CHAPTER

18

RECURSION

Objectives

- To describe what a recursive method is and the benefits of using recursion (§18.1).
- To develop recursive methods for recursive mathematical functions (§§18.2 and 18.3).
- To explain how recursive method calls are handled in a call stack (§§18.2 and 18.3).
- To solve problems using recursion (§18.4).
- To use an overloaded helper method to design a recursive method (§18.5).
- To implement a selection sort using recursion (§18.5.1).
- To implement a binary search using recursion (§18.5.2).
- To get the directory size using recursion (§18.6).
- To solve the Tower of Hanoi problem using recursion (§18.7).
- To draw fractals using recursion (§18.8).
- To discover the relationship and difference between recursion and iteration (§18.9).
- To know tail-recursive methods and why they are desirable (§18.10).

18.1 Introduction

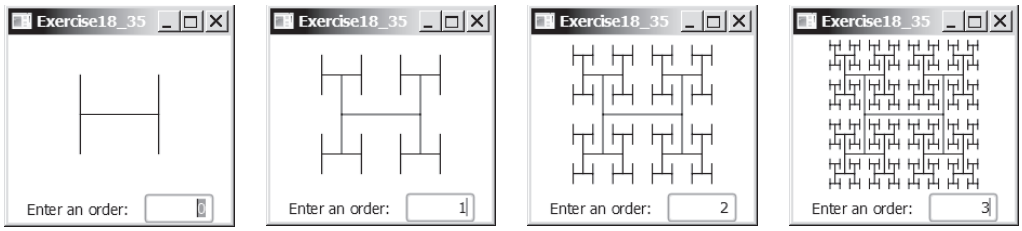
Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.

Suppose you want to find all the files under a directory that contains a particular word. How do you solve this problem? There are several ways to do so. An intuitive and effective solution is to use recursion by searching the files in the subdirectories recursively.

search word problem

H-tree problem

H-trees, depicted in Figure 18.1, are used in a very large-scale integration (VLSI) design as a clock distribution network for routing timing signals to all parts of a chip with equal propagation delays. How do you write a program to display H-trees? A good approach is to use recursion.



(a) (b) (c) (d)

FIGURE 18.1 An H-tree can be displayed using recursion. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

recursive method

To use recursion is to program using *recursive methods*—that is, to use methods that invoke themselves. Recursion is a useful programming technique. In some cases, it enables you to develop a natural, straightforward, simple solution to an otherwise difficult problem. This chapter introduces the concepts and techniques of recursive programming and illustrates with examples of how to "think recursively."

18.2 Case Study: Computing Factorials

A recursive method is one that invokes itself directly or indirectly.

Many mathematical functions are defined using recursion. Let's begin with a simple example. The factorial of a number `n` can be recursively defined as follows:

```
0! = 1;
n! = n × (n - 1)!; n > 0
```

How do you find `n!` for a given `n`? To find `1!` is easy because you know that `0!` is `1` and `1!` is `1 × 0!`. Assuming that you know `(n - 1)!`, you can obtain `n!` immediately by using `n × (n - 1)!`. Thus, the problem of computing `n!` is reduced to computing `(n - 1)!`. When computing `(n - 1)!`, you can apply the same idea recursively until `n` is reduced to `0`.

Let `factorial(n)` be the method for computing `n!`. If you call the method with `n = 0`, it immediately returns the result. The method knows how to solve the simplest case, which is referred to as the *base case* or the *stopping condition*. If you call the method with `n > 0`, it reduces the problem into a subproblem for computing the factorial of `n - 1`. The *subproblem* is essentially the same as the original problem, but it is simpler or smaller. Because the subproblem has the same property as the original problem, you can call the method with a different argument, which is referred to as a *recursive call*.

base case or stopping condition

recursive call

The recursive algorithm for computing `factorial(n)` can be simply described as follows:

```
if (n == 0)
  return 1;
```

```
else
  return n * factorial(n - 1);
```

A recursive call can result in many more recursive calls because the method keeps on dividing a subproblem into new subproblems. For a recursive method to terminate, the problem must eventually be reduced to a stopping case, at which point the method returns a result to its caller. The caller then performs a computation and returns the result to its own caller. This process continues until the result is passed back to the original caller. The original problem can now be solved by multiplying `n` by the result of `factorial(n - 1)`.

Listing 18.1 gives a complete program that prompts the user to enter a nonnegative integer and displays the factorial for the number.

LISTING 18.1 ComputeFactorial.java

```
1  import java.util.Scanner;
2
3  public class ComputeFactorial {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8      System.out.print("Enter a nonnegative integer: ");
9      int n = input.nextInt();
10
11     // Display factorial
12     System.out.println("Factorial of " + n + " is " + factorial(n));
13   }
14
15   /** Return the factorial for the specified number */
16   public static long factorial(int n) {
17     if (n == 0)  // Base case                                    base case
18       return 1;
19     else
20       return n *  factorial(n - 1); // Recursive call           recursion
21   }
22 }
```

```
Enter a nonnegative integer: 4 Enter
Factorial of 4 is 24
```

```
Enter a nonnegative integer: 10 Enter
Factorial of 10 is 3628800
```

The `factorial` method (lines 16–21) is essentially a direct translation of the recursive mathematical definition for the factorial into Java code. The call to `factorial` is recursive because it calls itself. The parameter passed to `factorial` is decremented until it reaches the base case of `0`.

You see how to write a recursive method. How does recursion work behind the scenes? how does it work?
Figure 18.2 illustrates the execution of the recursive calls, starting with `n = 4`. The use of stack space for recursive calls is shown in Figure 18.3.

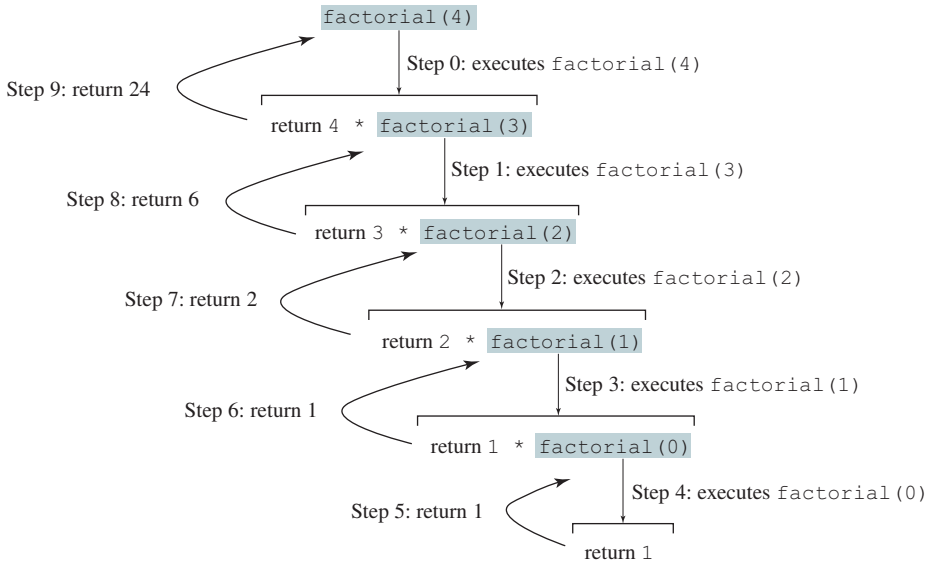


FIGURE 18.2 Invoking `factorial(4)` spawns recursive calls to `factorial`.

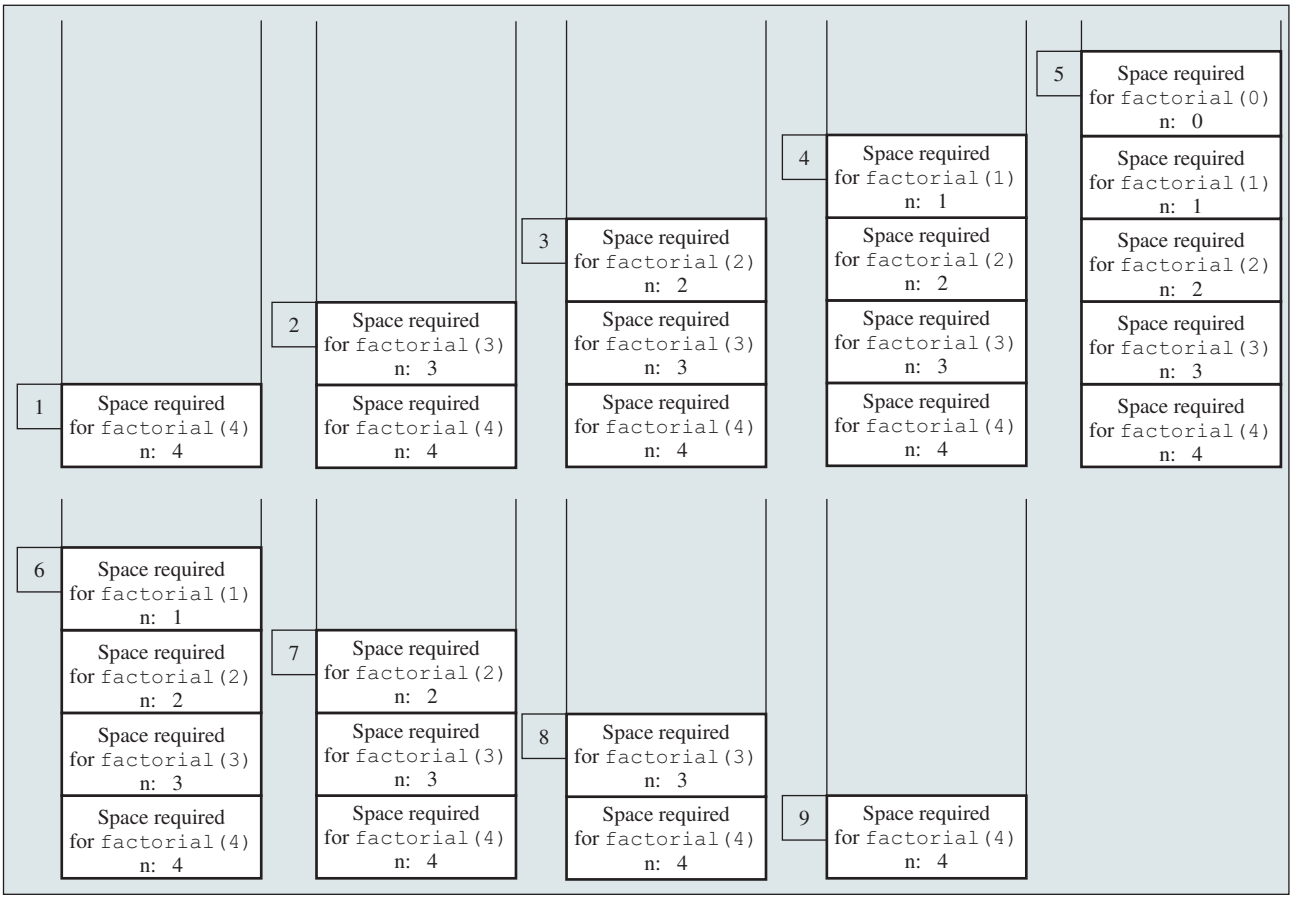


FIGURE 18.3 When `factorial(4)` is being executed, the `factorial` method is called recursively, causing the stack space to dynamically change.

