

18.7 Recursion vs. Iteration

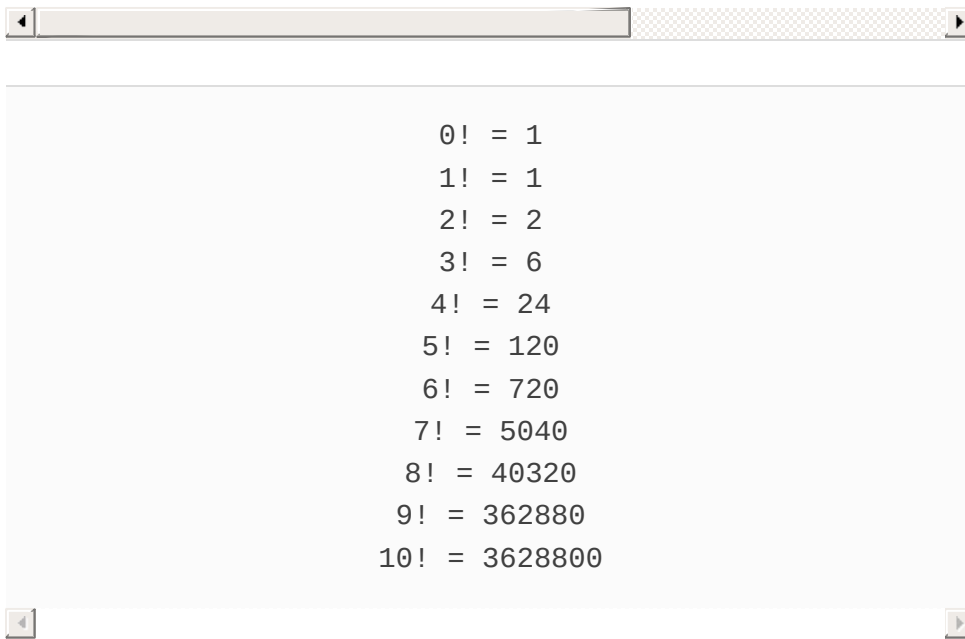
We've studied methods `factorial` and `fibonacci`, which can easily be implemented either recursively or iteratively. In this section, we compare the two approaches and discuss why you might choose one approach over the other in a particular situation.

Both iteration and recursion are *based on a control statement*: Iteration uses an iteration statement (e.g., `for`, `while` or `do...while`), whereas recursion uses a selection statement (e.g., `if`, `if...else` or `switch`):

- Both iteration and recursion involve *iteration*: Iteration explicitly uses an iteration statement, whereas recursion achieves iteration through repeated method calls.
- Iteration and recursion each involve a *termination test*: Iteration terminates when the loop-continuation condition fails, whereas recursion terminates when a base case is reached.
- Iteration with counter-controlled iteration and recursion both *gradually approach termination*: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail, whereas recursion keeps producing smaller versions of the original problem until the base case is reached.
- Both iteration and recursion *can occur infinitely*: An infinite loop occurs with iteration if the loop-continuation test never becomes false, whereas infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case, or if the base case is not tested.

To illustrate the differences between iteration and recursion, let's examine an iterative solution to the factorial problem (Fig. 18.9, lines 10–12). Here we use an iteration statement, rather than a selection statement (Fig. 18.3, lines 7–12). Both solutions use a termination test. In the recursive solution (Fig. 18.3), line 7 tests for the *base case*. In Fig. 18.9's iterative solution, line 10 tests the loop-continuation condition—if the test fails, the loop terminates. Finally, instead of producing smaller versions of the original problem, the iterative solution uses a counter that is modified until the loop-continuation condition becomes **false**.

```
1  // Fig. 18.9: FactorialCalculator.java
2  // Iterative factorial method.
3
4  public class FactorialCalculator {
5  // iterative declaration of method factorial
6  public long factorial(long number) {
7      long result = 1;
8
9      // iteratively calculate factorial
10     for (long i = number; i >= 1; i--) {
11         result *= i;
12     }
13
14     return result;
15 }
16
17 public static void main(String[] args) {
18     // calculate the factorials of 0 through 10
19     for (int counter = 0; counter <= 10; counter++)
20         System.out.printf("%d! = %d\n", counter, factorial(counter));
21 }
22 }
23 }
```



```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Fig. 18.9

Iterative factorial method.

Recursion has many *negatives*. It repeatedly invokes the mechanism, and consequently the *overhead, of method calls*. This iteration can be *expensive* in terms of both processor time and memory space. Each recursive call causes another *copy* of the method (actually, only the method's variables, stored in the stack frame) to be created—this set of copies *can consume considerable memory space*. Since iteration occurs within a method, repeated method calls and extra memory assignment are avoided.



Observation 18.1

Any problem that can be solved recursively can be solved iteratively and vice versa. A recursive approach is preferred over an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug. A recursive approach can often be implemented with fewer lines of code. Another reason to choose a recursive approach is that an iterative one might not be apparent.



Performance Tip 18.2

When performance is crucial, you might want to try various iterative and recursive approaches to see which achieve your goal.



Common Programming Error 18.2

Accidentally having a nonrecursive method call itself either directly or indirectly through another method can cause infinite recursion.