Chapter 15: Recursion

Starting Out with Java
From Control Structures through Data Structures 1e

by Tony Gaddis and Godfrey Muganda

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Chapter Topics

Chapter 15 discusses the following main topics:

- Introduction to Recursion
- Solving Problems with Recursion
- Examples of Recursive Methods
- A Recursive Binary Search Method
- The Towers of Hanoi

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

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

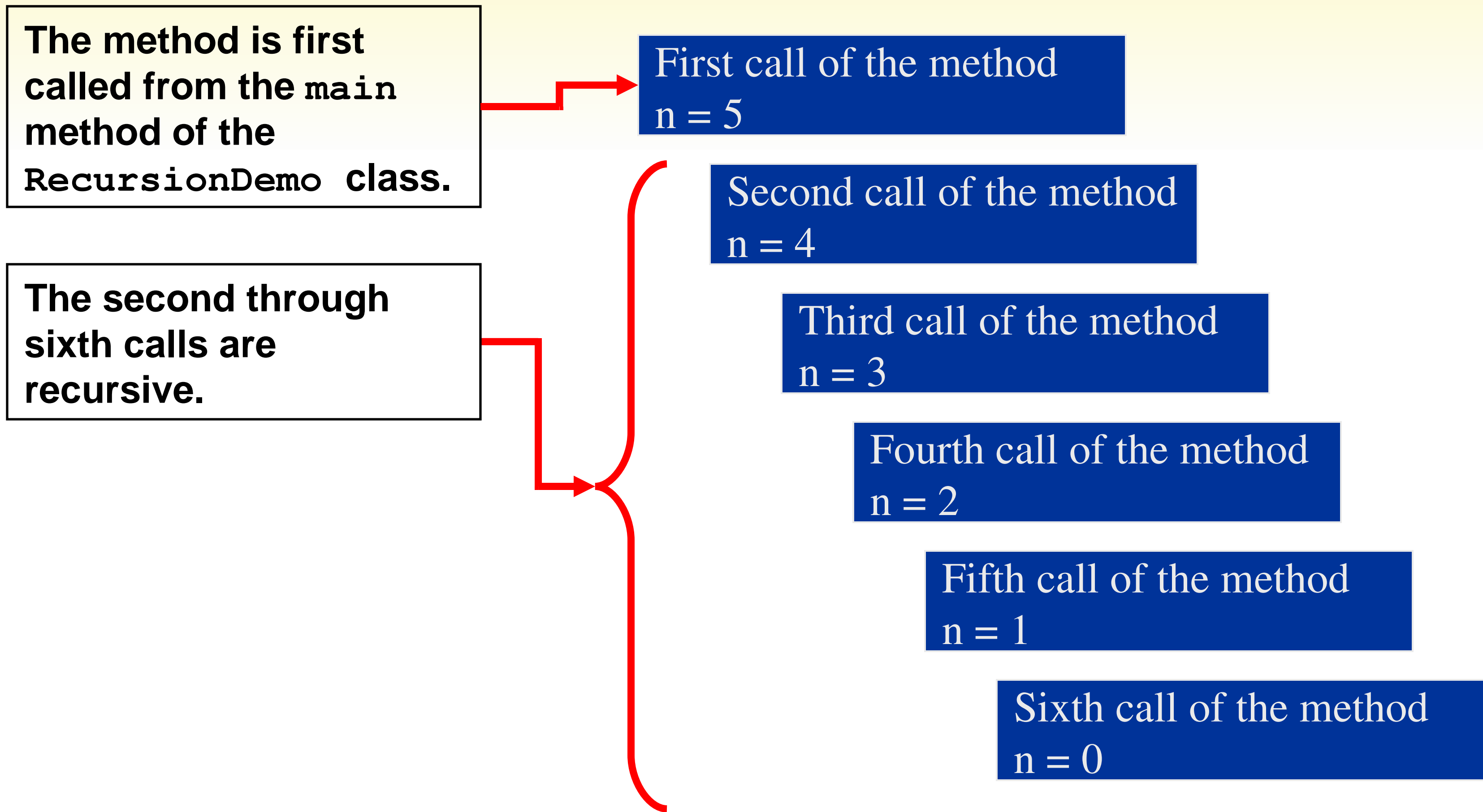
Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

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

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Introduction to Recursion



Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

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.

