18.1 Introduction

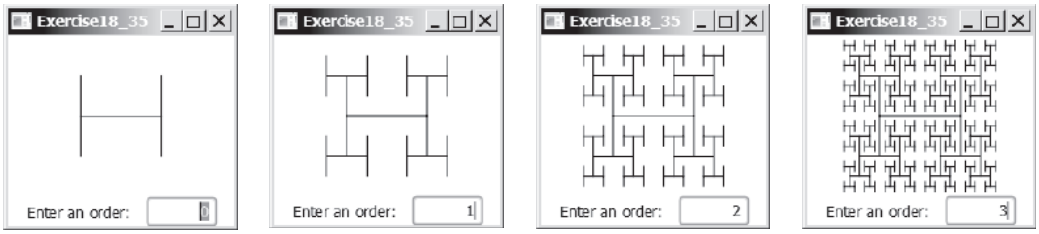
Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.

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? 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. how does it work?

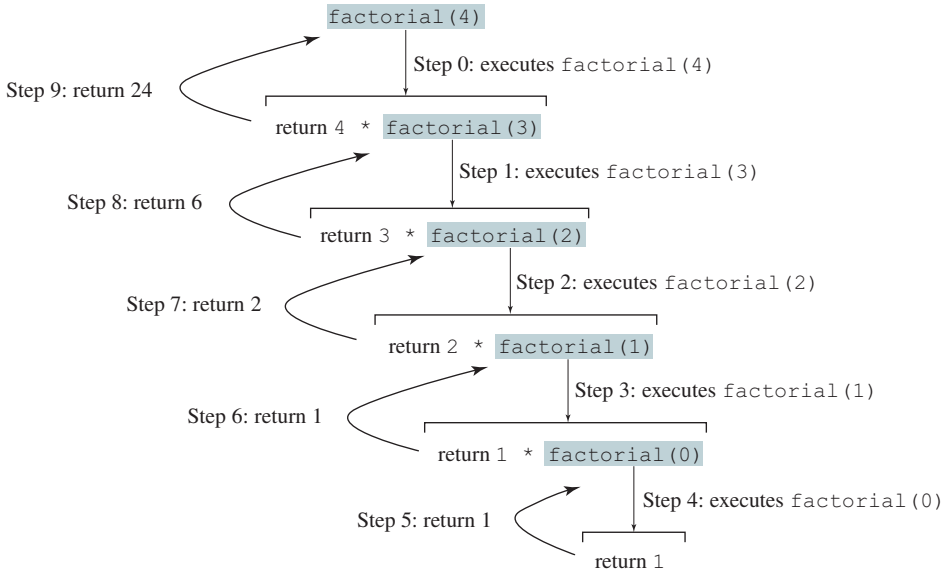


FIGURE 18.2 Invoking `factorial(4)` spawns recursive calls to `factorial`.

