Introduction to Recursion

- We have been calling other methods from a method.
- It's also possible for a method to call itself.
- A method that calls itself is a *recursive method*.
- Example: EndlessRecursion.java

Introduction to Recursion

- This method in the example displays the string "This is a recursive method.", and then calls itself.
- Each time it calls itself, the cycle is repeated endlessly.
- Like a loop, a recursive method must have some way to control the number of times it repeats.
- Example: Recursive.java, RecursionDemo.java

Introduction to Recursion

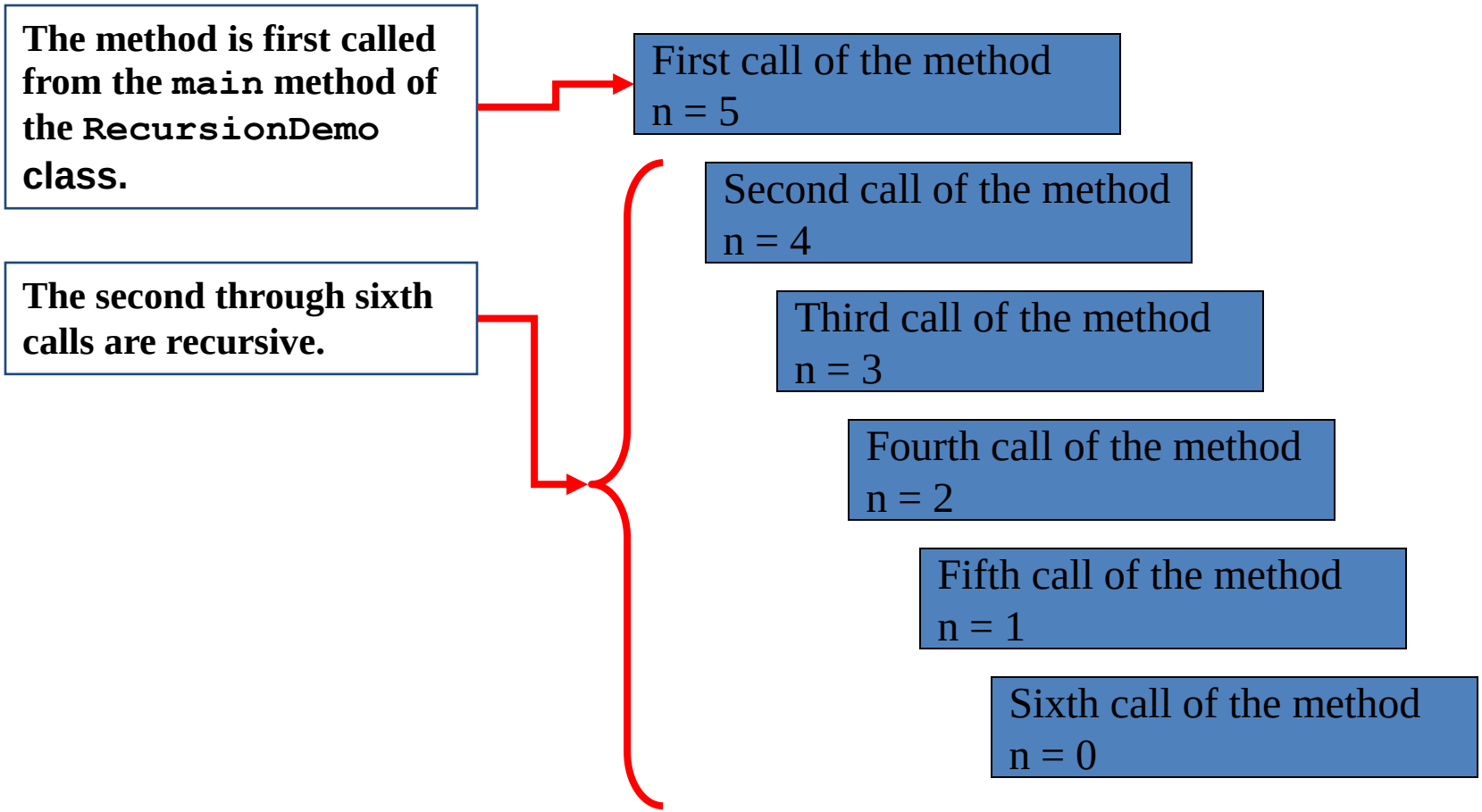


Figure 15-1 First two calls of the method

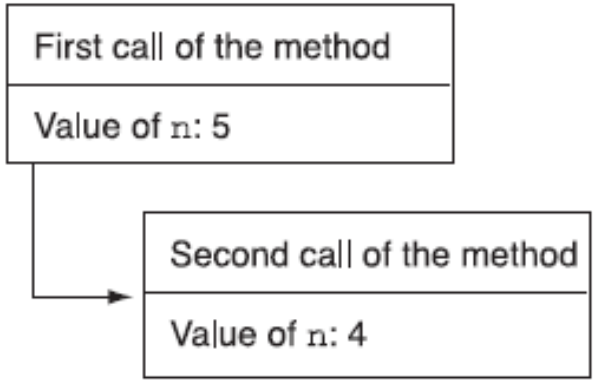
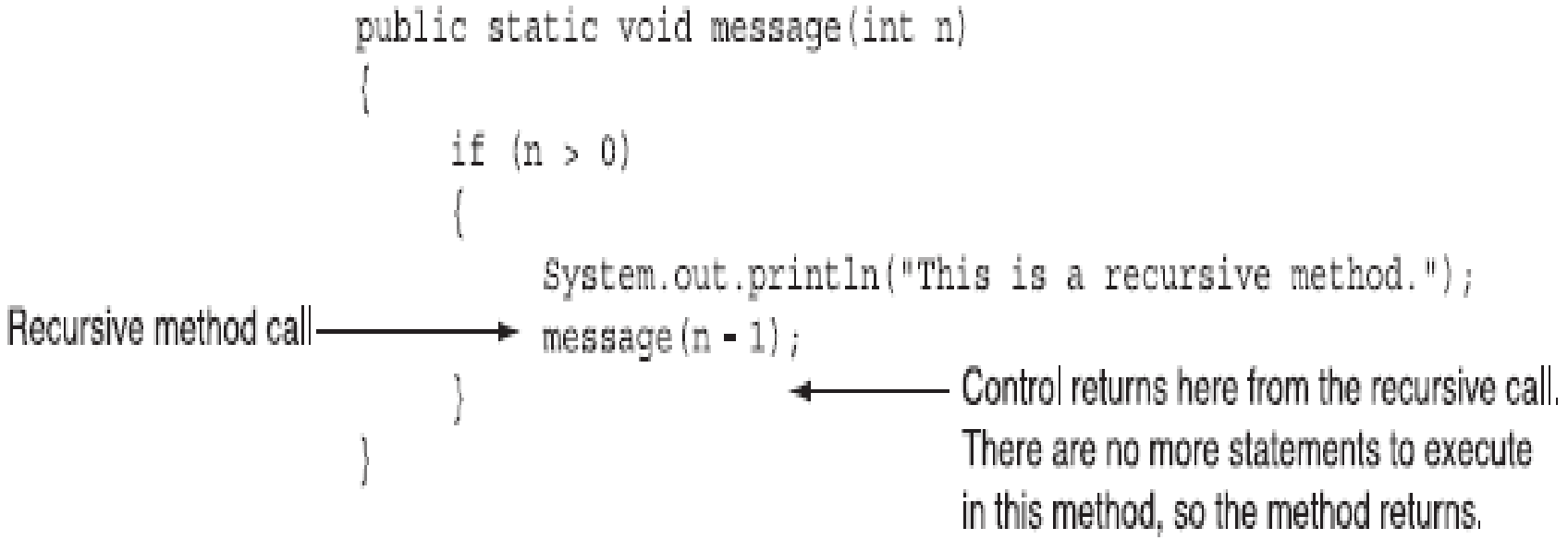


Figure 15-3 Control returns to the point after the recursive method call



Solving Problems With Recursion

- Recursion can be a powerful tool for solving repetitive problems.
- Recursion is never absolutely required to solve a problem.
- Any problem that can be solved recursively can also be solved iteratively, with a loop.
- In many cases, recursive algorithms are less efficient than iterative algorithms.

Solving Problems With Recursion

- Recursive solutions repetitively:
  - allocate memory for parameters and local variables, and
  - store the address of where control returns after the method terminates.
- These actions are called *overhead* and take place with each method call.
- This overhead does not occur with a loop.
- Some repetitive problems are more easily solved with recursion than with iteration.
  - Iterative algorithms might execute faster; however,
  - a recursive algorithm might be designed faster.

Solving Problems With Recursion

- Recursion works like this:
  - A base case is established.
    - If matched, the method solves it and returns.