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

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.

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

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.

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

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) = $1 \times 2 \times 3 \times \ldots \times n$

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

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
  - $1 \times 2 \times 3 \times 4 \times 5 \times 6$.
- 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)***

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Solving Problems With Recursion

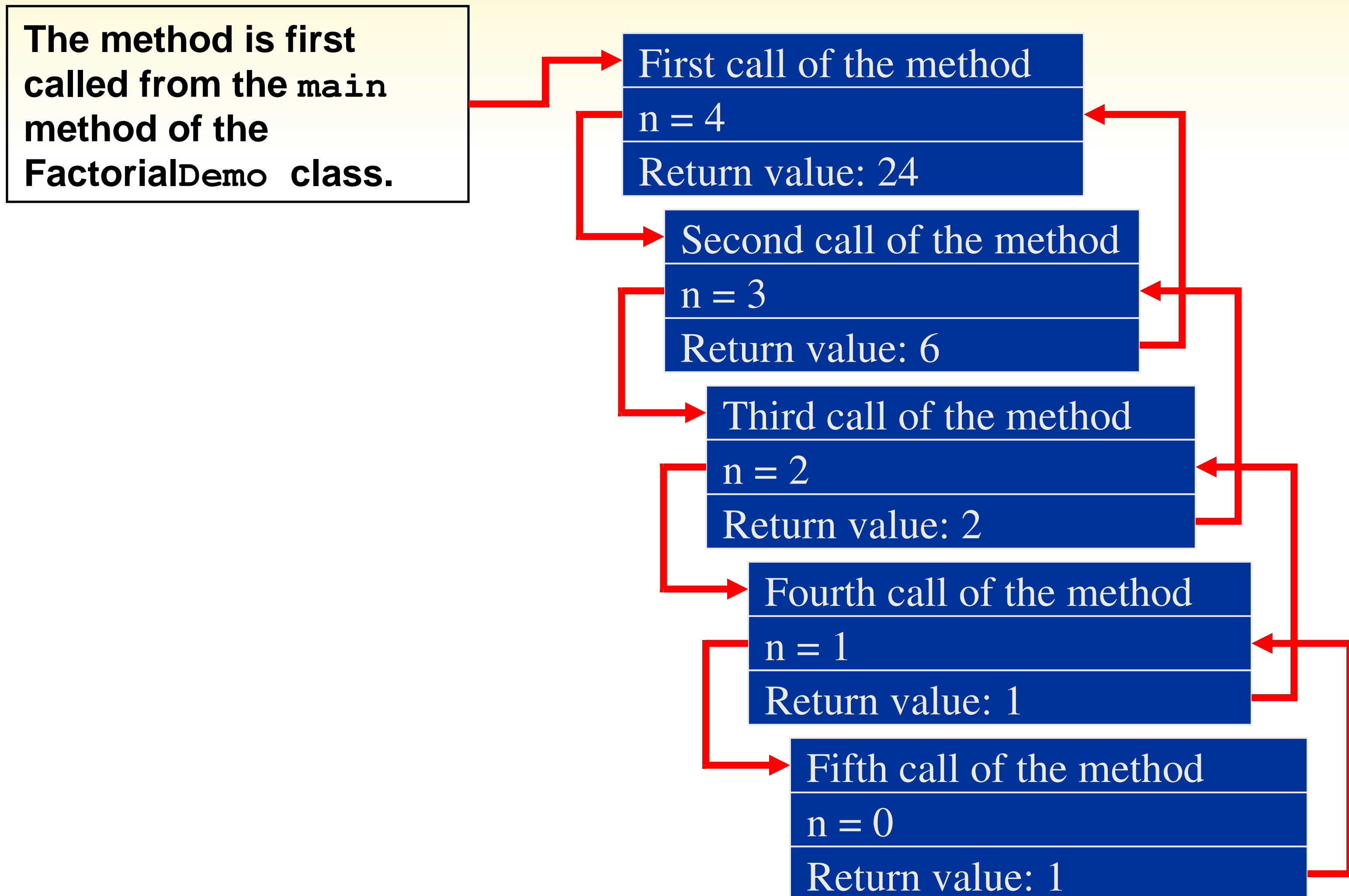
- The recursive call works on a reduced version of the problem, $n - 1$.
- The recursive rule for calculating the factorial:
  - If $n = 0$ then factorial(n) = 1
  - If $n > 0$ then factorial(n) = $n \times$ factorial($n - 1$)
- A Java based solution:

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

- Example: FactorialDemo.java

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Solving Problems With Recursion



Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Direct and Indirect Recursion

- When recursive methods directly call themselves it is known as *direct recursion*.
- *Indirect recursion* is when method A calls method B, which in turn calls method A.
- There can even be several methods involved in the recursion.
- Example, method A could call method B, which could call method C, which calls method A.
- Care must be used in indirect recursion to ensure that the proper base cases and return values are handled.

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Summing a Range of Array Elements

- **Recursion can be used to sum** a range of array elements.
- A method, rangeSum takes following arguments:
  - **int** array,
  - an **int** specifying the starting element of the range, and
  - an **int** specifying the ending element of the range.
  - How it might be called:

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int sum;
sum = rangeSum(numbers, 3, 7);
```

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Summing a Range of Array Elements

- The definition of the **`rangeSum`** method:

```
public static int rangeSum(int[] array,
                           int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start]
           + rangeSum(array, start + 1, end);
}
```

- Example:
  - RangeSum.java

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Drawing Concentric Circles

- The definition of the **`drawCircles`** method:

```
private void drawCircles(Graphics g, int n, int topXY, intƒsize)
{
    if (n > 0)
    {
        g.drawOval(topXY, topXY, size, size);
        drawCircles(g, n - 1, topXY + 15, size - 30);
    }
}
```

- Example:
  - Circles.java

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

The Fibonacci Series

- Some mathematical problems are designed to be solved recursively.
- One well known example is the calculation of *Fibonacci numbers.*:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,...
- After the second number, each number in the series is the sum of the two previous numbers.
- The Fibonacci series can be defined as:
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_N = F_{N-1} + F_{N-2}$ for $N \geq 2$.

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

The Fibonacci Series

```
public static int fib(int n)
{
  if (n == 0)
    return 0;
  else if (n == 1)
      return 1;
  else
      return fib(n - 1) + fib(n - 2);
}
```

- This method has two base cases:
  - when **n** is equal to 0, and
  - when **n** is equal to 1.
- Example: FibNumbers.java

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Greatest Common Divisor (GCD)

- The definition of the **`gcd`** method:

```
public static int gcd(int x, int y)
{
     if (x % y == 0)
        return y;
     else
        return gcd(y, x % y);
 }
```

- Example:
  - GCDdemo.java

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Recursive Binary Search

- The binary search algorithm can be implemented recursively.
- The procedure can be expressed as:

```
If array[middle] equals the search value, then the
  value is found.
Else
  if array[middle] is less than the search value,
   do a binary search on the upper half of the array.
Else
  if array[middle] is greater than the search value,
  perform a binary search on the lower half of the
  array.
```

- Example: RecursiveBinarySearch.java

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

The Towers of Hanoi

- The Towers of Hanoi is a mathematical game that uses:
  - three pegs and
  - a set of discs with holes through their centers.
- The discs are stacked on the leftmost peg, in order of size with the largest disc at the bottom.
- The object of the game is to move the pegs from the left peg to the right peg by these rules:
  - Only one disk may be moved at a time.
  - A disk cannot be placed on top of a smaller disc.
  - All discs must be stored on a peg except while being moved.

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

The Towers of Hanoi

- The overall solution to the problem is to move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.
- This algorithm solves the game.

```
If n > 0 Then
    Move n – 1 discs from peg A to peg B,
    using peg C as a temporary peg.
    Move the remaining disc from the peg A to
    peg C.
    Move n – 1 discs from peg B to peg C,
    using peg A as a temporary peg.
End If
```

- The base case for the algorithm is reached when there are no more discs to move.
- Example: Hanoi.java, HanoiDemo.java

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley