18.5 Example Using Recursion: Fibonacci Series

The **Fibonacci series**,

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two. This series occurs in nature and describes a form of spiral. The ratio of successive Fibonacci numbers converges on a constant value of 1.618…, a number that has been called the **golden ratio** or the **golden mean.** Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden-mean length-to-width ratio.

The Fibonacci series may be defined recursively as follows:

$$\begin{aligned}&\textbf{fibonacci}(0) = 0\\&\textbf{fibonacci}(1) = 1\\&\textbf{fibonacci}(n) = \textbf{fibonacci}(n-1) + \textbf{fibonacci}(n-2)\end{aligned}$$

There are *two base cases* for the Fibonacci calculation: `fibonacci(0)` is defined to be 0, and `fibonacci(1)` to be 1. Figure 18.5 calculates the *i*th Fibonacci number recursively, using method `fibonacci` (lines 9–18). Method

`main` (lines 20–26) tests `fibonacci`, displaying the Fibonacci values of 0–40. The variable `counter` created in the `for` header (line 22) indicates which Fibonacci number to calculate for each iteration of the loop. Fibonacci numbers tend to become large quickly (though not as quickly as factorials). Therefore, we use type `BigInteger` as the parameter type and the return type of method `fibonacci`.

```
1   // Fig. 18.5: FibonacciCalculator.java
2   // Recursive fibonacci method.
3   import java.math.BigInteger;
4
5   public class FibonacciCalculator {
6      private static BigInteger TWO = BigInteger.va
7
8      // recursive declaration of method fibonacci
9      public static BigInteger fibonacci(BigInteger
10        if (number.equals(BigInteger.ZERO) ||
11           number.equals(BigInteger.ONE)) { // b
12           return number;
13        }
14        else { // recursion step
15           return fibonacci(number.subtract(BigInt
16              fibonacci(number.subtract(TWO)));
17        }
18     }
19
20     public static void main(String[] args) {
21        // displays the fibonacci values from 0-40
22        for (int counter = 0; counter <= 40; count
23           System.out.printf("Fibonacci of %d is:
24              fibonacci(BigInteger.valueOf(counter
25        }
26     }
27  }
```

```
Fibonacci of 0 is: 0
Fibonacci of 1 is: 1
Fibonacci of 2 is: 1
Fibonacci of 3 is: 2
Fibonacci of 4 is: 3
Fibonacci of 5 is: 5
Fibonacci of 6 is: 8
Fibonacci of 7 is: 13
Fibonacci of 8 is: 21
Fibonacci of 9 is: 34
Fibonacci of 10 is: 55

…
Fibonacci of 37 is: 24157817
Fibonacci of 38 is: 39088169
Fibonacci of 39 is: 63245986
Fibonacci of 40 is: 102334155
```

Fig. 18.5

Recursive fibonacci method.

The call to method `fibonacci` (line 24) from `main` is *not* a recursive call, but all subsequent calls to `fibonacci` performed from lines 15–16 *are* recursive, because at that point the calls are initiated by method `fibonacci` itself. Each time `fibonacci` is called, it immediately tests for the *base cases*—`number` equal to 0 or `number` equal to 1 (lines 10–11). We use `BigInteger` constants `ZERO` and `ONE` to represent the values 0 and 1, respectively. If the condition in lines 10–11 is `true`, `fibonacci` simply returns `number`,

because `fibonacci(0)` is 0 and `fibonacci(1)` is 1. Interestingly, if `number` is greater than 1, the recursion step generates *two* recursive calls (lines 15–16), each for a slightly smaller problem than the original call to `fibonacci`. Lines 15–16 use `BigInteger` methods `add` and `subtract` to help implement the recursive step. We also use a constant of type `BigInteger` named `TWO` that we defined at line 6.

Analyzing the Calls to Method `Fibonacci`

Figure 18.6 shows how method `fibonacci` evaluates `fibonacci(3)`. At the bottom of the figure we're left with the values 1, 0 and 1—the results of evaluating the *base cases*. The first two return values (from left to right), 1 and 0, are returned as the values for the calls `fibonacci(1)` and `fibonacci(0)`. The sum 1 plus 0 is returned as the value of `fibonacci(2)`. This is added to the result (1) of the call to `fibonacci(1)`, producing the value 2. This final value is then returned as the value of `fibonacci(3)`.

