Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines:
  - Homework 5: this Friday @ 11:59 pm
  - Lab 4: next Monday @ 11:59 pm
  - Programming Assignment 1: ~~Friday Oct. 7th~~ Monday Oct. 10th
    - Autograder is up on GradeScope
- If you think you lost points in a lab assignment because of the autograder or because of a simple mistake
  - please reach out to Grader TA over Piazza
- **Live Remote Support Session** for Assignment 1
  - Recording and slides on Canvas
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous Lecture …

- ADT Stack
  - Linked implementation
  - Implementation using ADT List
  - Application: Building a simple parser of Algebraic expressions

Today …

- ADT Stack
  - Application: Building a simple parser of Algebraic expressions
  - Application: Runtime stack
- Recursion

Our Plan for Processing Algebraic Expressions

1. Check if input infix expression is balanced
2. Convert the expression from infix to postfix
3. Evaluate the postfix expression

Our Plan for Processing Algebraic Expressions

1. Check if input infix expression is balanced
2. Convert the expression from infix to postfix
3. Evaluate the postfix expression

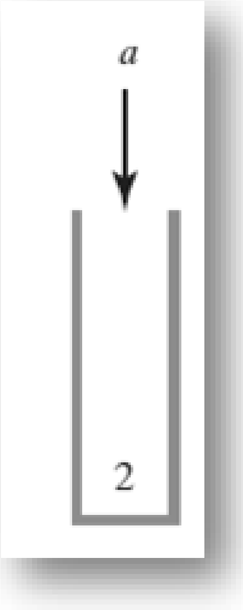
Step 3: Evaluating Postfix Expressions

1. Initialize an empty Stack
2. for each character in postfix expression
    1. if variable, push its value to Stack
    2. if operator
        1. pop second operand
        2. pop first operand
        3. apply operator to two operands
        4. push result
3. Return the remaining value in Stack

Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

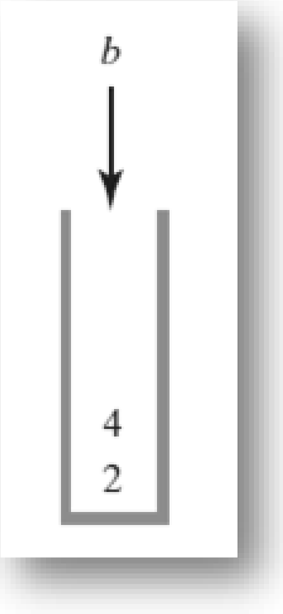
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

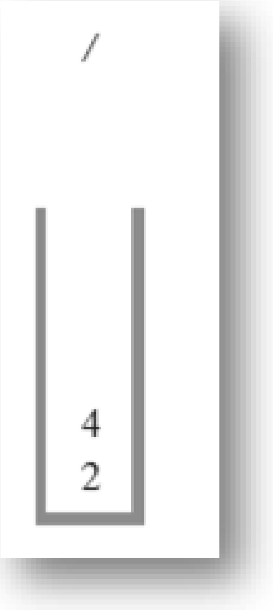
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

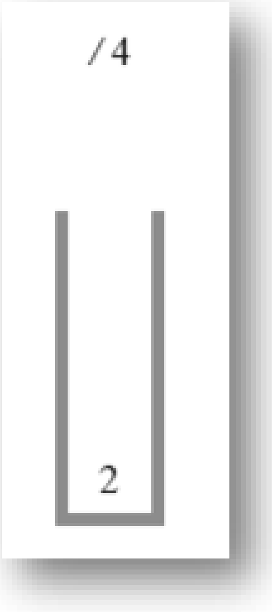
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

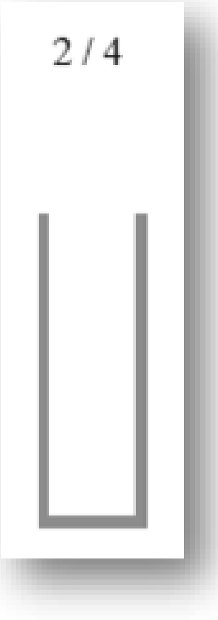
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

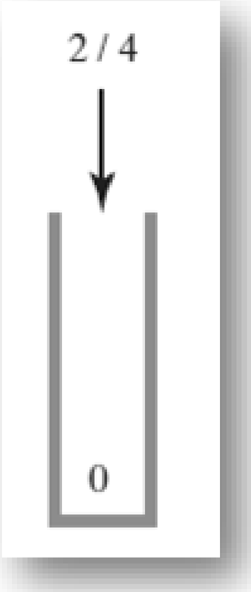
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