Pedagogical Note

It is simpler and more efficient to implement the `factorial` method using a loop. However, we use the recursive `factorial` method here to demonstrate the concept of recursion. Later in this chapter, we will present some problems that are inherently recursive and are difficult to solve without using recursion.

Note

If recursion does not reduce the problem in a manner that allows it to eventually converge into the base case or a base case is not specified, *infinite recursion* can occur. For example, suppose you mistakenly write the `factorial` method as follows:

infinite recursion

```
public static long factorial(int n) {
  return n * factorial(n - 1);
}
```

The method runs infinitely and causes a `StackOverflowError`.

The example discussed in this section shows a recursive method that invokes itself. This is known as *direct recursion*. It is also possible to create *indirect recursion*. This occurs when method `A` invokes method `B`, which in turn directly or indirectly invokes method `A`.

direct recursion
indirect recursion

18.2.1 What is a recursive method? What is an infinite recursion?

18.2.2 How many times is the `factorial` method in Listing 18.1 invoked for `factorial(6)`?

18.2.3 Show the output of the following programs and identify base cases and recursive calls.

```
public class Test {
  public static void main(String[] args) {
    System.out.println(
      "Sum is " + xMethod(5));
  }

  public static int xMethod(int n) {
    if (n == 1)
      return 1;
    else
      return n + xMethod(n - 1);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    xMethod(1234567);
  }

  public static void xMethod(int n) {
    if (n > 0) {
      System.out.print(n % 10);
      xMethod(n / 10);
    }
  }
}
```

18.2.4 Write a recursive mathematical definition for computing 2^n for a positive integer n.

18.2.5 Write a recursive mathematical definition for computing x^n for a positive integer n and a real number x.

18.2.6 Write a recursive mathematical definition for computing $1 + 2 + 3 + \cdots + n$ for a positive integer n.

18.3 Case Study: Computing Fibonacci Numbers

In some cases, recursion enables you to create an intuitive, straightforward, simple solution to a problem.

The `factorial` method in the preceding section could easily be rewritten without using recursion. In this section, we show an example for creating an intuitive solution to a problem using recursion. Consider the well-known Fibonacci-series problem:

The series:	0	1	1	2	3	5	8	13	21	34	55	89	...
indexes:	0	1	2	3	4	5	6	7	8	9	10	11	

The Fibonacci series begins with **0** and **1**, and each subsequent number is the sum of the preceding two. The series can be recursively defined as

```
fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index - 2) + fib(index - 1); index >= 2
```

The Fibonacci series was named for Leonardo Fibonacci, a medieval mathematician, who originated it to model the growth of the rabbit population. It can be applied in numeric optimization and in various other areas.

How do you find **fib(index)** for a given **index**? It is easy to find **fib(2)** because you know **fib(0)** and **fib(1)**. Assuming you know **fib(index - 2)** and **fib(index - 1)**, you can obtain **fib(index)** immediately. Thus, the problem of computing **fib(index)** is reduced to computing **fib(index - 2)** and **fib(index - 1)**. When doing so, you apply the idea recursively until **index** is reduced to **0** or **1**.

The base case is **index = 0** or **index = 1**. If you call the method with **index = 0** or **index = 1**, it immediately returns the result. If you call the method with **index >= 2**, it divides the problem into two subproblems for computing **fib(index - 1)** and **fib(index - 2)** using recursive calls. The recursive algorithm for computing **fib(index)** can be simply described as follows:

```
if (index == 0)
  return 0;
else if (index == 1)
  return 1;
else
  return fib(index - 1) + fib(index - 2);
```

Listing 18.2 gives a complete program that prompts the user to enter an index and computes the Fibonacci number for that index.

LISTING 18.2 ComputeFibonacci.java

```
1  import java.util.Scanner;
2
3  public class ComputeFibonacci {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8      System.out.print("Enter an index for a Fibonacci number: ");
9      int index = input.nextInt();
10
11     // Find and display the Fibonacci number
12     System.out.println("The Fibonacci number at index "
13       + index + " is " + fib(index));
14   }
15
16   /** The method for finding the Fibonacci number */
17   public static long fib(long index) {
18     if (index == 0) // Base case                          base case
19       return 0;
20     else if (index == 1) // Base case                     base case
21       return 1;
22     else  // Reduction and recursive calls               recursion
23       return fib(index - 1) + fib(index - 2);
24   }
25 }
```

```
Enter an index for a Fibonacci number: 1 Enter
The Fibonacci number at index 1 is 1
```

```
Enter an index for a Fibonacci number: 6 Enter
The Fibonacci number at index 6 is 8
```

```
Enter an index for a Fibonacci number: 7 Enter
The Fibonacci number at index 7 is 13
```

The program does not show the considerable amount of work done behind the scenes by the computer. Figure 18.4, however, shows the successive recursive calls for evaluating `fib(4)`. The original method, `fib(4)`, makes two recursive calls, `fib(3)` and `fib(2)`, and then returns `fib(3) + fib(2)`. However, in what order are these methods called? In Java, operands are evaluated from left to right, so `fib(2)` is called after `fib(3)` is completely evaluated. The labels in Figure 18.4 show the order in which the methods are called.

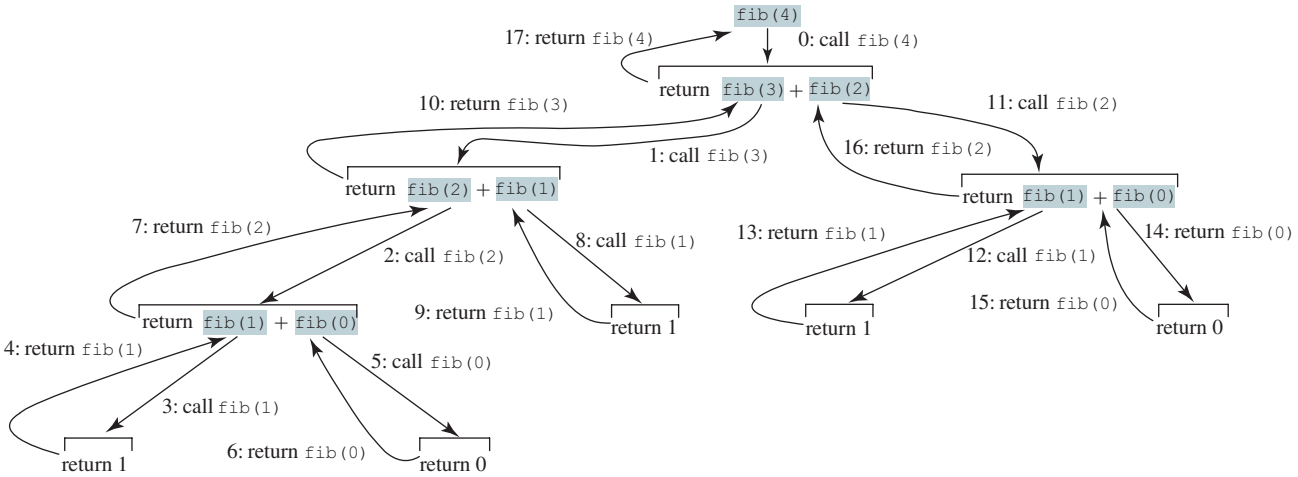


FIGURE 18.4 Invoking `fib(4)` spawns recursive calls to `fib`.

As shown in Figure 18.4, there are many duplicated recursive calls. For instance, `fib(2)` is called twice, `fib(1)` three times, and `fib(0)` twice. In general, computing `fib(index)` requires roughly twice as many recursive calls as does computing `fib(index - 1)`. As you try larger index values, the number of calls substantially increases, as given in Table 18.1.

TABLE 18.1 Number of Recursive Calls in `fib(index)`

`index`	2	3	4	10	20	30	40	50
# of calls	3	5	9	177	21,891	2,692,537	331,160,281	2,075,316,483

Pedagogical Note

The recursive implementation of the `fib` method is very simple and straightforward, but it isn't efficient, because it requires more time and memory to run recursive methods. See Programming Exercise 18.2 for an efficient solution using loops. Though it is not practical, the recursive `fib` method is a good example of how to write recursive methods.

18.3.1 Show the output of the following two programs:

```
public class Test {
  public static void main(String[] args) {
    xMethod(5);
  }

  public static void xMethod(int n) {
    if (n > 0) {
      System.out.print(n + " ");
      xMethod(n - 1);
    }
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    xMethod(5);
  }

  public static void xMethod(int n) {
    if (n > 0) {
      xMethod(n - 1);
      System.out.print(n + " ");
    }
  }
}
```

18.3.2 What is wrong in the following methods?

```
public class Test {
  public static void main(String[] args) {
    xMethod(1234567);
  }

  public static void xMethod(double n) {
    if (n != 0) {
      System.out.print(n);
      xMethod(n / 10);
    }
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Test test = new Test();
    System.out.println(test.toString());
  }

  public Test() {
    Test test = new Test();
  }
}
```

18.3.3 How many times is the `fib` method in Listing 18.2 invoked for `fib(6)`?

18.4 Problem Solving Using Recursion

If you think recursively, you can solve many problems using recursion.

The preceding sections presented two classic recursion examples. All recursive methods have the following characteristics:

recursion characteristics

`if-else`

- The method is implemented using an `if-else` or a `switch` statement that leads to different cases.

base cases

- One or more base cases (the simplest case) are used to stop recursion.

reduction

- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

In general, to solve a problem using recursion, you break it into subproblems. Each subproblem is the same as the original problem, but smaller in size. You can apply the same approach to each subproblem to solve it recursively.

think recursively

Recursion is everywhere. It is fun to *think recursively*. Consider drinking coffee. You may describe the procedure recursively as follows:

```
public static void drinkCoffee(Cup cup) {
  if (!cup.isEmpty()) {
    cup.takeOneSip(); // Take one sip
    drinkCoffee(cup);
  }
}
```

Assume `cup` is an object for a cup of coffee with the instance methods `isEmpty()` and `takeOneSip()`. You can break the problem into two subproblems: one is to drink one sip of coffee, and the other is to drink the rest of the coffee in the cup. The second problem is the same as the original problem, but smaller in size. The base case for the problem is when the cup is empty.

