# 18.4 Reimplementing Class `FactorialCalculator` Using `BigInteger`

8

Figure 18.4 reimplements class `FactorialCalculator` using `BigInteger` variables. To demonstrate larger values than what `long` variables can store, we calculate the factorials of the numbers 0–50. Line 3 imports class `BigInteger` from package `java.math`. The new `factorial` method (lines 7–15) receives a `BigInteger` as an argument and returns a `BigInteger`.

```
1  // Fig. 18.4: FactorialCalculator.java
2  // Recursive factorial method.
3  import java.math.BigInteger;
4
5  public class FactorialCalculator {
6     // recursive method factorial (assumes its pa
7     public static BigInteger factorial(BigInteger
8        if (number.compareTo(BigInteger.ONE) <= 0)
9           return BigInteger.ONE; // base cases: 0
10       }
11       else { // recursion step
12          return number.multiply(
13             factorial(number.subtract(BigInteger
14       }
15    }
```

```
                        16
 17      public static void main(String[] args) {
18         // calculate the factorials of 0 through 5
19         for (int counter = 0; counter <= 50; count
20            System.out.printf("%d! = %d%n", counter
21               factorial(BigInteger.valueOf(counter
                  22         }
                    23      }
                      24   }
```

```
                        0! = 1
                        1! = 1
                        2! = 2
                        3! = 6

                           …
21! = 51090942171709440000 —— 21! and larger values n
           22! = 1124000727777607680000

                           …
47! = 258623241511168180642964355153611979969197632380
48! = 12413915592536072670862289047373375038521486354
49! = 608281864034267560872252163321295376887552831379
50! = 30414093201713378043612608166064768844377641568
```

# Fig. 18.4

Factorial calculations with a recursive method.

Since `BigInteger` is *not* a primitive type, we can't use the arithmetic, relational and equality operators with `BigInteger`s; instead, we must use `BigInteger` methods

to perform these tasks. Line 8 tests for the base case using `BigInteger` method `compareTo`. This method compares the `BigInteger number` that calls the method to the method's `BigInteger` argument. The method returns `-1` if the `BigInteger` that calls the method is less than the argument, `0` if they're equal or `1` if the `BigInteger` that calls the method is greater than the argument. Line 8 compares the `BigInteger number` with the `BigInteger` constant `ONE`, which represents the integer value 1. If `compareTo` returns `-1` or `0`, then `number` is less than or equal to 1 (the base case) and the method returns the constant `BigInteger.ONE`. Otherwise, lines 12–13 perform the recursion step using `BigInteger` methods `multiply` and `subtract` to implement the calculations required to multiply `number` by the factorial of `number - 1`. The program's output shows that `BigInteger` handles the large values produced by the factorial calculation.

# 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.