Figure 18.6 raises some interesting issues about *the order in which Java compilers evaluate the operands of operators*. This order is different from that in which operators are applied to their operands—namely, the order dictated by the rules of operator precedence. From Figure 18.6, it appears that while `fibonacci(3)` is being evaluated, two recursive calls will be made—`fibonacci(2)` and `fibonacci(1)`. But in

what order will they be made? *The Java language specifies that the order of evaluation of the operands is from left to right.* Thus, the call `fibonacci(2)` is made first and the call `fibonacci(1)` second.

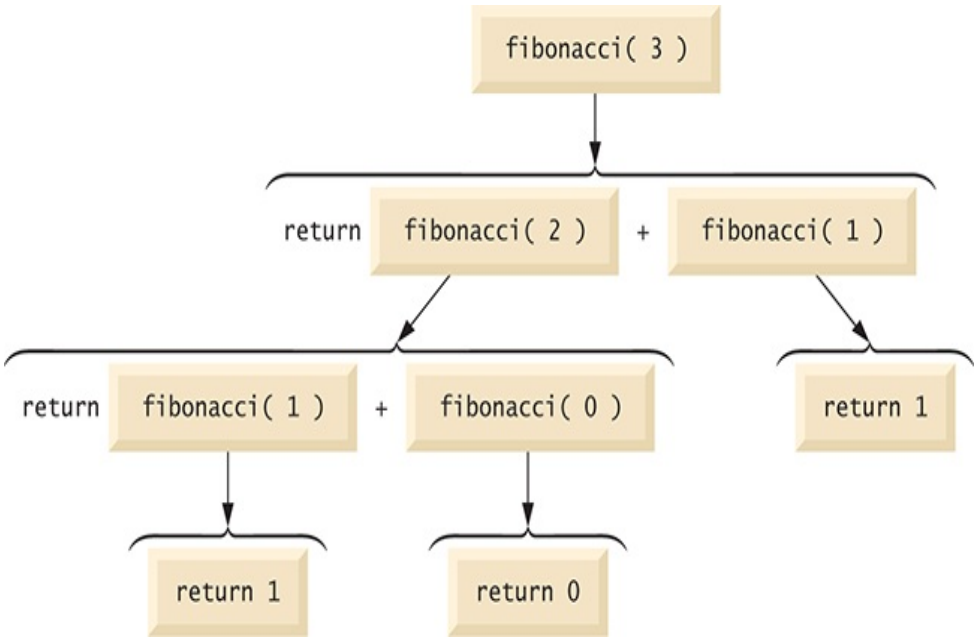

Fig. 18.6

Set of recursive calls for `fibonacci`(3).

Complexity Issues

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each invocation of the `fibonacci` method that does not match one of the *base cases* (0 or 1) results in *two more recursive*

calls to the `fibonacci` method. Hence, this set of recursive calls rapidly gets out of hand. Calculating the Fibonacci value of 20 with the program in Fig. 18.5 requires 21,891 calls to the `fibonacci` method; calculating the Fibonacci value of 30 requires 2,692,537 calls! As you try to calculate larger Fibonacci values, you'll notice that each consecutive Fibonacci number you use the application to calculate results in a substantial increase in calculation time and in the number of calls to the `fibonacci` method. For example, the Fibonacci value of 31 requires 4,356,617 calls, and the Fibonacci value of 32 requires 7,049,155 calls! As you can see, the number of calls to `fibonacci` increases quickly—1,664,080 additional calls between Fibonacci values of 30 and 31 and 2,692,538 additional calls between Fibonacci values of 31 and 32! The difference in the number of calls made between Fibonacci values of 31 and 32 is more than 1.5 times the difference in the number of calls for Fibonacci values between 30 and 31. Problems of this nature can humble even the world's most powerful computers.

[*Note:* In the field of complexity theory, computer scientists study how hard algorithms work to complete their tasks. Complexity issues are discussed in detail in the upper-level computer science curriculum course generally called "Algorithms." We introduce various complexity issues in Chapter 19, Searching, Sorting and Big O.]

In this chapter's exercises, you'll enhance the Fibonacci program of Fig. 18.5 so that it calculates the approximate amount of time required to perform the calculation. For this purpose, you'll call `static System` method

`currentTimeMillis`, which takes no arguments and returns the computer's current time in milliseconds since January 1, 1970.

Performance Tip 18.1

Avoid Fibonacci-style recursive programs, because they result in an exponential "explosion" of method calls.

Calculating Fibonacci Numbers with Lambdas and Streams

8

If you've read Chapter 17, consider doing Exercise 18.29, which asks you to calculate Fibonacci numbers using lambdas and streams, rather than recursion.