Consider the problem of printing a message `n` times. You can break the problem into two subproblems: one is to print the message one time, and the other is to print it `n - 1` times. The second problem is the same as the original problem, but it is smaller in size. The base case for the problem is `n == 0`. You can solve this problem using recursion as follows:

```
public static void nPrintln(String message, int times) {
  if (times >= 1) {
    System.out.println(message);
    nPrintln(message, times - 1);                    recursive call
  } // The base case is times == 0
}
```

Note the `fib` method in the preceding section returns a value to its caller, but the `drinkCoffee` and `nPrintln` methods are `void` and they do not return a value.

If you *think recursively*, you can use recursion to solve many of the problems presented in earlier chapters of this book. Consider the palindrome problem in Listing 5.14. Recall that a string is a palindrome if it reads the same from the left and from the right. For example, "mom" and "dad" are palindromes but "uncle" and "aunt" are not. The problem of checking whether a string is a palindrome can be divided into two subproblems: think recursively

- Check whether the first character and the last character of the string are equal.
- Ignore the two end characters and check whether the rest of the substring is a palindrome.

The second subproblem is the same as the original problem, but smaller in size. There are two base cases: (1) the two end characters are not the same and (2) the string size is `0` or `1`. In case 1, the string is not a palindrome; in case 2, the string is a palindrome. The recursive method for this problem can be implemented as given in Listing 18.3.

LISTING 18.3 RecursivePalindromeUsingSubstring.java

```
1  public class RecursivePalindromeUsingSubstring {
2    public static boolean isPalindrome(String s) {          method header
3      if (s.length() <= 1) // Base case                     base case
4        return true;
5      else if (s.charAt(0) != s.charAt(s.length() - 1)) // Base case   base case
6        return false;
7      else
8        return isPalindrome(s.substring(1, s.length() - 1));   recursive call
9    }
10
11   public static void main(String[] args) {
12     System.out.println("Is moon a palindrome? "
13       + isPalindrome("moon"));
14     System.out.println("Is noon a palindrome? "
15       + isPalindrome("noon"));
16     System.out.println("Is a a palindrome? " + isPalindrome("a"));
17     System.out.println("Is aba a palindrome? " +
18       isPalindrome("aba"));
19     System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
20   }
21 }
```

```
Is moon a palindrome? false
Is noon a palindrome? true
Is a a palindrome? true
Is aba a palindrome? true
Is ab a palindrome? false
```

The `substring` method in line 8 creates a new string that is same as the original string except without the first and the last characters. Checking whether a string is a palindrome is equivalent to checking whether the substring is a palindrome if the two end characters in the original string are the same.

18.4.1 Describe the characteristics of recursive methods.

18.4.2 For the `isPalindrome` method in Listing 18.3, what are the base cases? How many times is this method called when invoking `isPalindrome("abdxcxdba")`?

18.4.3 Show the call stack for `isPalindrome("abcba")` using the method defined in Listing 18.3.

18.5 Recursive Helper Methods

Sometimes you can find a solution to the original problem by defining a recursive function to a problem similar to the original problem. This new method is called a recursive helper method. The original problem can be solved by invoking the recursive helper method.

The recursive `isPalindrome` method in Listing 18.3 is not efficient because it creates a new string for every recursive call. To avoid creating new strings, you can use the low and high indices to indicate the range of the substring. These two indices must be passed to the recursive method. Since the original method is `isPalindrome(String s)`, you have to create the new method `isPalindrome(String s, int low, int high)` to accept additional information on the string, as given in Listing 18.4.

LISTING 18.4 RecursivePalindrome.java

helper method
base case
base case

```
1  public class RecursivePalindrome {
2    public static boolean isPalindrome(String s) {
3      return isPalindrome(s, 0, s.length() - 1);
4    }
5
6    private static boolean isPalindrome(String s, int low, int high) {
7      if (high <= low) // Base case
8        return true;
9      else if (s.charAt(low) != s.charAt(high)) // Base case
10       return false;
11     else
12       return isPalindrome(s, low + 1, high - 1);
13   }
14
15   public static void main(String[] args) {
16     System.out.println("Is moon a palindrome? "
17       + isPalindrome("moon"));
18     System.out.println("Is noon a palindrome? "
19       + isPalindrome("noon"));
20     System.out.println("Is a a palindrome? " + isPalindrome("a"));
21     System.out.println("Is aba a palindrome? " + isPalindrome("aba"));
22     System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
23   }
24 }
```

Two overloaded `isPalindrome` methods are defined. The first method `isPalindrome(String s)` checks whether a string is a palindrome and the second method `isPalindrome(String s, int low, int high)` checks whether a substring `s(low..high)` is a palindrome. The first method passes the string `s` with `low = 0` and `high = s.length() - 1` to the second method. The second method can be invoked recursively to check a palindrome in an ever-shrinking substring. It is a common design technique in recursive programming to define a second method that receives additional parameters. Such a method is known as a *recursive helper method*.

recursive helper method

Helper methods are very useful in designing recursive solutions for problems involving strings and arrays. The sections that follow give two more examples.

18.5.1 Recursive Selection Sort

Selection sort was introduced in Section 7.11. Recall that it finds the smallest element in the list and swaps it with the first element. It then finds the smallest element remaining and swaps it with the first element in the remaining list and so on until the remaining list contains only a single element. The problem can be divided into two subproblems:

- Find the smallest element in the list and swap it with the first element.
- Ignore the first element and sort the remaining smaller list recursively.

The base case is that the list contains only one element. Listing 18.5 gives the recursive sort method.

LISTING 18.5 RecursiveSelectionSort.java

```
1  public class RecursiveSelectionSort {
2    public static void sort(double[] list) {
3      sort(list, 0, list.length - 1); // Sort the entire list
4    }
5
6    private static void sort(double[] list, int low, int high) {          helper method
7      if (low < high) {                                                   base case
8        // Find the smallest number and its index in list[low .. high]
9        int indexOfMin = low;
10       double min = list[low];
11       for (int i = low + 1; i <= high; i++) {
12         if (list[i] < min) {
13           min = list[i];
14           indexOfMin = i;
15         }
16       }
17
18       // Swap the smallest in list[low .. high] with list[low]
19       list[indexOfMin] = list[low];
20       list[low] = min;
21                                                                       recursive call
22       // Sort the remaining list[low+1 .. high]
23       sort(list, low + 1, high);
24     }
25   }
26 }
```

Two overloaded `sort` methods are defined. The first method `sort(double[] list)` sorts an array in `list[0..list.length - 1]` and the second method `sort(double[] list, int low, int high)` sorts an array in `list[low..high]`. The second method can be invoked recursively to sort an ever-shrinking subarray.

VideoNote
Binary search

18.5.2 Recursive Binary Search

Binary search was introduced in Section 7.10.2. For binary search to work, the elements in the array must be in increasing order. The binary search first compares the key with the element in the middle of the array. Consider the following three cases:

- Case 1: If the key is less than the middle element, recursively search for the key in the first half of the array.
- Case 2: If the key is equal to the middle element, the search ends with a match.
- Case 3: If the key is greater than the middle element, recursively search for the key in the second half of the array.

Case 1 and Case 3 reduce the search to a smaller list. Case 2 is a base case when there is a match. Another base case is that the search is exhausted without a match. Listing 18.6 gives a clear, simple solution for the binary search problem using recursion.

LISTING 18.6 RecursiveBinarySearch.java

```
1  public class RecursiveBinarySearch {
2    public static int binarySearch(int[] list, int key) {
3      int low = 0;
4      int high = list.length - 1;
5      return binarySearch(list, key, low, high);
6    }
7
8    private static int binarySearch(int[] list, int key,
9        int low, int high) {
10     if (low > high) // The list has been exhausted without a match
11       return -low - 1;
12
13     int mid = (low + high) / 2;
14     if (key < list[mid])
15       return binarySearch(list, key, low, mid - 1);
16     else if (key == list[mid])
17       return mid;
18     else
19       return binarySearch(list, key, mid + 1, high);
20   }
21 }
```

helper method
base case
recursive call
base case
recursive call

The first method finds a key in the whole list. The second method finds a key in the list with index from `low` to `high`.

The first `binarySearch` method passes the initial array with `low = 0` and `high = list.length - 1` to the second `binarySearch` method. The second method is invoked recursively to find the key in an ever-shrinking subarray.

Check Point

18.5.1 Show the call stack for `isPalindrome("abcba")` using the method defined in Listing 18.4.

18.5.2 Show the call stack for `selectionSort(new double[]{2, 3, 5, 1})` using the method defined in Listing 18.5.

18.5.3 What is a recursive helper method?

18.6 Case Study: Finding the Directory Size

Recursive methods are efficient for solving problems with recursive structures.

Directory size

The preceding examples can easily be solved without using recursion. This section presents a problem that is difficult to solve without using recursion. The problem is to find the size of a directory. The size of a directory is the sum of the sizes of all files in the directory. A directory d may contain subdirectories. Suppose a directory contains files $f_1, f_2, \ldots, f_m$ and subdirectories $d_1, d_2, \ldots, d_n$, as shown in Figure 18.5.

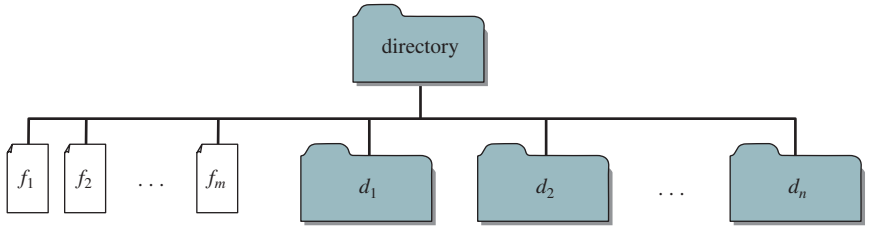


FIGURE 18.5 A directory contains files and subdirectories.

The size of the directory can be defined recursively as follows:

$$size(d) = size(f_1) + size(f_2) + \cdots + size(f_m) + size(d_1) + size(d_2) + \cdots + size(d_n)$$

The `File` class, introduced in Section 12.10, can be used to represent a file or a directory and obtain the properties for files and directories. Two methods in the `File` class are useful for this problem:

- The `length()` method returns the size of a file.
- The `listFiles()` method returns an array of `File` objects under a directory.

Listing 18.7 gives a program that prompts the user to enter a directory or a file and displays its size.

