18.3 Example Using Recursion: Factorials

Let’s write a recursive program to perform a popular mathematical calculation. Consider the *factorial* of a positive integer *n*, written *n*! (pronounced “*n* factorial”), which is the product

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

with 1! equal to 1 and 0! defined to be 1. For example, 5! is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1,$ which is equal to 120.

The factorial of integer `number` (where `number` ≥ 0) can be calculated *iteratively* (non-recursively) using a `for` statement as follows:

```
factorial = 1;
for (int counter = number; counter >= 1; counter--) {
   factorial *= counter;
}
```

A recursive declaration of the factorial calculation for integers greater than 1 is arrived at by observing the following relationship:

$$n! = n \cdot (n-1)!$$

For example, 5! is clearly equal to $5 \cdot 4!$, as shown by the following equations:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

The evaluation of 5! would proceed as shown in Fig. 18.2. Figure 18.2(a) shows how the succession of recursive calls proceeds until 1! (the base case) is evaluated to be 1, which terminates the recursion. Figure 18.2(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Figure 18.3 uses recursion to calculate and print the factorials of the integers 0 through 21. The recursive method `factorial` (lines 6–13) first tests to determine whether a *terminating condition* (line 7) is `true`. If `number` is less than or equal to `1` (the base case), `factorial` returns `1`, no further recursion is necessary and the method returns. (A precondition of calling method `factorial` in this example is that its argument must be nonnegative.) If `number` is greater than `1`, line 11 expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`, which is a slightly smaller problem than the original calculation, `factorial(number)`.

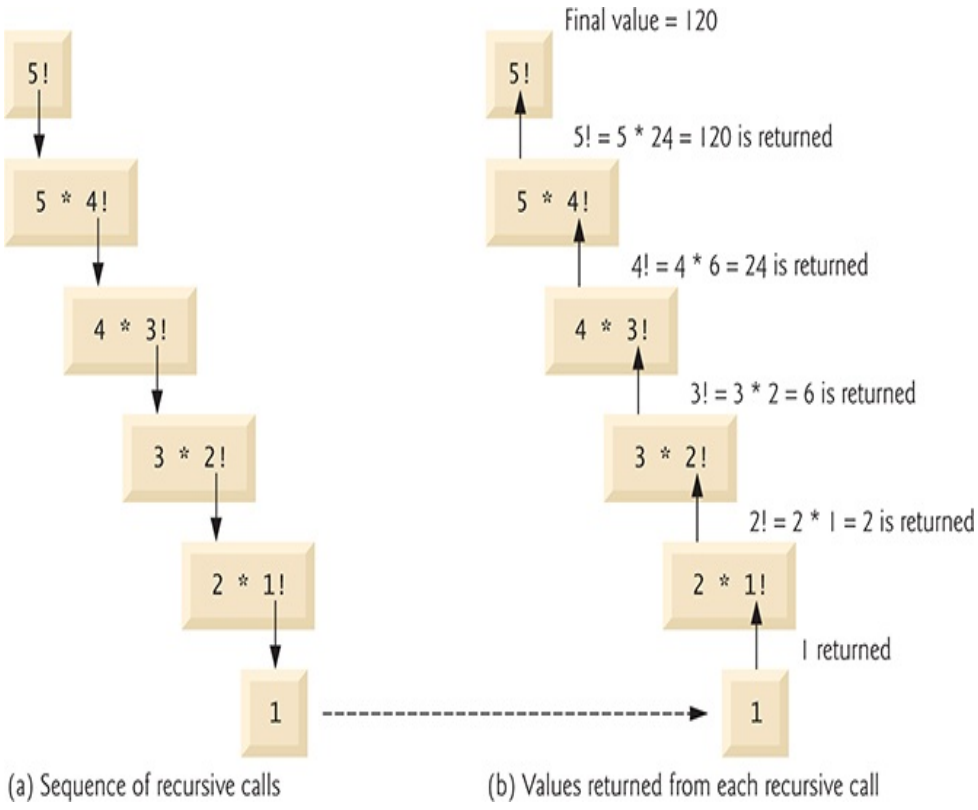

Fig. 18.2

Recursive evaluation of 5!.

Description

Common Programming Error 18.1

Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case can cause a logic error known as ***infinite recursion****, where*

recursive calls are continuously made until memory is exhausted or the method-call stack overflows. This error is analogous to the problem of an infinite loop in an iterative (non-recursive) solution.

```
1   // Fig. 18.3: FactorialCalculator.java
2   // Recursive factorial method.
3
4   public class FactorialCalculator {
5      // recursive method factorial (assumes its pa
6      public static long factorial(long number) {
7         if (number <= 1) { // test for base case
8            return 1; // base cases: 0! = 1 and 1!
9         }
10        else { // recursion step
11           return number * factorial(number - 1);
12        }
13     }
14
15     public static void main(String[] args) {
16        // calculate the factorials of 0 through 2
17        for (int counter = 0; counter <= 21; count
18           System.out.printf("%d! = %d%n", counter
19        }
20     }
21  }
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120

…
12! = 479001600 —— 12! causes overflow for int variab
```

```
...
20! = 2432902008176640000
21! = -4249290049419214848 ── 21! causes overflow for
```

Fig. 18.3

Recursive factorial method.

Method `main` (lines 15–20) displays the factorials of 0–21.[1] The call to the `factorial` method occurs in line 18. Method `factorial` receives a parameter of type `long` and returns a result of type `long`. The program's output shows that factorial values become large quickly. We use type `long` (which can represent relatively large integers) so the program can calculate factorials greater than 12!. Unfortunately, the `factorial` method produces large values so quickly that we exceed the largest `long` value when we attempt to calculate 21!, as you can see in the last line of the program's output.

1. The `for` loops in the `main` methods of this chapter's examples could be implemented with lambdas and streams by using `IntStream` and its `rangeClosed` and `forEach` methods (see Exercise 18.27).

Due to the limitations of integral types, `float` or `double` variables may ultimately be needed to calculate factorials of larger numbers. This points to a weakness in some programming languages—namely, that they aren't easily *extended with new types* to handle unique application requirements. As we saw in Chapter 9, Java is an *extensible*

language that allows us to create arbitrarily large integers if we wish. In fact, package `java.math` provides classes `BigInteger` and `BigDecimal` explicitly for arbitrary precision calculations that cannot be performed with primitive types. You can learn more about these classes at

```
http://docs.oracle.com/javase/8/docs/api/java/math/Bi
http://docs.oracle.com/javase/8/docs/api/java/math/Bi
```

Calculating Factorials with Lambdas and Streams

8

If you've read Chapter 17, consider doing Exercise 18.28, which asks you to calculate factorials using lambdas and streams, rather than recursion.