Recursion

*JAVA: An Introduction to Problem Solving & Programming, 6*th Ed. By Walter Savitch
ISBN 0132162709

Objectives

- Describe the concept of recursion
- Use recursion as a programming tool

Basics of Recursion

- A recursive algorithm will have one subtask that is a small version of the entire algorithm's task
- A recursive algorithm contains an invocation of itself
- Must be defined correctly else algorithm could call itself forever or not at all

Simple Example - Countdown

- Given an integer value *num* output all the numbers from *num* down to 1
- Can do this easier and faster with a loop; the recursive version is an example only
- First handle the simplest case; the **base case** or stopping condition

```
public static void countDown(int num)
{
    if (num <= 0)
    {
        System.out.println();
    }
}
```

Recursive Countdown

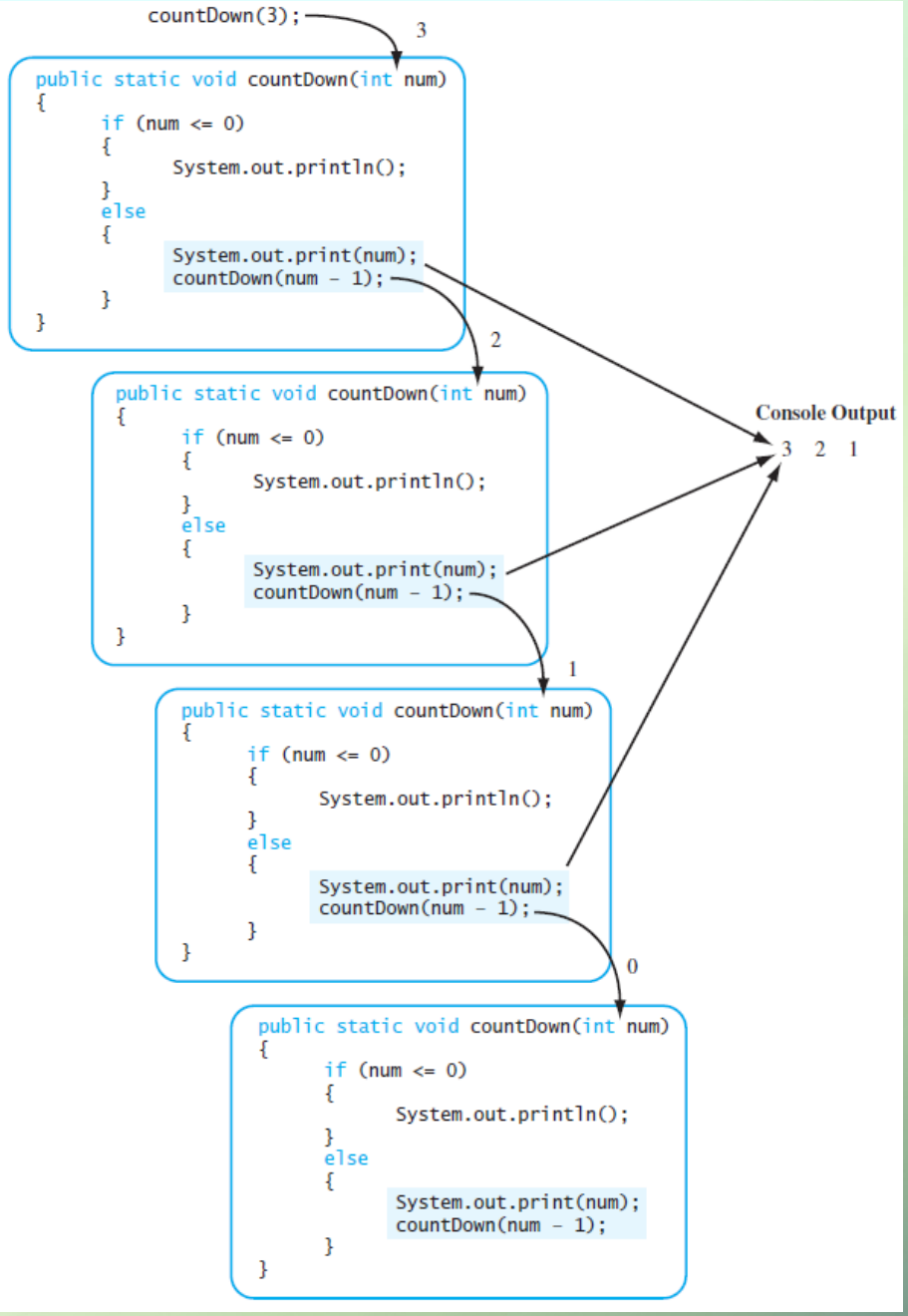
- Next handle larger cases; phrase solution in terms of a smaller version of the same problem
- `countDown(3)` is to output 3 then output the result of `countDown(2)`

View demonstration, listing 11.1
`class RecursionCountdown`

- See code for count down

Sequence of Calls

countDown(3)