LISTING 18.7 DirectorySize.java

```
1  import java.io.File;
2  import java.util.Scanner;
3
4  public class DirectorySize {
5    public static void main(String[] args) {
6      // Prompt the user to enter a directory or a file
7      System.out.print("Enter a directory or a file: ");
8      Scanner input = new Scanner(System.in);
9      String directory = input.nextLine();
10
11     // Display the size
12     System.out.println(getSize(new File(directory)) + " bytes");      invoke method
13   }
14
15   public static long getSize(File file) {                            getSize method
16     long size = 0; // Store the total size of all files
17
```

is directory?
all subitems
recursive call
base case

```
18      if (file.isDirectory()) {
19        File[] files = file.listFiles(); // All files and subdirectories
20        for (int i = 0; files != null && i < files.length; i++) {
21          size += getSize(files[i]); // Recursive call
22        }
23      }
24      else { // Base case
25        size += file.length();
26      }
27
28      return size;
29    }
30  }
```

```
Enter a directory or a file: c:\book Enter
48619631 bytes
```

```
Enter a directory or a file: c:\book\Welcome.java Enter
172 bytes
```

```
Enter a directory or a file: c:\book\NonExistentFile Enter
0 bytes
```

If the `file` object represents a directory (line 18), each subitem (file or subdirectory) in the directory is recursively invoked to obtain its size (line 21). If the `file` object represents a file (line 24), the file size is obtained and added to the total size (line 25).

What happens if an incorrect or a nonexistent directory is entered? The program will detect that it is not a directory and invoke `file.length()` (line 25), which returns `0`. Thus, in this case, the `getSize` method will return `0`.

testing all cases

Tip

To avoid mistakes, it is a good practice to test all cases. For example, you should test the program for an input of file, an empty directory, a nonexistent directory, and a nonexistent file.

18.6.1 What is the base case for the `getSize` method?

18.6.2 How does the program get all files and directories under a given directory?

18.6.3 How many times will the `getSize` method be invoked for a directory if the directory has three subdirectories and each subdirectory has four files?

18.6.4 Will the program work if the directory is empty (i.e., it does not contain any files)?

18.6.5 Will the program work if line 20 is replaced by the following code?

```
for (int i = 0; i < files.length; i++)
```

18.6.6 Will the program work if lines 20 and 21 are replaced by the following code?

```
for (File file: files)
  size += getSize(file); // Recursive call
```

18.7 Case Study: Tower of Hanoi

The Tower of Hanoi problem is a classic problem that can be solved easily using recursion, but it is difficult to solve otherwise.

The problem involves moving a specified number of disks of distinct sizes from one tower to another while observing the following rules:

- There are *n* disks labeled 1, 2, 3, . . . , *n* and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time and it must be the smallest disk on a tower.

Tower of Hanoi on Companion Website

The objective of the problem is to move all the disks from A to B with the assistance of C. For example, if you have three disks, the steps to move all of the disks from A to B are shown in Figure 18.6. For an interactive demo, see liveexample.pearsoncmg.com/dsanimation/TowerOfHanoieBook.html

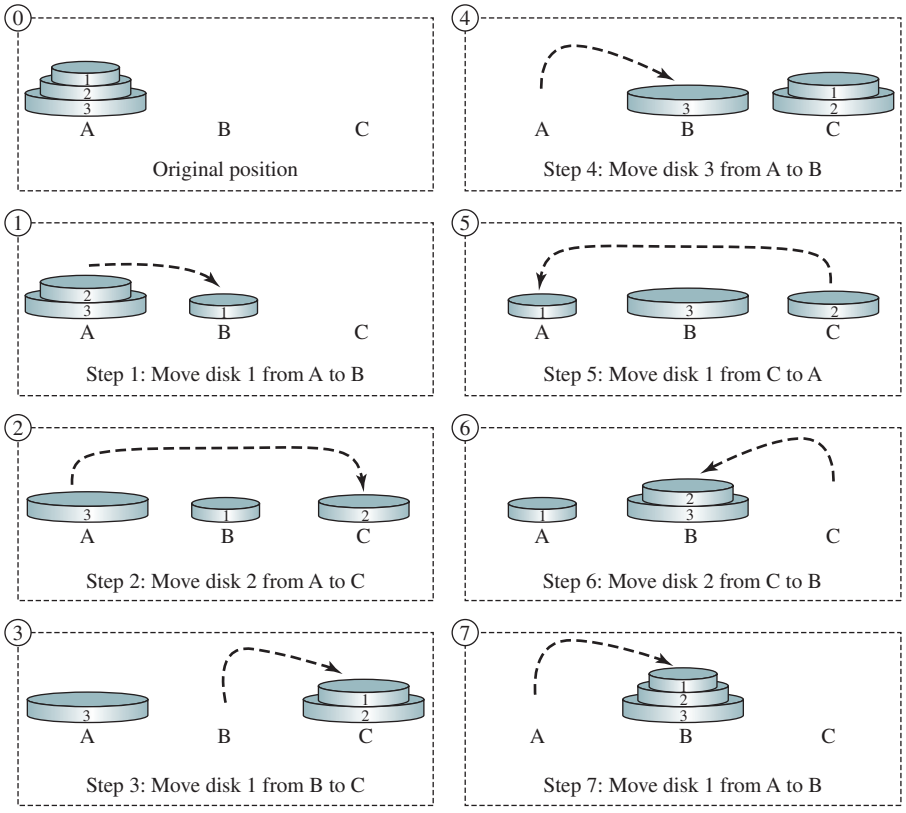


FIGURE 18.6 The goal of the Tower of Hanoi problem is to move disks from tower A to tower B without breaking the rules.

In the case of three disks, you can find the solution manually. For a larger number of disks, however—even for four—the problem is quite complex. Fortunately, the problem has an inherently recursive nature, which leads to a straightforward recursive solution.

The base case for the problem is `n = 1`. If `n == 1`, you could simply move the disk from A to B. When `n > 1`, you could split the original problem into the following three subproblems and solve them sequentially.

1. Move the first `n - 1` disks from A to C recursively with the assistance of tower B, as shown in Step 1 in Figure 18.7.
2. Move disk `n` from A to B, as shown in Step 2 in Figure 18.7.
3. Move `n - 1` disks from C to B recursively with the assistance of tower A, as shown in Step 3 in Figure 18.7.

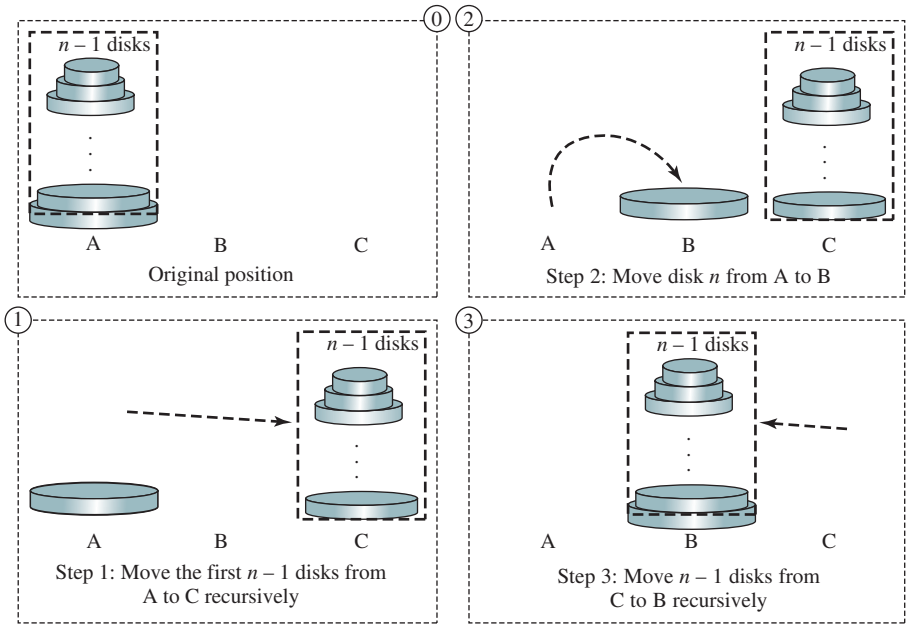


FIGURE 18.7 The Tower of Hanoi problem can be decomposed into three subproblems.

The following method moves *n* disks from the `fromTower` to the `toTower` with the assistance of the `auxTower`:

```
void moveDisks(int n, char fromTower, char toTower, char auxTower)
```

The algorithm for the method can be described as

```
if (n == 1) // Stopping condition
  Move disk 1 from the fromTower to the toTower;
else {
  moveDisks(n - 1, fromTower, auxTower, toTower);
  Move disk n from the fromTower to the toTower;
  moveDisks(n - 1, auxTower, toTower, fromTower);
}
```

Listing 18.8 gives a program that prompts the user to enter the number of disks and invokes the recursive method `moveDisks` to display the solution for moving the disks.

LISTING 18.8 TowerOfHanoi.java

```
1  import java.util.Scanner;
2
3  public class TowerOfHanoi {
4    /** Main method */
```

```
 5    public static void main(String[] args) {
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8      System.out.print("Enter number of disks: ");
 9      int n = input.nextInt();
10
11      // Find the solution recursively
12      System.out.println("The moves are:");
13      moveDisks(n, 'A', 'B', 'C');
14    }
15
16    /** The method for finding the solution to move n disks
17        from fromTower to toTower with auxTower */
18    public static void moveDisks(int n, char fromTower,
19        char toTower, char auxTower) {
20      if (n == 1) // Stopping condition                          base case
21        System.out.println("Move disk " + n + " from " +
22          fromTower + " to " + toTower);
23      else {
24        moveDisks(n - 1, fromTower, auxTower, toTower);         recursion
25        System.out.println("Move disk " + n + " from " +
26          fromTower + " to " + toTower);
27        moveDisks(n - 1, auxTower, toTower, fromTower);         recursion
28      }
29    }
30  }
```

```
Enter number of disks:  4 Enter
The moves are:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Move disk 4 from A to B
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 3 from C to B
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
```

This problem is inherently recursive. Using recursion makes it possible to find a natural, simple solution. It would be difficult to solve the problem without using recursion.

Consider tracing the program for `n = 3`. The successive recursive calls are shown in Figure 18.8. As you can see, writing the program is easier than tracing the recursive calls. The system uses stacks to manage the calls behind the scenes. To some extent, recursion provides a level of abstraction that hides iterations and other details from the user.

18.7.1 How many times is the `moveDisks` method in Listing 18.8 invoked for `moveDisks(5, 'A', 'B', 'C')`?

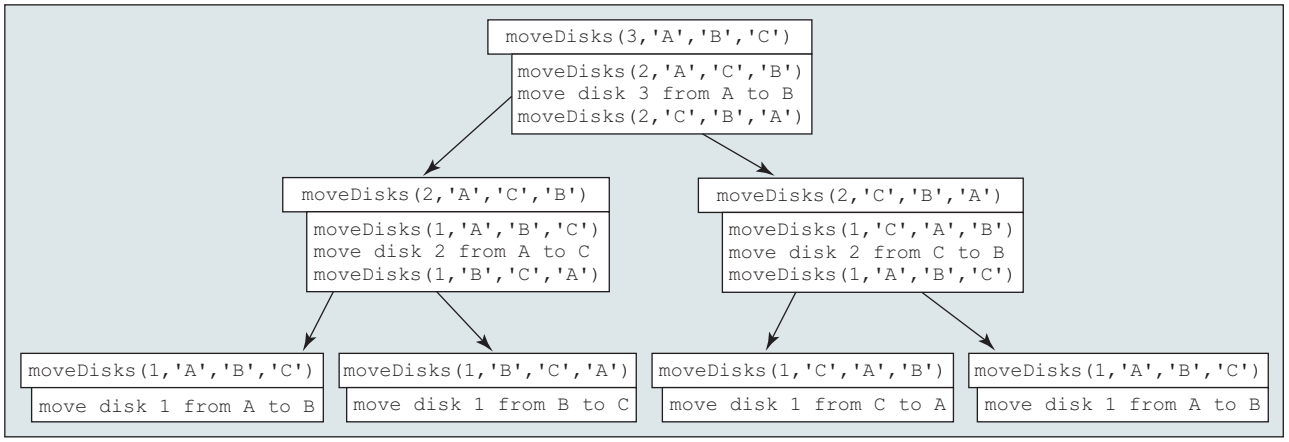


FIGURE 18.8 Invoking `moveDisks(3, 'A', 'B', 'C')` spawns calls to `moveDisks` recursively.

18.8 Case Study: Fractals

Using recursion is ideal for displaying fractals because fractals are inherently recursive.

A *fractal* is a geometrical figure, but unlike triangles, circles, and rectangles, fractals can be divided into parts, each of which is a reduced-size copy of the whole. There are many interesting examples of fractals. This section introduces a simple fractal, the *Sierpinski triangle*, named after a famous Polish mathematician.

A Sierpinski triangle is created as follows:

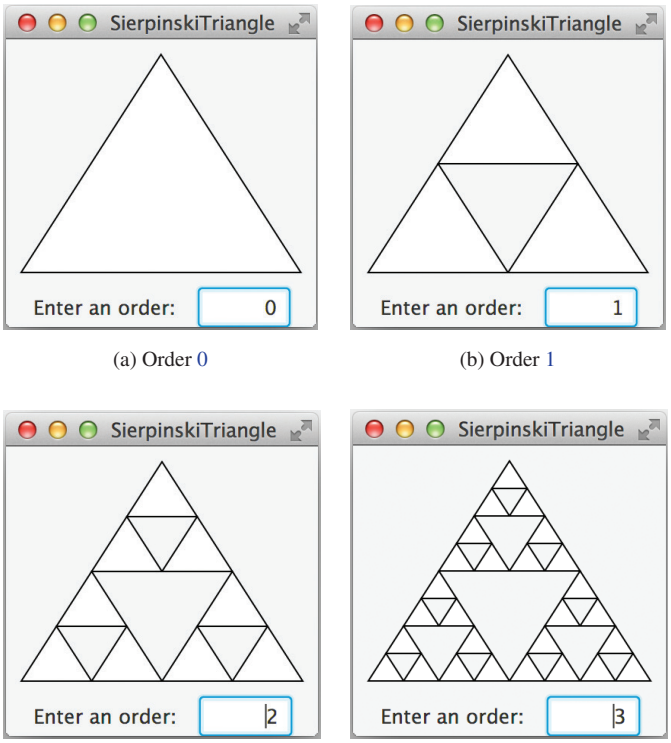
1. Begin with an equilateral triangle, which is considered to be a Sierpinski fractal of order (or level) `0`, as shown in Figure 18.9a.
2. Connect the midpoints of the sides of the triangle of order `0` to create a Sierpinski triangle of order `1` (see Figure 18.9b).
3. Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski triangle of order `2` (see Figure 18.9c).
4. You can repeat the same process recursively to create a Sierpinski triangle of order `3`, `4`, . . . , and so on (see Figure 18.9d). For an interactive demo, see liveexample.pearsoncmg.com/dsanimation/SierpinskiTriangleUsingHTML.html.

The problem is inherently recursive. How do you develop a recursive solution for it? Consider the base case when the order is `0`. It is easy to draw a Sierpinski triangle of order `0`. How do you draw a Sierpinski triangle of order `1`? The problem can be reduced to drawing three Sierpinski triangles of order `0`. How do you draw a Sierpinski triangle of order `2`? The problem can be reduced to drawing three Sierpinski triangles of order `1`, so the problem of drawing a Sierpinski triangle of order ***n*** can be reduced to drawing three Sierpinski triangles of order ***n*** `-` `1`.

Listing 18.9 gives a program that displays a Sierpinski triangle of any order, as shown in Figure 18.9. You can enter an order in a text field to display a Sierpinski triangle of the specified order.

LISTING 18.9 SierpinskiTriangle.java

```
1  import javafx.application.Application;
2  import javafx.geometry.Point2D;
3  import javafx.geometry.Pos;
```



(a) Order 0 (b) Order 1

(c) Order 2 (d) Order 3

FIGURE 18.9 A Sierpinski triangle is a pattern of recursive triangles. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

```
 4  import javafx.scene.Scene;
 5  import javafx.scene.control.Label;
 6  import javafx.scene.control.TextField;
 7  import javafx.scene.layout.BorderPane;
 8  import javafx.scene.layout.HBox;
 9  import javafx.scene.layout.Pane;
10  import javafx.scene.paint.Color;
11  import javafx.scene.shape.Polygon;
12  import javafx.stage.Stage;
13
14  public class SierpinskiTriangle extends Application {
15    @Override // Override the start method in the Application class
16    public void start(Stage primaryStage) {
17      SierpinskiTrianglePane pane = new SierpinskiTrianglePane();          recursive triangle pane
18      TextField tfOrder = new TextField();
19      tfOrder.setOnAction(
20        e -> pane.setOrder(Integer.parseInt(tfOrder.getText())));          listener for text field
21      tfOrder.setPrefColumnCount(4);
22      tfOrder.setAlignment(Pos.BOTTOM_RIGHT);
23
24      // Pane to hold label, text field, and a button
25      HBox hBox = new HBox(10);                                          hold label and text field
26      hBox.getChildren().addAll(new Label("Enter an order: "), tfOrder);
27      hBox.setAlignment(Pos.CENTER);
28
29      BorderPane borderPane = new BorderPane();
30      borderPane.setCenter(pane);
31      borderPane.setBottom(hBox);
32
```

```
33      // Create a scene and place it in the stage
34      Scene scene = new Scene(borderPane, 200, 210);
35      primaryStage.setTitle("SierpinskiTriangle"); // Set the stage title
36      primaryStage.setScene(scene); // Place the scene in the stage
37      primaryStage.show(); // Display the stage
38
39      pane.widthProperty().addListener(ov -> pane.paint());
40      pane.heightProperty().addListener(ov -> pane.paint());
41    }
42
43    /** Pane for displaying triangles */
44    static class SierpinskiTrianglePane extends Pane {
45      private int order = 0;
46
47      /** Set a new order */
48      public void setOrder(int order) {
49        this.order = order;
50        paint();
51      }
52
53      SierpinskiTrianglePane() {
54      }
55
56      protected void paint() {
57        // Select three points in proportion to the pane size
58        Point2D p1 = new Point2D(getWidth() / 2, 10);
59        Point2D p2 = new Point2D(10, getHeight() - 10);
60        Point2D p3 = new Point2D(getWidth() - 10, getHeight() - 10);
61
62        this.getChildren().clear(); // Clear the pane before redisplay
63
64        displayTriangles(order, p1, p2, p3);
65      }
66
67      private void displayTriangles(int order, Point2D p1,
68          Point2D p2, Point2D p3) {
69        if (order == 0) {
70          // Draw a triangle to connect three points
71          Polygon triangle = new Polygon();
72          triangle.getPoints().addAll(p1.getX(), p1.getY(), p2.getX(),
73              p2.getY(), p3.getX(), p3.getY());
74          triangle.setStroke(Color.BLACK);
75          triangle.setFill(Color.WHITE);
76
77          this.getChildren().add(triangle);
78        }
79        else {
80          // Get the midpoint on each edge in the triangle
81          Point2D p12 = p1.midpoint(p2);
82          Point2D p23 = p2.midpoint(p3);
83          Point2D p31 = p3.midpoint(p1);
84
85          // Recursively display three triangles
86          displayTriangles(order - 1, p1, p12, p31);
87          displayTriangles(order - 1, p12, p2, p23);
88          displayTriangles(order - 1, p31, p23, p3);
89        }
90      }
91    }
92  }
```

listener for resizing

three initial points

clear the pane

draw a triangle

create a triangle

top subtriangle
left subtriangle
right subtriangle

The initial triangle has three points set in proportion to the pane size (lines 58–60). If `order == 0`, the `displayTriangles(order, p1, p2, p3)` method displays a triangle that connects the three points `p1`, `p2`, and `p3` (lines 71–77), as shown in Figure 18.10a. Otherwise, it performs the following tasks:

`displayTriangle` method

1. Obtain the midpoint between `p1` and `p2` (line 81), the midpoint between `p2` and `p3` (line 82), and the midpoint between `p3` and `p1` (line 83), as shown in Figure 18.10b.
2. Recursively invoke `displayTriangles` with a reduced order to display three smaller Sierpinski triangles (lines 86–88). Note each small Sierpinski triangle is structurally identical to the original big Sierpinski triangle except that the order of a small triangle is one less, as shown in Figure 18.10b.