  - If the base case cannot be solved now:
    - the method reduces it to a smaller problem (recursive case) and calls itself to solve the smaller problem.
- By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop.
- In mathematics, the notation *n*! represents the factorial of the number *n*.

Solving Problems With Recursion

- The factorial of a nonnegative number can be defined by the following rules:
  - If $n = 0$ then $n! = 1$
  - If $n > 0$ then $n! = 1 \times 2 \times 3 \times \ldots \times n$
- Let's replace the notation $n!$ with factorial(n), which looks a bit more like computer code, and rewrite these rules as:
  - If $n = 0$ then factorial(n) = 1
  - If $n > 0$ then factorial(n) = $1 \times 2 \times 3 \times \ldots \times n$

Solving Problems With Recursion

- These rules state that:
  - when *n* is 0, its factorial is 1, and
  - when *n* greater than 0, its factorial is the product of all the positive integers from 1 up to *n*.
- Factorial(6) is calculated as
  - 1 × 2 × 3 × 4 × 5 × 6.
- The base case is where *n* is equal to 0:

  if *n* = 0 then factorial(*n*) = 1
- The recursive case, or the part of the problem that we use recursion to solve is:
  - ***if n > 0 then factorial(n) = n × factorial(n – 1)***

Solving Problems With Recursion

- The recursive call works on a reduced version of the problem, $n - 1$.
- The recursive rule for calculating the factorial:
  - If $n = 0$ then factorial(n) = 1
  - If $n > 0$ then factorial(n) = $n \times$ factorial($n - 1$)
- A Java based solution:

```
private static int factorial(int n)
{
  if (n == 0) return 1; // Base case
  else return n * factorial(n - 1);
}
```

- Example: FactorialDemo.java

Solving Problems With Recursion

The method is first called from the `main` method of the `FactorialDemo` class.