Keys to Successful Recursion

- Must have a branching statement that leads to different cases
- One or more of the branches should have a recursive call of the method
  - Recursive call must us "smaller" version of the original argument
- One or more branches must include *no* recursive call
  - This is the base or stopping case

Infinite Recursion

- Nothing stops the method from repeatedly invoking itself
  - Program will eventually crash when computer exhausts its resources (stack overflow)

Recursive Versus Iterative

- Any method including a recursive call can be rewritten
  - To do the same task
  - Done *without* recursion
- Non recursive algorithm uses *iteration*
  - Method which implements is *iterative method*

Recursive Versus Iterative

- Recursive method
  - Uses more storage space than iterative version
  - Due to overhead during runtime
  - Also runs slower
- However in *some* programming tasks, recursion is a better choice, a more elegant solution

Recursion

- For some problems, it's useful to have a method call itself.
- A method that does so is known as a **recursive method**.
- A recursive method can call itself either *directly* or *indirectly through another method*.

Recursion

- Recursion is the process of defining something in terms of itself. As it relates to java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

Tail Recursion

- Tail recursion is defined as occurring when the recursive call is at the end of the recursive instruction. It is useful to notice when ones algorithm uses tail recursion because in such a case, the algorithm can usually be rewritten to use iteration instead. (ex: count Down)

Factorial

- with 1! equal to 1 and 0! defined to be 1.
- For example, 5! is the product 5 · 4 · 3 · 2 · 1, which is equal to 120.
- The factorial of integer number (where number ≥ 0) can be calculated iteratively (nonrecursively) using a for statement as follows:

```
factorial = 1;

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

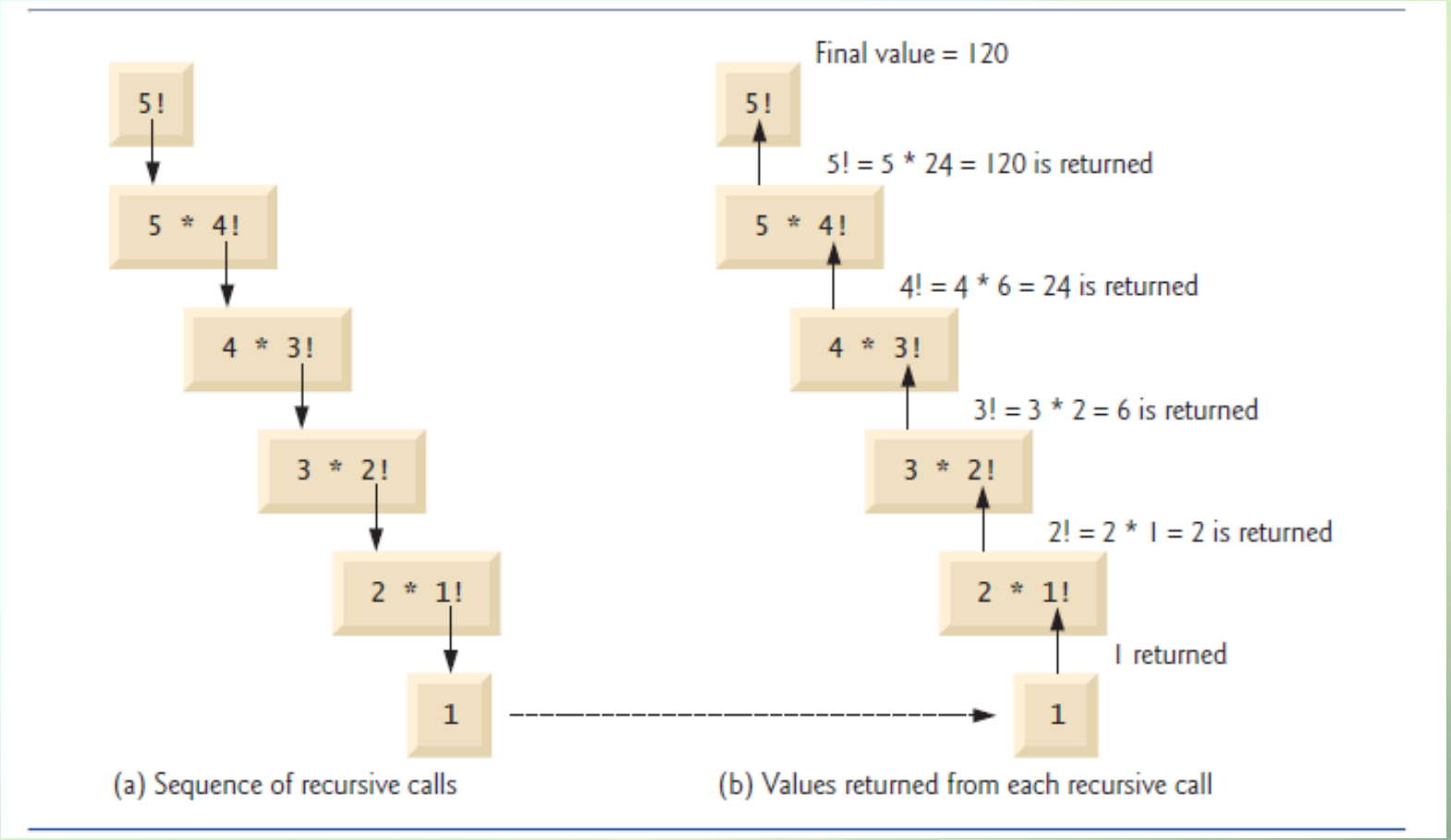
- A recursive declaration of the factorial method is arrived at by observing the following relationship:

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

- For example, 5! is clearly equal to 5 · 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 from 0– 21.



- The recursive method factorial (lines 7–13) first tests to determine whether a terminating condition (line 9) 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.
- If number is greater than 1, line 12 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).

- See code for factorial

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
...
12! = 479001600 —— 12! causes overflow for int variables
...
20! = 2432902008176640000
21! = -4249290049419214848 —— 21! causes overflow for long variables
```

