this example, we use overloading to define two methods with the same name. The `isPalindrome` method without parameters is the method that we expect the public to use. The second method, with two `int` parameters, is the recursive helper method. If you prefer, you can avoid overloaded methods by choosing a different name for the helper method, such as `substringIsPalindrome`.

601

602

Use the technique of recursive helper methods whenever it is easier to solve a recursive problem that is slightly different from the original problem.

SELF CHECK

5. Do we have to give the same name to both `isPalindrome` methods?
6. When does the recursive `isPalindrome` method stop calling itself?

13.4 The Efficiency of Recursion

As you have seen in this chapter, recursion can be a powerful tool to implement complex algorithms. On the other hand, recursion can lead to algorithms that perform poorly. In this section, we will analyze the question of when recursion is beneficial and when it is inefficient.

Consider the Fibonacci sequence introduced in Exercise P6.4: a sequence of numbers defined by the equation

$$f_1 = 1$$

$$f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

That is, each value of the sequence is the sum of the two preceding values. The first ten terms of the sequence are

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

It is easy to extend this sequence indefinitely. Just keep appending the sum of the last two values of the sequence. For example, the next entry is $34 + 55 = 89$.

We would like to write a function that computes f_n for any value of n . Let us translate the definition directly into a recursive method:

ch13/fib/RecursiveFib.java

```
1  import java.util.Scanner;
2
3  /**
4   This program computes Fibonacci numbers using a recursive
5   method.
6   */
7  public class RecursiveFib
8  {
9      public static void main(String[] args)
10     {
11         Scanner in = new Scanner(System.in);
12         System.out.print("Enter n: ");
13         int n = in.nextInt();
14
15         for (int i = 1; i <= n; i++)
16         {
17             long f = fib(i);
18             System.out.println("fib(" + i +
19 ") = " + f);
20         }
```

602

603

Java Concepts, 5th Edition

```
21
22     /**
23     Computes a Fibonacci number.
24     @param n an integer
25     @return the nth Fibonacci number
26     */
27     public static long fib(int n)
28     {
29         if (n <= 2) return 1;
30         else return fib(n - 1) + fib(n - 2);
31     }
32 }
```

Output

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
...
fib(50) = 12586269025
```

That is certainly simple, and the method will work correctly. But watch the output closely as you run the test program. The first few calls to the `fib` method are quite fast. For larger values, though, the program pauses an amazingly long time between outputs.

That makes no sense. Armed with pencil, paper, and a pocket calculator you could calculate these numbers pretty quickly, so it shouldn't take the computer anywhere near that long.

To find out the problem, let us insert *trace messages* into the method:

603

ch13/fib/RecursiveFibTracer.java

604

```
1     import java.util.Scanner;
2
```

Java Concepts, 5th Edition

```
3  /**
4   This program prints trace messages that show how often the
5   recursive method for computing Fibonacci numbers calls itself.
6   */
7   public class RecursiveFibTracer
8   {
9       public static void main(String[] args)
10      {
11          Scanner in = new Scanner(System.in);
12          System.out.print("Enter n: ");
13          int n = in.nextInt();
14
15          long f = fib(n);
16
17          System.out.println("fib(" + n + ") =
" + f);
18      }
19
20      /**
21       Computes a Fibonacci number.
22       @param n an integer
23       @return the nth Fibonacci number
24       */
25      public static long fib(int n)
26      {
27          System.out.println("Entering fib: n =
" + n);
28          long f;
29          if (n <= 2) f = 1;
30          else f = fib(n - 1) + fib(n - 2);
31          System.out.println("Exiting fib: n =
" + n
32                           + " return value = " + f);
33          return f;
34      }
35  }
```

Output

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
```

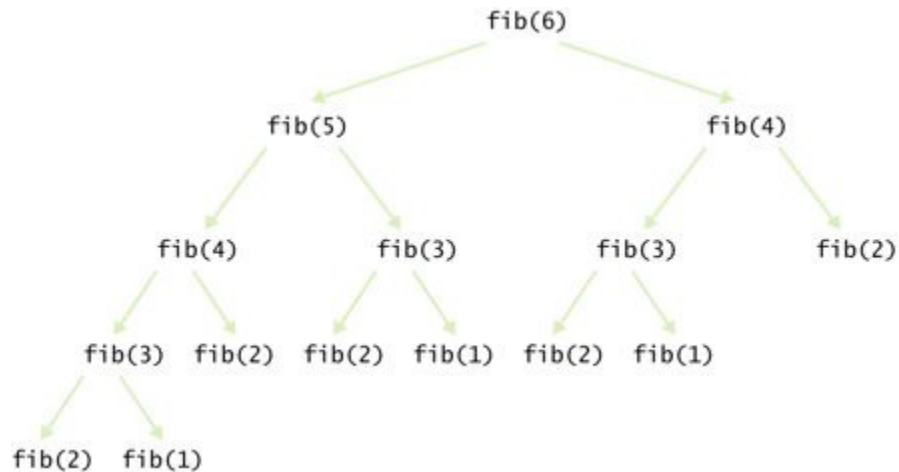

Java Concepts, 5th Edition

Entering fib: n = 4 Entering fib: n = 3 Entering fib: n = 2 Exiting fib: n = 2 return value = 1 Entering fib: n = 1 Exiting fib: n = 1 return value = 1 Exiting fib: n = 3 return value = 2 Entering fib: n = 2 Exiting fib: n = 2 return value = 1 Exiting fib: n = 4 return value = 3	604
Entering fib: n = 3 Entering fib: n = 2 Exiting fib: n = 2 return value = 1 Entering fib: n = 1 Exiting fib: n = 1 return value = 1 Exiting fib: n = 3 return value = 2 Exiting fib: n = 5 return value = 5 Entering fib: n = 4 Entering fib: n = 3 Entering fib: n = 2 Exiting fib: n = 2 return value = 1 Entering fib: n = 1 Exiting fib: n = 1 return value = 1 Exiting fib: n = 3 return value = 2 Entering fib: n = 2 Exiting fib: n = 2 return value = 1 Exiting fib: n = 4 return value = 3 Exiting fib: n = 6 return value = 8 fib(6) = 8	605

[Figure 2](#) shows the call tree for computing `fib(6)`. Now it is becoming apparent why the method takes so long. It is computing the same values over and over. For example, the computation of `fib(6)` calls `fib(4)` twice and `fib(3)` three times. That is very different from the computation we would do with pencil and paper. There we would just write down the values as they were computed and add up the last two to get the next one until we reached the desired entry; no sequence value would ever be computed twice.

If we imitate the pencil-and-paper process, then we get the following program.

Figure 2



Call Pattern of the Recursive fib Method

605

606

ch13/fib/LoopFib.java

```
1  import java.util.Scanner;
2
3  /**
4   This program computes Fibonacci numbers using an iterative method.
5   */
6  public class LoopFib
7  {
8      public static void main(String[] args)
9      {
10         Scanner in = new Scanner(System.in);
11         System.out.print("Enter n: ");
12         int n = in.nextInt();
13
14         for (int i = 1; i <= n; i++)
15         {
16             long f = fib(i);
17             System.out.println("fib(" + i +
18 ") = " + f);
19         }
20     }
```

Java Concepts, 5th Edition

```
21      /**
22      Computes a Fibonacci number.
23      @param n an integer
24      @return the nth Fibonacci number
25      */
26      public static long fib(int n)
27      {
28          if (n <= 2) return 1;
29          long fold = 1;
30          long fold2 = 1;
31          long fnew = 1;
32          for (int i = 3; i <= n; i++)
33          {
34              fnew = fold + fold2;
35              fold2 = fold;
36              fold = fnew;
37          }
38          return fnew;
39      }
40 }
```

Output

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
...
fib(50) = 12586269025
```

606

This method runs *much* faster than the recursive version.

607

In this example of the `fib` method, the recursive solution was easy to program because it exactly followed the mathematical definition, but it ran far more slowly than the iterative solution, because it computed many intermediate results multiple times.

Can you always speed up a recursive solution by changing it into a loop? Frequently, the iterative and recursive solution have essentially the same performance. For example, here is an iterative solution for the palindrome test.

```
public boolean isPalindrome()
{
    int start = 0;
    int end = text.length() - 1;
    while (start < end)
    {
        char first =
        Character.toLowerCase(text.charAt(start));
        char last =
        Character.toLowerCase(text.charAt(end));
        if (Character.isLetter(first) &&
            Character.isLetter(last))
        {
            // Both are letters.
            if (first == last)
            {
                start++;
                end--;
            }
            else
                return false;
        }
        if (!Character.isLetter(last))
            end--;
        if (!Character.isLetter(first))
            start++;
    }
    return true;
}
```

This solution keeps two index variables: `start` and `end`. The first index starts at the beginning of the string and is advanced whenever a letter has been matched or a nonletter has been ignored. The second index starts at the end of the string and moves toward the beginning. When the two index variables meet, the iteration stops.

Both the iteration and the recursion run at about the same speed. If a palindrome has n characters, the iteration executes the loop between $n/2$ and n times, depending on how many of the characters are letters, since one or both index variables are moved in each

Java Concepts, 5th Edition

step. Similarly, the recursive solution calls itself between $n/2$ and n times, because one or two characters are removed in each step.

Occasionally, a recursive solution runs much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.

In such a situation, the iterative solution tends to be a bit faster, because each recursive method call takes a certain amount of processor time. In principle, it is possible for a smart compiler to avoid recursive method calls if they follow simple patterns, but most compilers don't do that. From that point of view, an iterative solution is preferable.

607

There are quite a few problems that are dramatically easier to solve recursively than iteratively. For example, it is not at all obvious how you can come up with a nonrecursive solution for the permutation generator. As Exercise P13.11 shows, it is possible to avoid the recursion, but the resulting solution is quite complex (and no faster).

608

In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

Often, recursive solutions are easier to understand and implement correctly than their iterative counterparts. There is a certain elegance and economy of thought to recursive solutions that makes them more appealing. As the computer scientist (and creator of the Ghost-Script interpreter for the PostScript graphics description language) L. Peter Deutsch put it: “To iterate is human, to recurse divine”.

SELF CHECK

7. You can compute the factorial function either with a loop, using the definition that $n! = 1 \times 2 \times \dots \times n$, or recursively, using the definition that $0! = 1$ and $n! = (n - 1)! \times n$. Is the recursive approach inefficient in this case?
8. Why isn't it easy to develop an iterative solution for the permutation generator?



RANDOM FACT 13.1: The Limits of Computation

Have you ever wondered how your instructor or grader makes sure your programming homework is correct? In all likelihood, they look at your solution and perhaps run it with some test inputs. But usually they have a correct solution available. That suggests that there might be an easier way. Perhaps they could feed your program and their correct program into a “program comparator”, a computer program that analyzes both programs and determines whether they both compute the same results. Of course, your solution and the program that is known to be correct need not be identical—what matters is that they produce the same output when given the same input.

How could such a program comparator work? Well, the Java compiler knows how to read a program and make sense of the classes, methods, and statements. So it seems plausible that someone could, with some effort, write a program that reads two Java programs, analyzes what they do, and determines whether they solve the same task. Of course, such a program would be very attractive to instructors, because it could automate the grading process. Thus, even though no such program exists today, it might be tempting to try to develop one and sell it to universities around the world.

However, before you start raising venture capital for such an effort, you should know that theoretical computer scientists have proven that it is impossible to develop such a program, *no matter how hard you try*.

There are quite a few of these unsolvable problems. The first one, called the *halting problem*, was discovered by the British researcher Alan Turing in 1936 (see photo below). Because his research occurred before the first actual computer was constructed, Turing had to devise a theoretical device, the *Turing machine*, to explain how computers could work. The Turing machine consists of a long magnetic tape, a read/write head, and a program that has numbered instructions of the form: "If the current symbol under the head is x , then replace it with y , move the head one unit left or right, and continue with instruction n " (see A Turing Machine). Interestingly enough, with only these instructions, you can program just as much as with Java, even though it is incredibly tedious to do so. Theoretical computer scientists like Turing machines because they can be described using nothing more than the laws of mathematics.

608

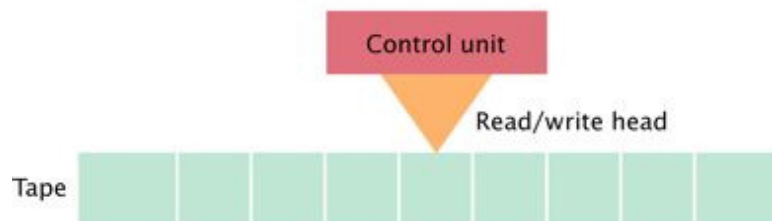
609



Alan Turing

Program

Instruction number	If tape symbol is	Replace with	Then move head	Then go to instruction
1	0	2	right	2
1	1	1	left	4
2	0	0	right	2
2	1	1	right	2
2	2	0	left	3
3	0	0	left	3
3	1	1	left	3
3	2	2	right	1
4	1	1	right	5
4	2	0	left	4



A Turing Machine

Expressed in terms of Java, the halting problem states: “It is impossible to write a program with two inputs, namely the source code of an arbitrary Java program P and a string I , and that decides whether the program P , when executed with the input I , will halt without getting into an infinite loop”. Of course, for some kinds of programs and inputs, it is possible to decide whether the program halts with the given input. The halting problem asserts that it is impossible to come up with a single decision-making algorithm that works with all programs and inputs. Note that you can't simply run the program P on the input I to settle this question. If the program runs for 1,000 days, you don't know that the program is in an infinite loop. Maybe you just have to wait another day for it to stop.

Such a “halt checker”, if it could be written, might also be useful for grading homework. An instructor could use it to screen student submissions to see if they get into an infinite loop with a particular input, and then stop checking them. However, as Turing demonstrated, such a program cannot be written. His argument is ingenious and quite simple.

Suppose a “halt checker” program existed. Let's call it H . From H , we will develop another program, the “killer” program K . K does the following computation. Its input is a string containing the source code for a program R . It then applies the halt checker on the input program R and the input string R . That is, it checks whether the program R halts if its input is its own source code. It sounds bizarre to feed a program to itself, but it isn't impossible. For example, the Java compiler is written in Java, and you can use it to compile itself. Or, as a simpler example, a word counting program can count the words in its own source code.

When K gets the answer from H that R halts when applied to itself, it is programmed to enter an infinite loop. Otherwise K exits. In Java, the program might look like this:

```
public class Killer
{
    public static void main(String[] args)
    {
        String r = read_program_input();
        HaltChecker checker = new HaltChecker();
        if (checker.check(r, r))
            while (true) {} // Infinite loop
        else
            return;
    }
}
```



```
    }  
}
```

Now ask yourself: What does the halt checker answer when asked whether *K* halts when given *K* as the input? Maybe it finds out that *K* gets into an infinite loop with such an input. But wait, that can't be right. That would mean that `checker.check(r, r)` returns `false` when *r* is the program code of *K*. As you can plainly see, in that case, the `killer` method returns, so *K* didn't get into an infinite loop. That shows that *K* must halt when analyzing itself, so `checker.check(r, r)` should return `true`. But then the `killer` method doesn't terminate—it goes into an infinite loop. That shows that it is logically impossible to implement a program that can check whether *every* program halts on a particular input.

It is sobering to know that there are *limits* to computing. There are problems that no computer program, no matter how ingenious, can answer.

Theoretical computer scientists are working on other research involving the nature of computation. One important question that remains unsettled to this day deals with problems that in practice are very time-consuming to solve. It may be that these problems are intrinsically hard, in which case it would be pointless to try to look for better algorithms. Such theoretical research can have important practical applications. For example, right now, nobody knows whether the most common encryption schemes used today could be broken by discovering a new algorithm (see Random Fact 19.1 for more information on encryption algorithms). Knowing that no fast algorithms exist for breaking a particular code could make us feel more comfortable about the security of encryption.

610

611

13.5 Mutual Recursions

In the preceding examples, a method called itself to solve a simpler problem. Sometimes, a set of cooperating methods calls each other in a recursive fashion. In this section, we will explore a typical situation of such a mutual recursion. This technique is significantly more advanced than the simple recursion that we discussed in the preceding sections. Feel free to skip this section if this is your first exposure to recursion.

In a mutual recursion, a set of cooperating methods calls each other repeatedly.

We will develop a program that can compute the values of arithmetic expressions such as

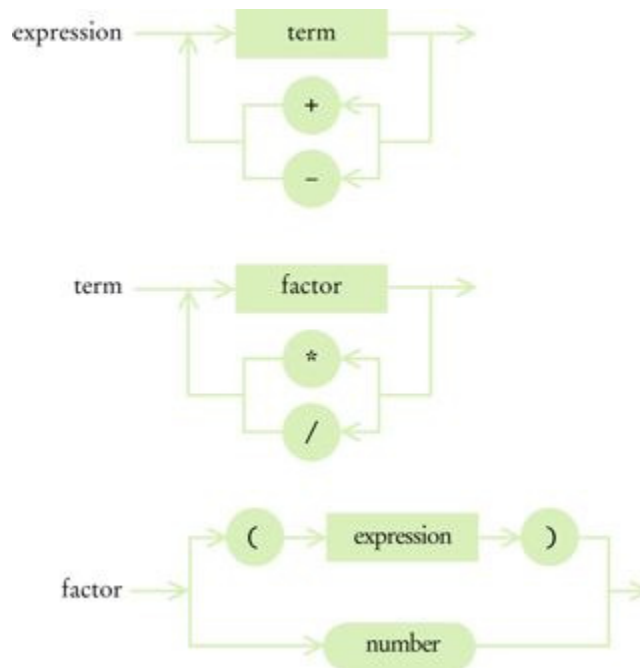
```
3+4*5
(3+4)*5
1-(2-(3-(4-5)))
```

Computing such an expression is complicated by the fact that `*` and `/` bind more strongly than `+` and `-`, and that parentheses can be used to group subexpressions.

[Figure 3](#) shows a set of *syntax diagrams* that describes the syntax of these expressions. To see how the syntax diagrams work, consider the expression `3+4*5`. 611

When you enter the *expression* syntax diagram, the arrow points directly to *term*, 612
giving you no alternative but to enter the *term* syntax diagram. The arrow points to *factor*, again giving you no choice. You enter the *factor* diagram, and now you have two choices: to follow the top branch or the bottom branch. Because the first input token is the number `3` and not a `(`, you must follow the bottom branch. You accept the input token because it matches the *number*. Follow the arrow out of *number* to the end of *factor*. Just like in a method call, you now back up, returning to the end of the *factor* element of the *term* diagram. Now you have another choice—to loop back in the *term* diagram, or to exit. The next input token is a `+`, and it matches neither the `*` or the `/` that would be required to loop back. So you exit, returning to *expression*. Again, you have a choice, to loop back or to exit. Now the `+` matches one of the choices in the loop. Accept the `+` in the input and move back to the *term* element.

Figure 3



Syntax Diagrams for Evaluating an Expression

In this fashion, an expression is broken down into a sequence of terms, separated by + or −, each term is broken down into a sequence of factors, each separated by * or /, and each factor is either a parenthesized expression or a number. You can draw this breakdown as a tree. [Figure 4](#) shows how the expressions $3+4*5$ and $(3+4)*5$ are derived from the syntax diagram.

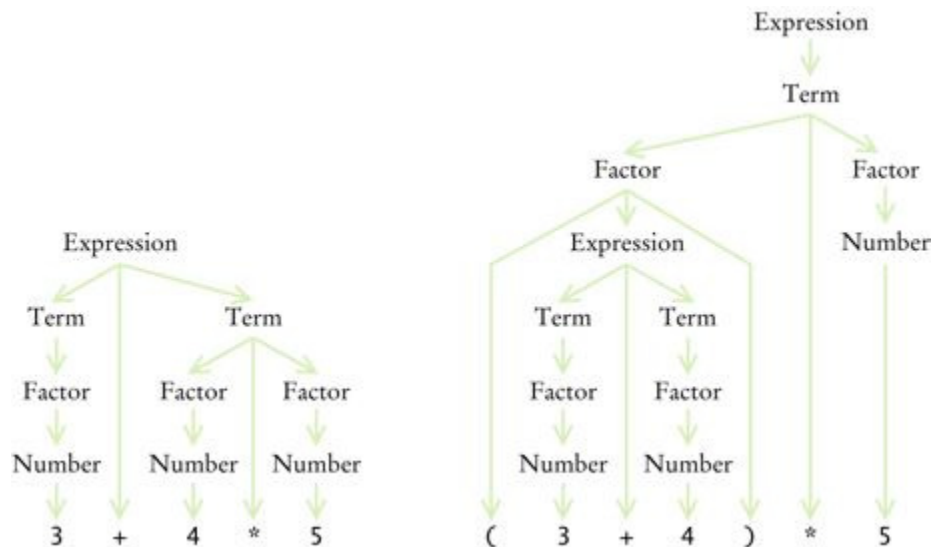
Why do the syntax diagrams help us compute the value of the tree? If you look at the syntax trees, you will see that they accurately represent which operations should be carried out first. In the first tree, 4 and 5 should be multiplied, and then the result should be added to 3. In the second tree, 3 and 4 should be added, and the result should be multiplied by 5.

At the end of this section, you will find the implementation of the `Evaluator` class, which evaluates these expressions. The `Evaluator` makes use of an `Expression-Tokenizer` class, which breaks up an input string into tokens—

Java Concepts, 5th Edition

numbers, operators, and parentheses. (For simplicity, we only accept positive integers as numbers, and we don't allow spaces in the input.)

Figure 4



Syntax Trees for Two Expressions

612

When you call `nextToken`, the next input token is returned as a string. We also supply another method, `peekToken`, which allows you to see the next token without consuming it. To see why the `peekToken` method is necessary, consider the syntax diagram of the `factor` type. If the next token is a `"*"` or `"/"`, you want to continue adding and subtracting terms. But if the next token is another character, such as a `"+"` or `"-"`, you want to stop without actually consuming it, so that the token can be considered later.

613

To compute the value of an expression, we implement three methods: `getExpressionValue`, `getTermValue`, and `getFactorValue`. The `getExpressionValue` method first calls `getTermValue` to get the value of the first term of the expression. Then it checks whether the next input token is one of `+` or `-`. If so, it calls `getTermValue` again and adds or subtracts it.

```
public int getExpressionValue()
{
    int value = getTermValue();
```

Java Concepts, 5th Edition

```
        boolean done = false;
        while (!done)
        {
            String next = tokenizer.peekToken();
            if ("+".equals(next) || "-".equals(next))
            {
                tokenizer.nextToken(); // Discard "+" or
                "- "
                int value2 = getTermValue();
                if ("+".equals(next)) value = value +
                value2;
                else value = value - value2;
            }
            else done = true;
        }
        return value;
    }
}
```

The `getTermValue` method calls `getFactorValue` in the same way, multiplying or dividing the factor values.

Finally, the `getFactorValue` method checks whether the next input is a number, or whether it begins with a `(` token. In the first case, the value is simply the value of the number. However, in the second case, the `getFactorValue` method makes a recursive call to `getExpressionValue`. Thus, the three methods are mutually recursive.

```
public int getFactorValue()
{
    int value;
    String next = tokenizer.peekToken();
    if ("(".equals(next))
    {
        tokenizer.nextToken(); // Discard "("
        value = getExpressionValue();
        tokenizer.nextToken(); // Discard ")"
    }
    else
        value =
        Integer.parseInt(tokenizer.nextToken());
    return value;
}
```

613

To see the mutual recursion clearly, trace through the expression $(3+4) * 5$:

614

Java Concepts, 5th Edition

- `getExpressionValue` calls `getTermValue`
 - `getTermValue` calls `getFactorValue`
 - `getFactorValue` consumes the `(` input
 - `getFactorValue` calls `getExpressionValue`
 - `getExpressionValue` returns eventually with the value of 7, having consumed `3 + 4`. This is the recursive call.
 - `getFactorValue` consumes the `)` input
 - `getFactorValue` returns 7
 - `getTermValue` consumes the inputs `*` and 5 and returns 35
- `getExpressionValue` returns 35

As always with a recursive solution, you need to ensure that the recursion terminates. In this situation, that is easy to see. If `getExpressionValue` calls itself, the second call works on a shorter subexpression than the original expression. At each recursive call, at least some of the tokens of the input string are consumed, so eventually the recursion must come to an end.

ch13/expr/Evaluator.java

```
1  /**
2   A class that can compute the value of an arithmetic expression.
3   */
4  public class Evaluator
5  {
6      /**
7       Constructs an evaluator.
8       @param anExpression a string containing the
9       expression
10      to be evaluated
11      */
12      public Evaluator(String anExpression)
13      {
```

```
13         tokenizer = new
ExpressionTokenizer(anExpression);
14     }
15
16     /**
17     Evaluates the expression.
18     @return the value of the expression
19     */
20     public int getExpressionValue()
21     {
22         int value = getTermValue();
23         boolean done = false;
24         while (!done)
25         {
26             String next =
tokenizer.peekToken();
27             if ("+".equals(next) ||
28             "-".equals(next))
29             {
30                 tokenizer.nextToken(); //
Discard "+" or "-"
31                 int value2 = getTermValue();
32                 if ("+".equals(next)) value =
value + value2;
33                 else value = value - value2;
34             }
35             else done = true;
36         }
37         return value;
38     }
39     /**
40     Evaluates the next term found in the expression.
41     @return the value of the term
42     */
43     public int getTermValue()
44     {
45         int value = getFactorValue();
46         boolean done = false;
47         while (!done)
48         {
```

614

615

```
49         String next =
tokenizer.peekToken();
50         if ("*".equals(next) ||
"/".equals(next))
51         {
52             tokenizer.nextToken();
53             int value2 = getFactorValue();
54             if ("*".equals(next)) value =
value * value2;
55             else value = value / value2;
56         }
57         else done = true;
58     }
59     return value;
60 }
61
62 /**
63  Evaluates the next factor found in the expression.
64  @return the value of the factor
65  */
66 public int getFactorValue()
67 {
68     int value;
69     String next = tokenizer.peekToken();
70     if ("(".equals(next))
71     {
72         tokenizer.nextToken(); // Discard "("
73         value = getExpressionValue();
74         tokenizer.nextToken(); // Discard ")"
75     }
76     else
77         value =
Integer.parseInt(tokenizer.nextToken());
78     return value;
79 }
80
81 private ExpressionTokenizer tokenizer;
82 }
```

615

616

ch13/expr/ExpressionTokenizer.java

1 /**


```
2  This class breaks up a string describing an expression
3  into tokens: numbers, parentheses, and operators.
4  */
5  public class ExpressionTokenizer
6  {
7      /**
8       Constructs a tokenizer.
9       @param anInput the string to tokenize
10      */
11     public ExpressionTokenizer(String anInput)
12     {
13         input = anInput;
14         start = 0;
15         end = 0;
16         nextToken();
17     }
18
19     /**
20     Peeks at the next token without consuming it.
21     @return the next token or null if there are no more
22     tokens
23     */
24     public String peekToken()
25     {
26         if (start >= input.length()) return
27         null;
28         else return input.substring(start,
29         end);
30     }
31
32     /**
33     Gets the next token and moves the tokenizer to the following token.
34     @return the next token or null if there are no more
35     tokens
36     */
37     public String nextToken()
38     {
39         String r = peekToken();
40         start = end;
41         if (start >= input.length()) return r;
```

Java Concepts, 5th Edition

```
38         if
(Character.isDigit(input.charAt(start)))
39     {
40         end = start + 1;
41         while (end < input.length()
42             &&
Character.isDigit(input.charAt(end)))
43             end++;
44     }
45     else
46         end = start + 1;
47     return r;
48 }
49
50 private String input;
51 private int start;
52 private int end;
53 }
```

616

ch13/expr/ExpressionCalculator.java

617

```
1  import java.util.Scanner;
2
3  /**
4   This program calculates the value of an expression
5   consisting of numbers, arithmetic
6   operators, and parentheses.
7   */
8  public class ExpressionCalculator
9  {
10     public static void main(String[] args)
11     {
12         Scanner in = new Scanner(System.in);
13         System.out.print("Enter an expression:
14 ");
15         String input = in.nextLine();
16         Evaluator e = new Evaluator(input);
17         int value = e.getExpressionValue();
18         System.out.println(input + "=" + value);
19     }
20 }
```

Output

```
Enter an expression: 3+4*5
3+4*5=23
```

SELF CHECK

- [9.](#) What is the difference between a term and a factor? Why do we need both concepts?
- [10.](#) Why does the expression parser use mutual recursion?
- [11.](#) What happens if you try to parse the illegal expression $3+4 *) 5$? Specifically, which method throws an exception?

CHAPTER SUMMARY

1. A recursive computation solves a problem by using the solution of the same problem with simpler values.
2. For a recursion to terminate, there must be special cases for the simplest values.
3. Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.
4. Occasionally, a recursive solution runs much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.
5. In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.
6. In a mutual recursion, a set of cooperating methods calls each other repeatedly.

617

REVIEW EXERCISES

618

★ Exercise R13.1. Define the terms

- a. Recursion
- b. Iteration

c. Infinite recursion

d. Recursive helper method

★★ **Exercise R13.2.** Outline, but do not implement, a recursive solution for finding the smallest value in an array.

★★ **Exercise R13.3.** Outline, but do not implement, a recursive solution for sorting an array of numbers. *Hint:* First find the smallest value in the array.

★★ **Exercise R13.4.** Outline, but do not implement, a recursive solution for generating all subsets of the set $\{1, 2, \dots, n\}$.

★★★ **Exercise R13.5.** Exercise P13.12 shows an iterative way of generating all permutations of the sequence $(0, 1, \dots, n-1)$. Explain why the algorithm produces the correct result.

★ **Exercise R13.6.** Write a recursive definition of x^n , where $n \geq 0$, similar to the recursive definition of the Fibonacci numbers. *Hint:* How do you compute x^n from x^{n-1} ? How does the recursion terminate?

★★ **Exercise R13.7.** Improve upon Exercise R13.6 by computing x^n as $(x^{n/2})^2$ if n is even. Why is this approach significantly faster? (*Hint:* Compute x^{1023} and x^{1024} both ways.)

★ **Exercise R13.8.** Write a recursive definition of $n! = 1 \times 2 \times \dots \times n$, similar to the recursive definition of the Fibonacci numbers.

★★ **Exercise R13.9.** Find out how often the recursive version of `fib` calls itself. Keep a static variable `fibCount` and increment it once in every call of `fib`. What is the relationship between `fib(n)` and `fibCount`?

★★★ **Exercise R13.10.** How many moves are required in the "Towers of Hanoi" problem of Exercise P13.13 to move n disks? *Hint:* As explained in the exercise,

$$\text{moves}(1) = 1$$

$$\text{moves}(n) = 2 \cdot \text{moves}(n - 1) + 1$$

PROGRAMMING EXERCISES

- ★ **Exercise P13.1.** Write a recursive method `void reverse()` that reverses a sentence. For example:

```
Sentence greeting = new Sentence("Hello!");  
greeting.reverse();  
System.out.println(greeting.getText());
```

prints the string `!olleH`. Implement a recursive solution by removing the first character, reversing a sentence consisting of the remaining text, and combining the two.

- ★★ **Exercise P13.2.** Redo Exercise P13.1 with a recursive helper method that reverses a substring of the message text.

- ★ **Exercise P13.3.** Implement the reverse method of Exercise P13.1 as an iteration.

- ★★ **Exercise P13.4.** Use recursion to implement a method `boolean find(String t)` that tests whether a string is contained in a sentence:

```
Sentence s = new Sentence("Mississippi!");  
boolean b = s.find("sip"); // Returns true
```

Hint: If the text starts with the string you want to match, then you are done. If not, consider the sentence that you obtain by removing the first character.

- ★★ **Exercise P13.5.** Use recursion to implement a method `int indexOf(String t)` that returns the starting position of the first substring of the text that matches `t`. Return `-1` if `t` is not a substring of `s`. For example,

```
Sentence s = new Sentence("Mississippi!");  
int n = s.indexOf("sip"); // Returns 6
```

Hint: This is a bit trickier than the preceding problem, because you must keep track of how far the match is from the beginning of the sentence. Make that value a parameter of a helper method.

- ★ **Exercise P13.6.** Using recursion, find the largest element in an array.

```
public class DataSet
{
    public DataSet(int[] values, int first, int
last) { ... }
    public int getMaximum() { ... }
    ...
}
```

Hint: Find the largest element in the subset containing all but the last element. Then compare that maximum to the value of the last element.

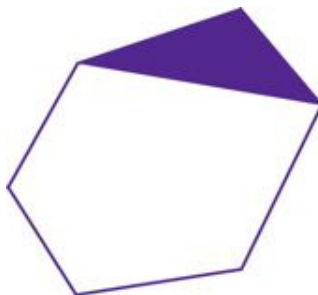
- ★ **Exercise P13.7.** Using recursion, compute the sum of all values in an array.

```
public class DataSet
{
    public DataSet(int[] values, int first, int
last) { ... }
    public int getSum() { ... }
    ...
}
```

619
620

- ★★ **Exercise P13.8.** Using recursion, compute the area of a polygon. Cut off a triangle and use the fact that a triangle with corners (x_1, y_1) , (x_2, y_2) , (x_3, y_3) has area

$$\frac{|x_1y_2 + x_2y_3 + x_3y_1 - y_1x_2 - y_2x_3 - y_3x_1|}{2}$$



★★★ **Exercise P13.9.** Implement a `SubstringGenerator` that generates all substrings of a string. For example, the substrings of the string "rum" are the seven strings

"r", "ru", "rum", "u", "um", "m", ""

Hint: First enumerate all substrings that start with the first character. There are n of them if the string has length n . Then enumerate the substrings of the string that you obtain by removing the first character.

★★★ **Exercise P13.10.** Implement a `SubsetGenerator` that generates all subsets of the characters of a string. For example, the subsets of the characters of the string "rum" are the eight strings

"rum", "ru", "rm", "r", "um", "u", "m", ""

Note that the subsets don't have to be substrings—for example, "rm" isn't a substring of "rum".

★★★ **Exercise P13.11.** In this exercise, you will change the `PermutationGenerator` of [Section 13.2](#) (which computed all permutations at once) to a `PermutationIterator` (which computes them one at a time.)

```
public class PermutationIterator
{
    public PermutationIterator(String s) { ... }
    public String nextPermutation() { ... }
    public boolean hasMorePermutations() { ... }
}
```

Here is how you would print out all permutations of the string "eat":

```
PermutationIterator iter = new
PermutationIterator("eat");
while (iter.hasMorePermutations())
    System.out.println(iter.nextPermutation());
```

620

621

Now we need a way to iterate through the permutations recursively. Consider the string "eat". As before, we'll generate all permutations that start with the letter 'e', then those that start with 'a', and finally those that start with 't'. How do we generate the permutations that start

Java Concepts, 5th Edition

with 'e'? Make another `PermutationIterator` object (called `tailIterator`) that iterates through the permutations of the substring "at". In the `nextPermutation` method, simply ask `tailIterator` what *its* next permutation is, and then add the 'e' at the front. However, there is one special case. When the tail generator runs out of permutations, all permutations that start with the current letter have been enumerated. Then

- Increment the current position.
- Compute the tail string that contains all letters except for the current one.
- Make a new permutation iterator for the tail string.

You are done when the current position has reached the end of the string.

★★★ **Exercise P13.12.** The following class generates all permutations of the numbers 0, 1, 2, ..., $n - 1$, without using recursion.

```
public class NumberPermutationIterator
{
    public NumberPermutationIterator(int n)
    {
        a = new int[n];
        done = false;
        for (int i = 0; i < n; i++) a[i] = i;
    }
    public int[] nextPermutation()
    {
        if (a.length <= 1) return a;
        for (int i = a.length - 1; i > 0; i--)
        {
            if (a[i - 1] < a[i])
            {
                int j = a.length - 1;
                while (a[i - 1] > a[j]) j--;
                swap(i - 1, j);
                reverse(i, a.length - 1);
                return a;
            }
        }
        return a;
    }
}
```



```
    }
    public boolean hasMorePermutations()
    {
        if (a.length <= 1) return false;
        for (int i = a.length - 1; i > 0; i--)
        {
            if (a[i - 1] < a[i]) return true;
        }
        return false;
    }
}
public void swap(int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
public void reverse(int i, int j)
{
    while (i < j) { swap(i, j); i++; j--; }
}
private int[] a;
}
```

621
622

The algorithm uses the fact that the set to be permuted consists of distinct numbers. Thus, you cannot use the same algorithm to compute the permutations of the characters in a string. You can, however, use this class to get all permutations of the character positions and then compute a string whose *i*th character is `word.charAt(a[i])`. Use this approach to reimplement the `PermutationIterator` of Exercise P13.11 without recursion.

★★ **Exercise P13.13.** *Towers of Hanoi.* This is a well-known puzzle. A stack of disks of decreasing size is to be transported from the leftmost peg to the rightmost peg. The middle peg can be used as temporary storage (see [Figure 5](#)). One disk can be moved at one time, from any peg to any other peg. You can place smaller disks only on top of larger ones, not the other way around.

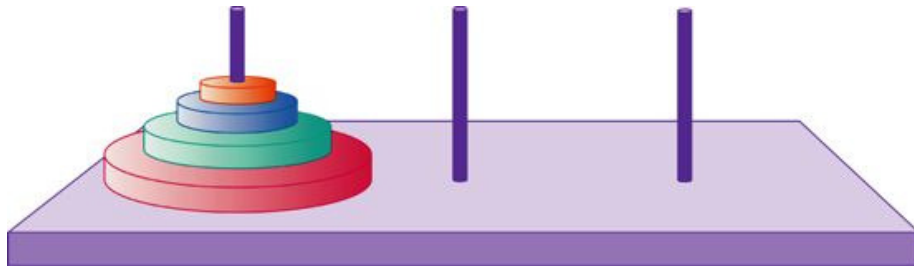
Write a program that prints the moves necessary to solve the puzzle for *n* disks. (Ask the user for *n* at the beginning of the program.) Print moves in the form

Move disk from peg 1 to peg 3

Hint: Implement a class `DiskMover`. The constructor takes

- The source peg from which to move the disks (1, 2, or 3)
- The target peg to which to move the disks (1, 2, or 3)
- The number of disks to move

Figure 5



Towers of Hanoi

622

A disk mover that moves a single disk from one peg to another simply has a `nextMove` method that returns a string

623

Move disk from peg *source* to peg *target*

A disk mover with more than one disk to move must work harder. It needs another `DiskMover` to help it. In the constructor, construct a `DiskMover(source, other, disks - 1)` where *other* is the peg other than *from* and *target*.

The `nextMove` asks that disk mover for its next move until it is done. The effect is to move the first `disks - 1` disks to the other peg. Then the `nextMove` method issues a command to move a disk from the *from* peg to the *to* peg. Finally, it constructs another disk mover `DiskMover(other, target, disks - 1)` that generates the moves that move the disks from the other peg to the target peg.

Hint: It helps to keep track of the state of the disk mover:

Java Concepts, 5th Edition

- `BEFORE_LARGEST`: The helper mover moves the smaller pile to the other peg.
- `LARGEST`: Move the largest disk from the source to the destination.
- `AFTER_LARGEST`: The helper mover moves the smaller pile from the other peg to the target.
- `DONE`: All moves are done.

Test your program as follows:

```
DiskMover mover = new DiskMover(1, 3, n);
while (mover.hasMoreMoves())
    System.out.println(mover.nextMove());
```

★★★ **Exercise P13.14.** *Escaping a Maze.* You are currently located inside a maze. The walls of the maze are indicated by asterisks (*).

```

*  * * * * *
*      *  *
*  * * * *  *
*  *  *    *
*  *  * *   *
*      *    *
* * * *  *  *
* * *  *  *  *
*      *    *
* * * * *  *

```

Use the following recursive approach to check whether you can escape from the maze: If you are at an exit, return `true`. Recursively check whether you can escape from one of the empty neighboring locations without visiting the current location. This method merely tests whether there is a path out of the maze. Extra credit if you can print out a path that leads to an exit.

★★★G **Exercise P13.15.** *The Koch Snowflake.* A snowflake-like shape is recursively defined as follows. Start with an equilateral triangle:

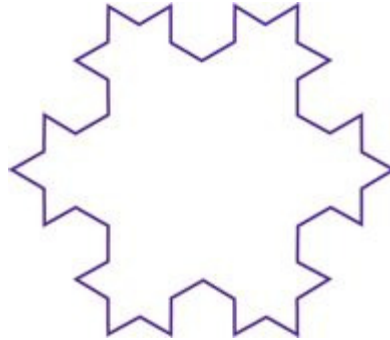


Next, increase the size by a factor of three and replace each straight line with four line segments.

623




Repeat the process.



Write a program that draws the iterations of this curve. Supply a button that, when clicked, produces the next iteration.

★★ **Exercise P13.16.** The recursive computation of Fibonacci numbers can be speeded up significantly by keeping track of the values that have already been computed. Provide an implementation of the `fib` method that uses this strategy. Whenever you return a new value, also store it in an auxiliary array. However, before embarking on a computation, consult the array to find whether the result has already been computed. Compare the running time of your improved implementation with that of the original recursive implementation and the loop implementation.

 Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 13.1.** Enhance the expression parser of [Section 13.5](#) to handle more sophisticated expressions, such as exponents, and mathematical functions, such as `sqrt` or `sin`.

- ★★★G **Project 13.2.** Implement a graphical version of the Towers of Hanoi program (see Exercise P13.13). Every time the user clicks on a button labeled "Next", draw the next move.

624

625

ANSWERS TO SELF-CHECK QUESTIONS

1. Suppose we omit the statement. When computing the area of a triangle with width 1, we compute the area of the triangle with width 0 as 0, and then add 1, to arrive at the correct area.

2. You would compute the smaller area recursively, then return

`smallerArea + width + width - 1.`



Of course, it would be simpler to compute the area simply as `width * width`. The results are identical because

$$1 + 0 + 2 + 1 + 3 + 2 + \dots + n + n - 1 = \frac{n(n+1)}{2} + \frac{(n-1)n}{2} = n^2.$$

3. They are b followed by the six permutations of eat, e followed by the six permutations of bat, a followed by the six permutations of bet, and t followed by the six permutations of bea.
4. Simply change `if (word.length() == 0)` to `if (word.length() <= 1)`, because a word with a single letter is also its sole permutation.
5. No—the first one could be given a different name such as `substringIsPalindrome`.
6. When `start >= end`, that is, when the investigated string is either empty or has length 1.

7. No, the recursive solution is about as efficient as the iterative approach. Both require $n - 1$ multiplications to compute $n!$.
8. An iterative solution would have a loop whose body computes the next permutation from the previous ones. But there is no obvious mechanism for getting the next permutation. For example, if you already found permutations `eat`, `eta`, and `aet`, it is not clear how you use that information to get the next permutation. Actually, there is an ingenious mechanism for doing just that, but it is far from obvious—see Exercise P13.12.
9. Factors are combined by multiplicative operators (`*` and `/`), terms are combined by additive operators (`+`, `-`). We need both so that multiplication can bind more strongly than addition.
10. To handle parenthesized expressions, such as `2+3*(4+5)`. The subexpression `4+5` is handled by a recursive call to `getExpressionValue`.
11. The `Integer.parseInt` call in `getFactorValue` throws an exception when it is given the string `)`.