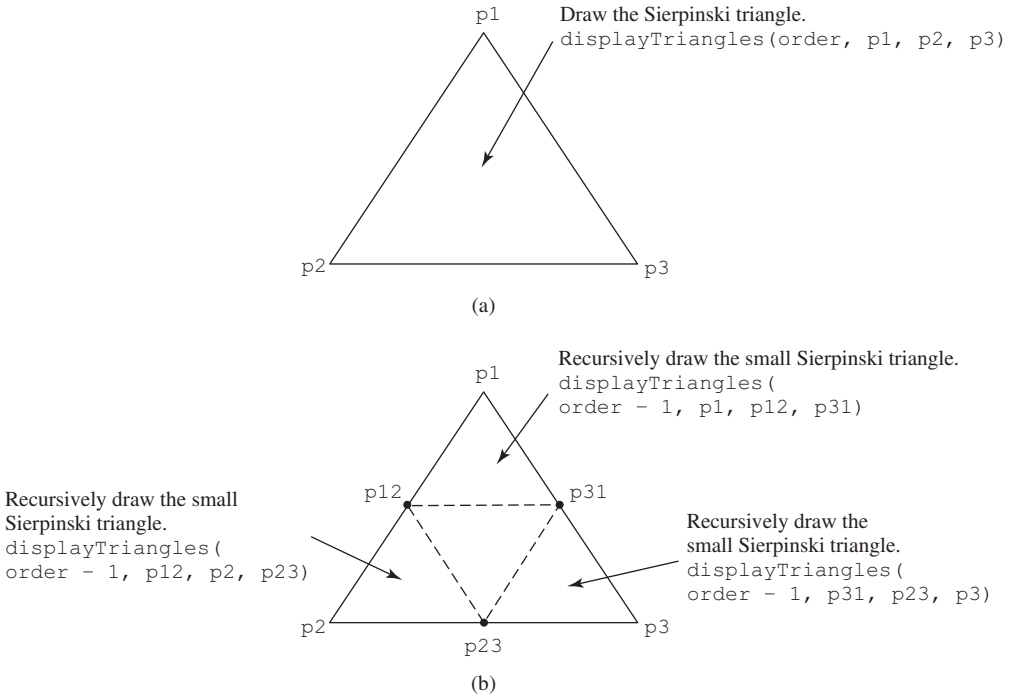


FIGURE 18.10 Drawing a Sierpinski triangle spawns calls to draw three small Sierpinski triangles recursively.

A Sierpinski triangle is displayed in a `SierpinskiTrianglePane`. The `order` property in the inner class `SierpinskiTrianglePane` specifies the order for the Sierpinski triangle. The `Point2D` class, introduced in Section 9.6.3, The `Point2D` Class, represents a point with *x*- and *y*-coordinates. Invoking `p1.midpoint(p2)` returns a new `Point2D` object that is the midpoint between `p1` and `p2` (lines 81–83).

18.8.1 How do you obtain the midpoint between two points?

18.8.2 What is the base case for the `displayTriangles` method?

18.8.3 How many times is the `displayTriangles` method invoked for a Sierpinski triangle of order 0, order 1, order 2, and order *n*?

18.8.4 What happens if you enter a negative order? How do you fix this problem in the code?

18.8.5 Instead of drawing a triangle using a polygon, rewrite the code to draw a triangle by drawing three lines to connect the points in lines 71–77.

18.9 Recursion vs. Iteration

Recursion is an alternative form of program control. It is essentially repetition without a loop.

When you use loops, you specify a loop body. The repetition of the loop body is controlled by the loop control structure. In recursion, the method itself is called repeatedly. A selection statement must be used to control whether to call the method recursively or not.

recursion overhead

Recursion bears substantial overhead. Each time the program calls a method, the system must allocate memory for all of the method's local variables and parameters. This can consume considerable memory and requires extra time to manage the memory.

recursion advantages

Any problem that can be solved recursively can be solved nonrecursively with iterations. Recursion has some negative aspects: it uses up too much time and too much memory. Why, then, should you use it? In some cases, using recursion enables you to specify a clear, simple solution for an inherently recursive problem that would otherwise be difficult to obtain. Examples are the directory-size problem, the Tower of Hanoi problem, and the fractal problem, which are rather difficult to solve without using recursion.

recursion or iteration?

The decision whether to use recursion or iteration should be based on the nature of, and your understanding of, the problem you are trying to solve. The rule of thumb is to use whichever approach can best develop an intuitive solution that naturally mirrors the problem. If an iterative solution is obvious, use it. It will generally be more efficient than the recursive option.

`StackOverflowError`

Note

Recursive programs can run out of memory, causing a `StackOverflowError`.

performance concern

Tip

If you are concerned about your program's performance, avoid using recursion because it takes more time and consumes more memory than iteration. In general, recursion can be used to solve the inherent recursive problems such as Tower of Hanoi, recursive directories, and Sierpinski triangles.

18.9.1 Which of the following statements are true?

a. Any recursive method can be converted into a nonrecursive method.

b. Recursive methods take more time and memory to execute than nonrecursive methods.

c. Recursive methods are *always* simpler than nonrecursive methods.

d. There is always a selection statement in a recursive method to check whether a base case is reached.

18.9.2 What is a cause for a stack-overflow exception?

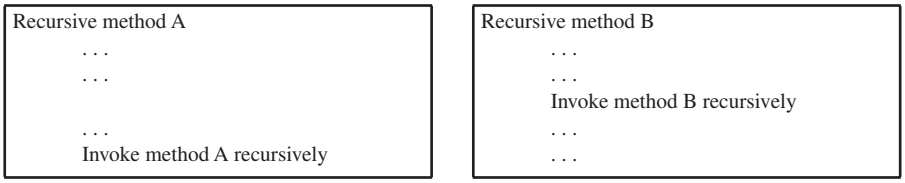
18.10 Tail Recursion

A tail-recursive method is efficient.

tail recursion

A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call, as illustrated in Figure 18.11a. However, method `B` in Figure 18.11b is not tail recursive because there are pending operations after a method call is returned.

For example, the recursive `isPalindrome` method (lines 6–13) in Listing 18.4 is tail recursive because there are no pending operations after recursively invoking `isPalindrome` in line 12. However, the recursive `factorial` method (lines 16–21) in Listing 18.1 is not tail recursive because there is a pending operation, namely multiplication, to be performed on return from each recursive call.



(a) Tail recursion (b) Nontail recursion

FIGURE 18.11 A tail-recursive method has no pending operations after a recursive call.

Tail recursion is desirable because the method ends when the last recursive call ends, and there is no need to store the intermediate calls in the stack. Compilers can optimize tail recursion to reduce stack size.

A nontail-recursive method can often be converted to a tail-recursive method by using auxiliary parameters. These parameters are used to contain the result. The idea is to incorporate the pending operations into the auxiliary parameters in such a way that the recursive call no longer has a pending operation. You can define a new auxiliary recursive method with the auxiliary parameters. This method may overload the original method with the same name but a different signature. For example, the `factorial` method in Listing 18.1 is written in a tail-recursive way in Listing 18.10.

LISTING 18.10 ComputeFactorialTailRecursion.java

```
1  public class ComputeFactorialTailRecursion {
2    /** Return the factorial for a specified number */
3    public static long factorial(int n) {                    original method
4      return factorial(n, 1); // Call auxiliary method       invoke auxiliary method
5    }
6
7    /** Auxiliary tail-recursive method for factorial */
8    private static long factorial(int n, int result) {       auxiliary method
9      if (n == 0)
10       return result;
11     else
12       return factorial(n - 1, n * result); // Recursive call   recursive call
13   }
14 }
```

The first `factorial` method (line 3) simply invokes the second auxiliary method (line 4). The second method contains an auxiliary parameter `result` that stores the result for the factorial of `n`. This method is invoked recursively in line 12. There is no pending operation after a call is returned. The final result is returned in line 10, which is also the return value from invoking `factorial(n, 1)` in line 4.

18.10.1 Identify tail-recursive methods in this chapter.

18.10.2 Rewrite the `fib` method in Listing 18.2 using tail recursion.

KEY TERMS

Chapter Summary

1. A *recursive method* is one that directly or indirectly invokes itself. For a recursive method to terminate, there must be one or more *base cases*.
2. *Recursion* is an alternative form of program control. It is essentially repetition without a loop control. It can be used to write simple, clear solutions for inherently recursive problems that would otherwise be difficult to solve.
3. Sometimes the original method needs to be modified to receive additional parameters in order to be invoked recursively. A *recursive helper method* can be defined for this purpose.
4. Recursion bears substantial overhead. Each time the program calls a method, the system must allocate memory for all of the method's local variables and parameters. This can consume considerable memory and requires extra time to manage the memory.
5. A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call. Some compilers can optimize tail recursion to reduce stack size.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

Sections 18.2 and 18.3

***18.1** (*Identifying anagrams*) Two words are anagrams of each other if they contain the same letters that are arranged in different orders. Write a recursive method that can identify if two given words are anagrams of each other.

***18.2** (*Fibonacci numbers*) Rewrite the `fib` method in Listing 18.2 using iterations.

Hint: To compute `fib(n)` without recursion, you need to obtain `fib(n - 2)` and `fib(n - 1)` first. Let `f0` and `f1` denote the two previous Fibonacci numbers. The current Fibonacci number would then be `f0 + f1`. The algorithm can be described as follows:

```
f0 = 0; // For fib(0)
f1 = 1; // For fib(1)

for (int i = 1; i <= n; i++) {
  currentFib = f0 + f1;
  f0 = f1;
  f1 = currentFib;
}
// After the loop, currentFib is fib(n)
```

Write a test program that prompts the user to enter an index and displays its Fibonacci number.

***18.3** Write a recursive method `public static void dec2b(double x,int b, int n)` that displays x, where $0 \leq x < 1$ in base b with at most n digits after the decimal point. For instance, `dec2b(0.625, 2, 10)` should return 0.101, and `dec2b(0.625, 3, 10)` should return 0.1212121212.

18.4 Combinations refer to the combination of n things taken p at a time without repetition. A recursive definition of C(n, p) is C(n, 0) = C(n, n) = 1, otherwise C(n, p) = C(n − 1, p) + C(n − 1, p − 1).

Write a method `public static long C(long n, long p)` that computes and returns C(n, p).

18.5 The previous method is probably not efficient if you wrote it exactly as its recursive definition. It is not because the method is recursive but because it has not been well-written. It is quite easy to verify that there is another recurrent definition of C(n, p), which is C(n, 0) = C(n, n) = 1, otherwise C(n, p) = C(n, p − 1) × (n − p + 1) /p. Keeping this in mind, write the previous method.

***18.6** Write a recursive method `public static void randomFillSortedArray (int[] x, int l, int r, int a, int b)` that fills the array x between l and r with random values between a and b such that x is sorted. Here is a sample run:

```
Enter the array size: 10 Enter
Enter the limits: 0 1000 Enter
[235, 280, 382, 428, 458, 462, 484, 495, 536, 850]
Enter the array size: 10 Enter
Enter the limits: 0 9 Enter
[0, 0, 3, 3, 4, 6, 6, 8, 8, 9]
```

***18.7** (*Fibonacci series*) Modify Listing 18.2, ComputeFibonacci.java, so that the program finds the number of times the `fib` method is called. (*Hint*: Use a static variable and increment it every time the method is called.)