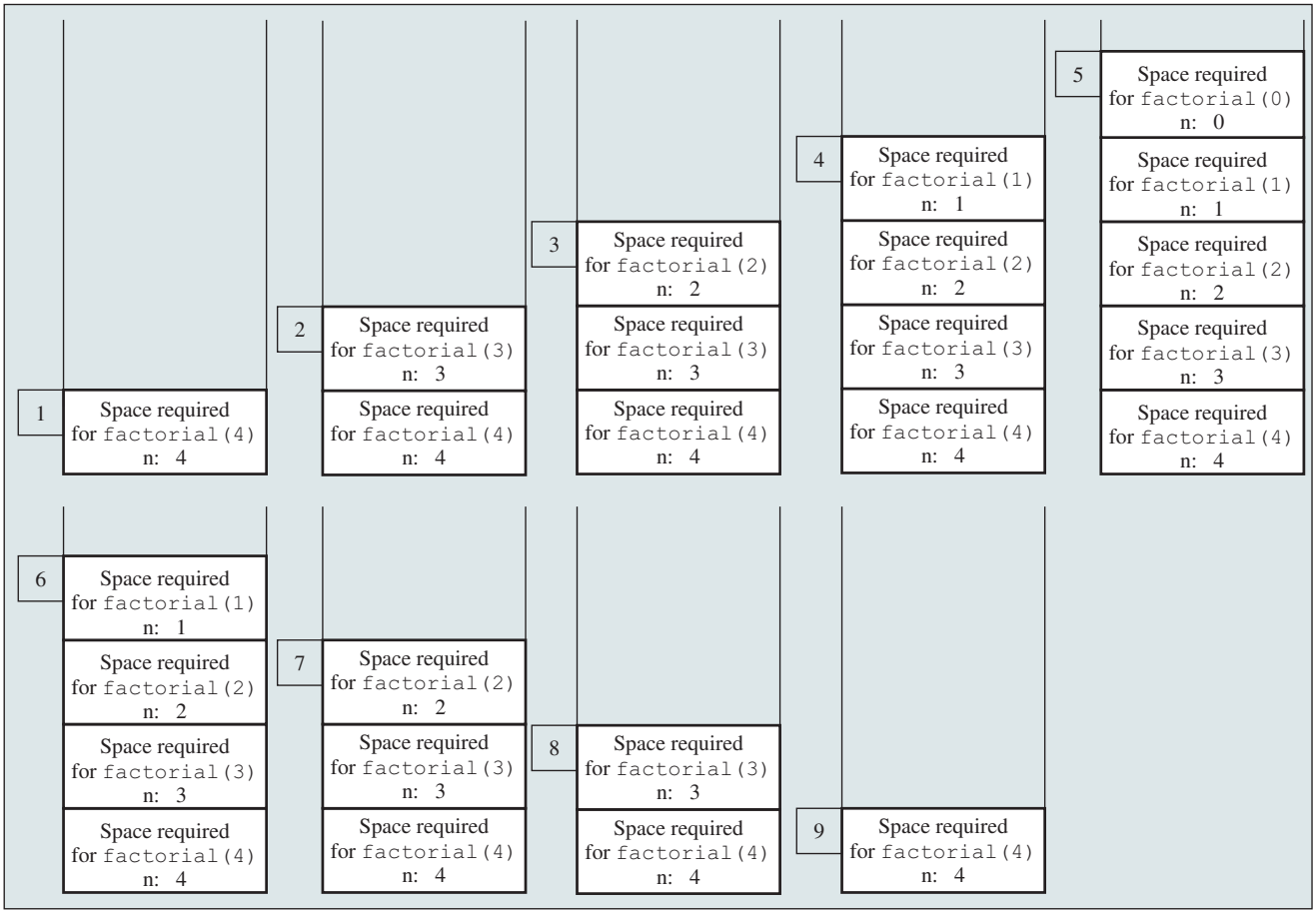


FIGURE 18.3 When `factorial(4)` is being executed, the `factorial` method is called recursively, causing the stack space to dynamically change.

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
19       return 0;
20     else if (index == 1) // Base case
21       return 1;
22     else  // Reduction and recursive calls
23       return fib(index - 1) + fib(index - 2);
24   }
25 }
```

base case

base case
recursion

```
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.

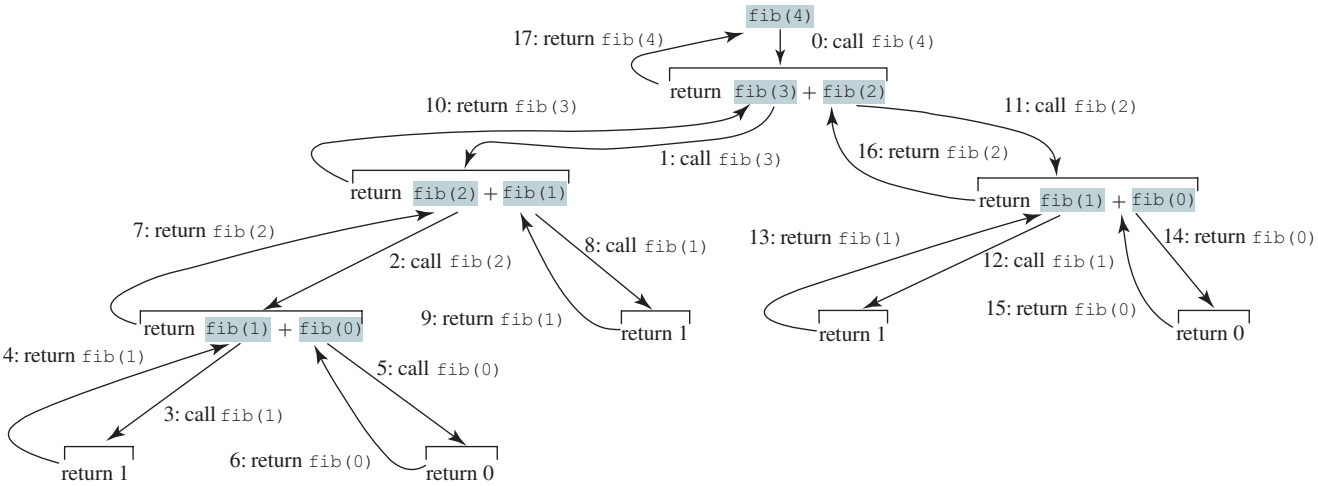


FIGURE 18.4 Invoking `fib(4)` spawns recursive calls to `fib`.

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);
  }
}
```