Example Using Recursion: Fibonacci Series

- The Fibonacci series, begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two.
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- The Fibonacci series may be defined recursively as follows:
  - fibonacci(0) = 0
    fibonacci(1) = 1
    fibonacci(*n*) = *fibonacci(n – 1) + fibonacci(n – 2)*

18.4 Example Using Recursion: Fibonacci Series (cont.)

- *Two base cases for*
  - `fibonacci(0)` is defined to be 0
  - `fibonacci(1)` to be 1
- Fibonacci numbers tend to become large quickly.
  - We use type `BigInteger` as the parameter type and the return type of method `fibonacci`.

- See code for Fibonacci

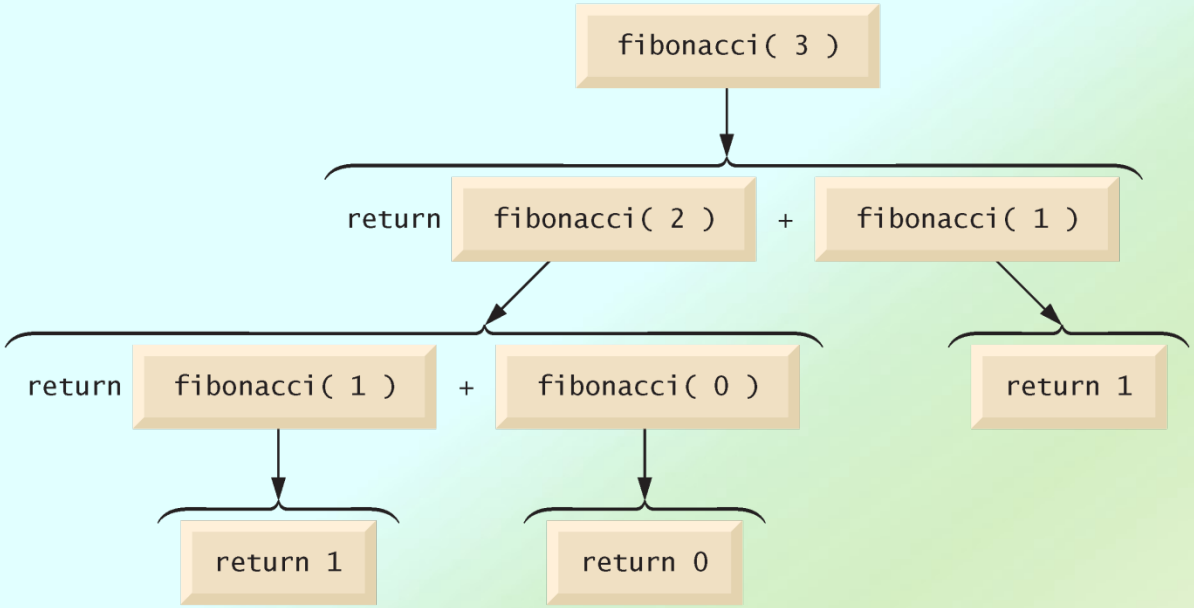


Fig. 18.6 | Set of recursive calls for `fibonacci( 3 )`.

(Recursive power Method)

Write a recursive method power(base, exponent) that, when called, returns $base^{exponent}$

For example, power(3,4) = 3 * 3 * 3 * 3.

Assume that exponent is an integer greater than or equal to 1.

[*Hint:* The recursion step should use the relationship

$base^{exponent} = base \cdot base^{exponent - 1}$

and the terminating condition occurs when exponent is equal to 1, because

$base^{1} = base$

Incorporate this method into a program that enables the user to enter the base and exponent.]

See Code for Pow

Sample output

```
run:
Enter base: 3
Enter exponent: 4
Value is 81
BUILD SUCCESSFUL (total time: 10 seconds)
```

- See code

Summary

- Method with self invocation
  - Invocation considered a recursive call
- Recursive calls
  - Legal in Java
  - Can make some method definitions clearer
- Algorithm with one subtask that is smaller version of entire task
  - Algorithm is a recursive method