Section 18.4

***18.8** Let f be a function. Let l, r, and e be real numbers. Write a recursive method that computes and returns a value v such that $|f(v)| <= e$, and $l < v < r$. That is, a method that finds an approximation of a zero of f between l and r. We assume that $f(l)f(r) < 0$.

***18.9** (*Print the characters in a string reversely*) Write a recursive method that displays a string reversely on the console using the following header:

```
public static void reverseDisplay(String value)
```

For example, `reverseDisplay("abcd")` displays `dcba`. Write a test program that prompts the user to enter a string and displays its reversal.

***18.10** (*Occurrences of a specified character in a string*) Write a recursive method that finds the number of occurrences of a specified letter in a string using the following method header:

```
public static int count(String str, char a)
```

For example, `count("Welcome", 'e')` returns `2`. Write a test program that prompts the user to enter a string and a character, and displays the number of occurrences for the character in the string.

***18.11** Write a recursive method that displays all permutations of a given array of integers. Here is a sample run:

```
Enter the array size : 3 Enter
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

Section 18.5

****18.12** (*Print the characters in a string reversely*) Rewrite Programming Exercise 18.9 using a helper method to pass the substring `high` index to the method. The helper method header is

```
public static void reverseDisplay(String value, int high)
```

***18.13** (*Find the largest number in an array*) Write a recursive method that returns the largest integer in an array. Write a test program that prompts the user to enter a list of eight integers and displays the largest element.

***18.14** Assume that you are not able to perform other operations on an int value than to decrement it and compare it with zero and one. Write two methods odd(int x) and even(int x) that respectively return if x is odd or even. We assume that $x >= 0$.

***18.15** (*Occurrences of a specified character in a string*) Rewrite Programming Exercise 18.10 using a helper method to pass the substring `high` index to the method. The helper method header is

```
public static int count(String str, char a, int high)
```

***18.16** Write a recursive method `public static int pos(int[] a, int l, int r)` that positions a[l] at its rank if the array a were sorted between l and r, and that returns this rank. That is, we assume that a is an unsorted array of int, l and r are int such that 0 <= l <= r < a.length, after this call : k = pos(a, l, r); a contains the same elements before and after this call, but it is such that a[k] >= a[i] for all i in [l, k−1] and a[k] 6 a[j] for all j in [k+1, r].

For instance, consider this fragment

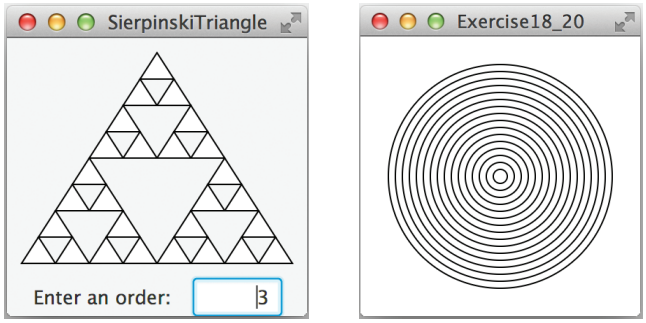
```
int[] a = {8,7,4,1,9,6,2,5,3,0};
System.out.println(Arrays.toString(a));
// [8, 7, 4, 1, 9, 6, 2, 5, 3, 0]
pos(a, 2, 7); // a[2..7] = {4, 1, 9, 6, 2, 5}
System.out.println(Arrays.toString(a));
/* [8, 7, 1, 2, 4, 6, 5, 9, 3, 0] is possible because
4 >= 2, 4 >= 1, 4 < 6, 4 < 5, 4 < 9.
But It could also be [8, 7, 2, 1, 4, 6, 5, 9, 3, 0], or
[8, 7, 2, 1, 4, 9, 5, 6, 3, 0] */
```

***18.17** Write a recursive method that finds the kth-smallest value of a given array. Use the method `pos` of Programming Exercise 18.16.

Sections 18.6–18.10

***18.18** (*Tower of Hanoi*) Modify Listing 18.8, TowerOfHanoi.java, so the program finds the number of moves needed to move n disks from tower A to tower B. (*Hint*: Use a static variable and increment it every time the method is called.)

***18.19** (*Sierpinski triangle*) Revise Listing 18.9 to develop a program that lets the user use the +/− buttons, primary/secondary mouse buttons, and UP/DOWN arrow keys to increase or decrease the current order by 1, as shown in Figure 18.12a. The initial order is 0. If the current order is 0, the *Decrease* button is ignored.



(a) (b)

FIGURE 18.12 (a) Programming Exercise 18.19 uses the + or − buttons to increase or decrease the current order by 1. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) Programming Exercise 18.20 draws ovals using a recursive method.

***18.20** (*Display circles*) Write a Java program that displays ovals, as shown in Figure 18.12b. The circles are centered in the pane. The gap between two adjacent circles is 10 pixels, and the gap between the border of the pane and the largest circle is also 10.

***18.21** (*Decimal to binary*) Write a recursive method that converts a decimal number into a binary number as a string. The method header is

```
public static String dec2Bin(int value)
```

Write a test program that prompts the user to enter a decimal number and displays its binary equivalent.

***18.22** (*Decimal to hex*) Write a recursive method that converts a decimal number into a hex number as a string. The method header is

```
public static String dec2Hex(int value)
```

Write a test program that prompts the user to enter a decimal number and displays its hex equivalent.

***18.23** (*Binary to decimal*) Write a recursive method that parses a binary number as a string into a decimal integer. The method header is

```
public static int bin2Dec(String binaryString)
```

Write a test program that prompts the user to enter a binary string and displays its decimal equivalent.

***18.24** Chess is played on a board with 64 squares arranged in a 8×8 grid. The knight, a piece in this game, can move only in an L-shaped pattern, that is, two squares sideways and then one up or two squares up and one sideways. Can a knight access every square of the chessboard from any other square using this pattern? Write a recursive program that is able to show this.

**18.25 (*String permutation*) Write a recursive method to print all the permutations of a string. For example, for the string `abc`, the permutation is

```
abc
acb
bac
bca
cab
cba
```

(*Hint*: Define the following two methods. The second is a helper method.)

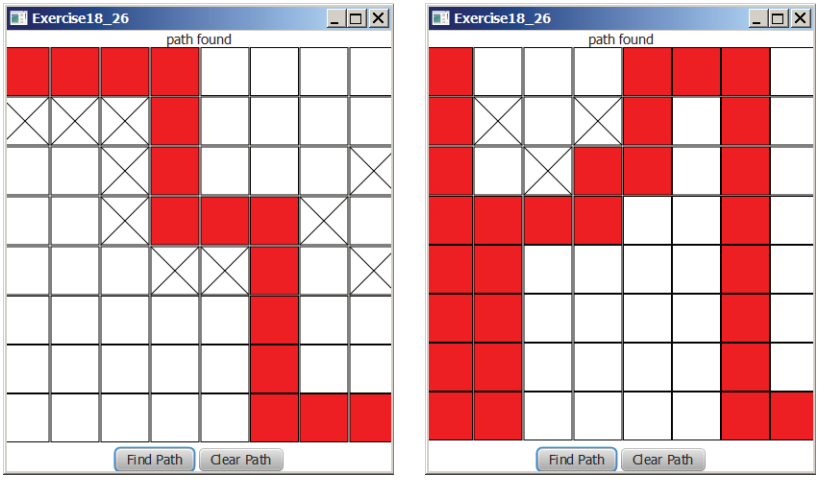
```
public static void displayPermutation(String s)
public static void displayPermutation(String s1, String s2)
```

The first method simply invokes `displayPermutation(" ", s)`. The second method uses a loop to move a character from `s2` to `s1` and recursively invokes it with new `s1` and `s2`. The base case is that `s2` is empty and prints `s1` to the console.

Write a test program that prompts the user to enter a string and displays all its permutations.

**18.26 (*Create a maze*) Write a program that will find a path in a maze, as shown in Figure 18.13a. The maze is represented by a 8 × 8 board. The path must meet the following conditions:

- The path is between the upper-left corner cell and the lower-right corner cell in the maze.



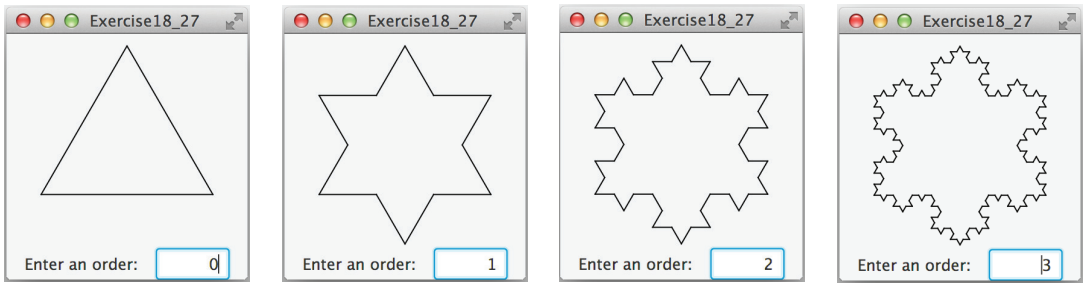
(a) Correct path (b) Illegal path

FIGURE 18.13 The program finds a path from the upper-left corner to the bottom-right corner. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

- The program enables the user to place or remove a mark on a cell. A path consists of adjacent unmarked cells. Two cells are said to be adjacent if they are horizontal or vertical neighbors.
- The path does not contain cells that form a square. The path in Figure 18.13b, for example, does not meet this condition. (The condition makes a path easy to identify on the board.)

**18.27 (*Koch snowflake fractal*) The text presented the Sierpinski triangle fractal. In this exercise, you will write a program to display another fractal, called the *Koch snowflake*, named after a famous Swedish mathematician. A Koch snowflake is created as follows:

1. Begin with an equilateral triangle, which is considered to be the Koch fractal of order (or level) 0, as shown in Figure 18.14a.
2. Divide each line in the shape into three equal line segments and draw an outward equilateral triangle with the middle line segment as the base to create a Koch fractal of order 1, as shown in Figure 18.14b.
3. Repeat Step 2 to create a Koch fractal of order 2, 3, . . . , and so on, as shown in Figures 18.14c and d.



(a) (b) (c) (d)

FIGURE 18.14 A Koch snowflake is a fractal starting with a triangle. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

**18.28 (*Nonrecursive directory size*) Rewrite Listing 18.7, DirectorySize.java, without using recursion.

*18.29 Write a program that prompts the user for a path and displays the directory structure on the command tool as follows.

```
testdir                 An example of a testdir directory
+--f1                   with two directories d1 and d2 and
+--d2                   two files f1 and f2. The directory
    +--d22              d1 has one file f12, and so on.
           +--f221
    +--f21              Hint: To do this, you will have to
    +--d21              use java.io.File
           +--f212
           +--f211
+--f2
+--d1
    +--f12
```

**18.30 (*Find words*) Write a program that finds all occurrences of a word in all the files under a directory, recursively. Pass the parameters from the command line as follows:

```
java Exercise18_30 dirName word
```

VideoNote
Search a string in a directory

**18.31 (*Replace words*) Write a program that replaces all occurrences of a word with a new word in all the files under a directory, recursively. Pass the parameters from the command line as follows:

```
java Exercise18_31 dirName oldWord newWord
```

***18.32 (*Game: Knight's Tour*) The Knight's Tour is an ancient puzzle. The objective is to move a knight, starting from any square on a chessboard, to every other square once, as shown in Figure 18.15a. Note the knight makes only L-shaped moves (two spaces in one direction and one space in a perpendicular direction). As shown in Figure 18.15b, the knight can move to eight squares. Write a program that displays the moves for the knight, as shown in Figure 18.15c. When you click a cell, the knight is placed at the cell. This cell will be the starting point for the knight. Click the *Solve* button to display the path for a solution.

(*Hint*: A brute-force approach for this problem is to move the knight from one square to another available square arbitrarily. Using such an approach, your program will take a long time to finish. A better approach is to employ some heuristics. A knight has two, three, four, six, or eight possible moves, depending on its location. Intuitively, you should attempt to move the knight to the least accessible squares first and leave those more accessible squares open, so there will be a better chance of success at the end of the search.)

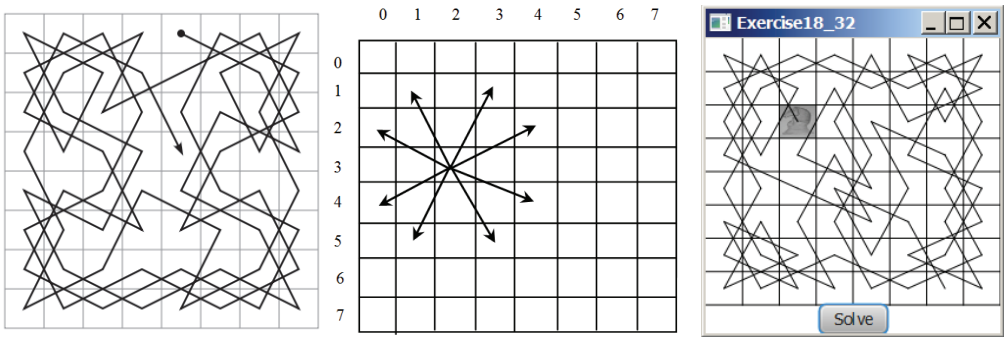


FIGURE 18.15 (a) A knight traverses all squares once. (b) A knight makes an L-shaped move. (c) A program displays a Knight's Tour path. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***18.33 (*Game: Knight's Tour animation*) Write a program for the Knight's Tour problem. Your program should let the user move a knight to any starting square and click the *Solve* button to animate a knight moving along the path, as shown in Figure 18.16.

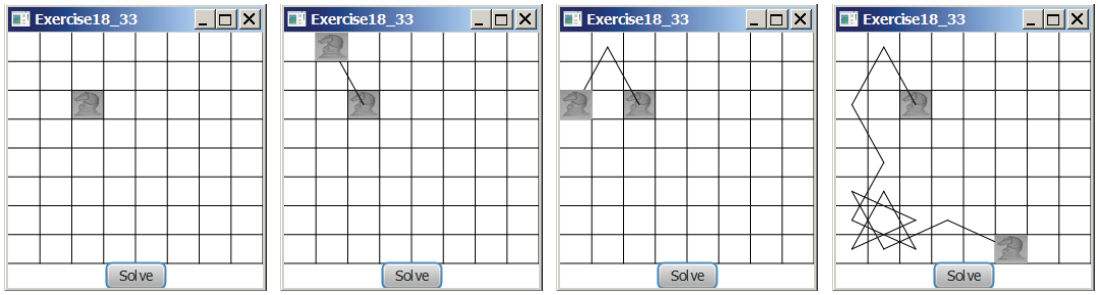


FIGURE 18.16 A knight traverses along the path. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

**18.34 (*Game: Eight Queens*) The Eight Queens problem is to find a solution to place a queen in each row on a chessboard such that no two queens can attack each other. Write a program to solve the Eight Queens problem using recursion and display the result as shown in Figure 18.17.

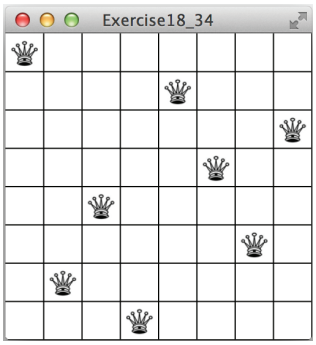


FIGURE 18.17 The program displays a solution to the Eight Queens problem. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

**18.35 (*H-tree fractal*) An H-tree (introduced at the beginning of this chapter in Figure 18.1) is a fractal defined as follows:

1. Begin with a letter H. The three lines of H are of the same length, as shown in Figure 18.1a.
2. The letter H (in its sans-serif form, H) has four endpoints. Draw an H centered at each of the four endpoints to an H-tree of order 1, as shown in Figure 18.1b. These Hs are half the size of the H that contains the four endpoints.
3. Repeat Step 2 to create an H-tree of order 2, 3, . . . , and so on, as shown in Figures 18.1c and d.

Write a program that draws an H-tree, as shown in Figure 18.1.

**18.36 (*Sierpinski triangle*) Write a program that lets the user to enter the order and display the filled Sierpinski triangles as shown in Figure 18.18.

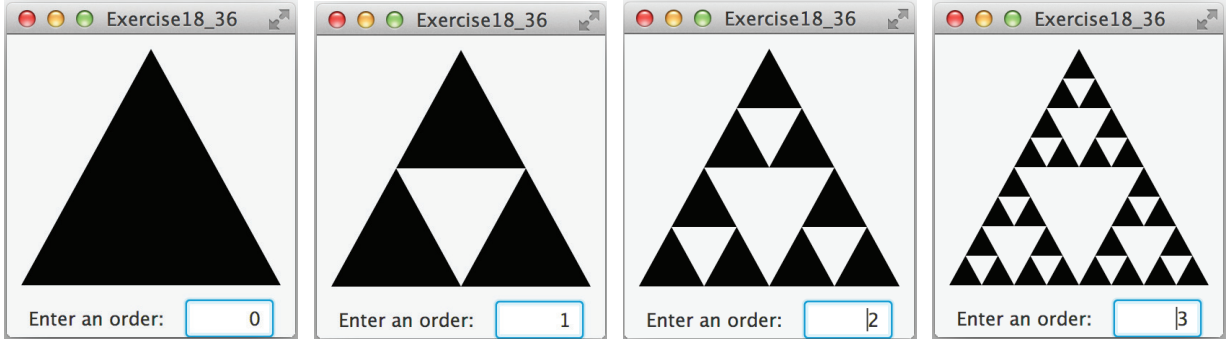
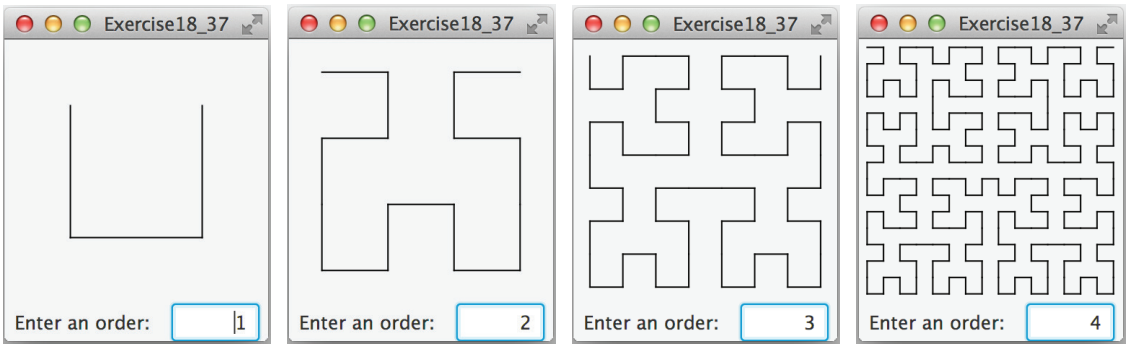


FIGURE 18.18 A filled Sierpinski triangle is displayed. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

**18.37 (*Hilbert curve*) The Hilbert curve, first described by German mathematician David Hilbert in 1891, is a space-filling curve that visits every point in a square grid with a size of 2×2, 4×4, 8×8, 16×16, or any other power of 2. Write a program that displays a Hilbert curve for the specified order, as shown in Figure 18.19.

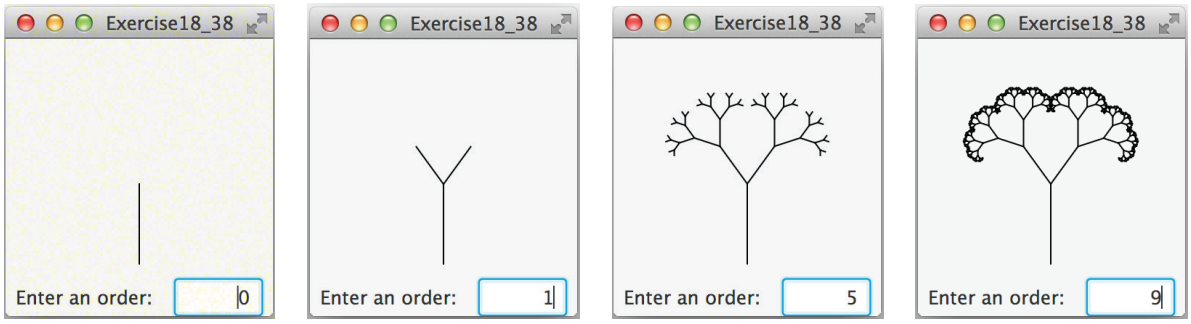


(a) (b) (c) (d)

FIGURE 18.19 A Hilbert curve with the specified order is drawn. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

VideoNote
Recursive tree

**18.38 (*Recursive tree*) Write a program to display a recursive tree as shown in Figure 18.20.



(a) (b) (c) (d)

FIGURE 18.20 A recursive tree with the specified depth is drawn. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

**18.39 (*Drag the tree*) Revise Programming Exercise 18.38 to move the tree to where the mouse is dragged.