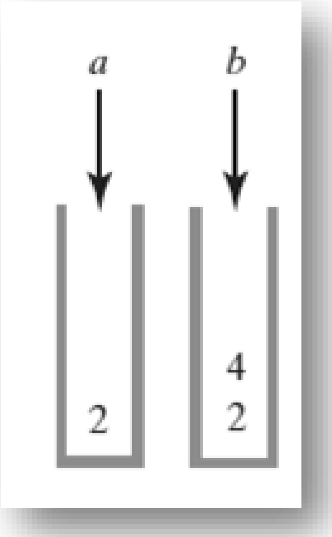
The stack during the evaluation of the postfix expression

a b /   when *a* is 2 and b is 4



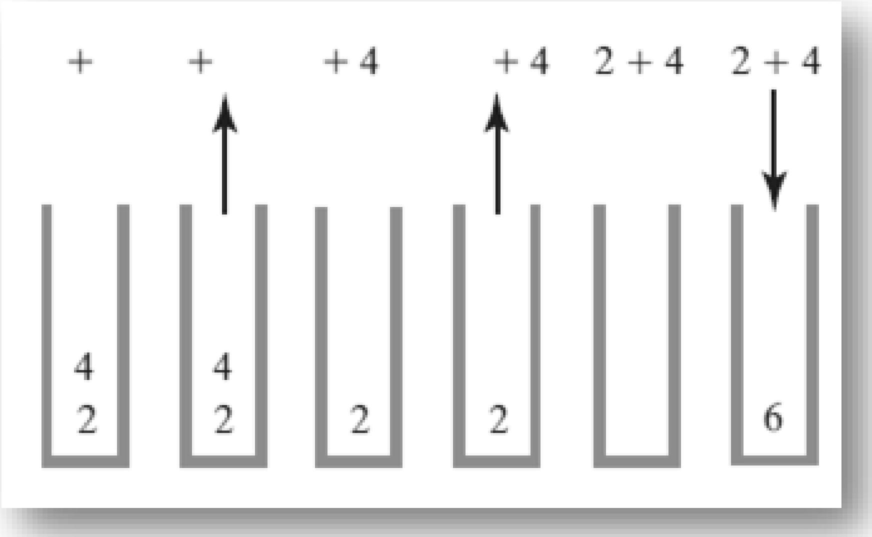
Evaluating Postfix Expressions: Example 2

The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3



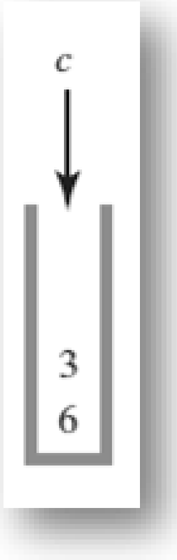
Evaluating Postfix Expressions: Example 2

The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3



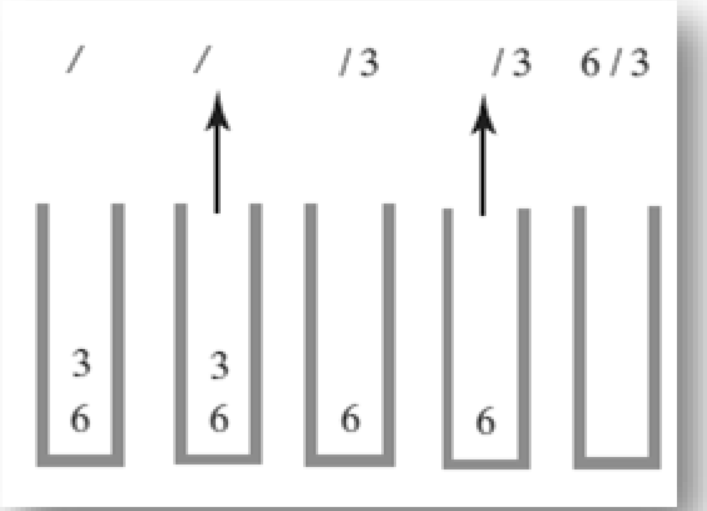
Evaluating Postfix Expressions: Example 2

The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3



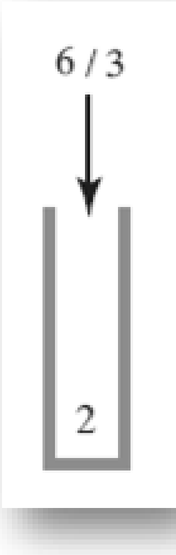
Evaluating Postfix Expressions: Example 2

The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3



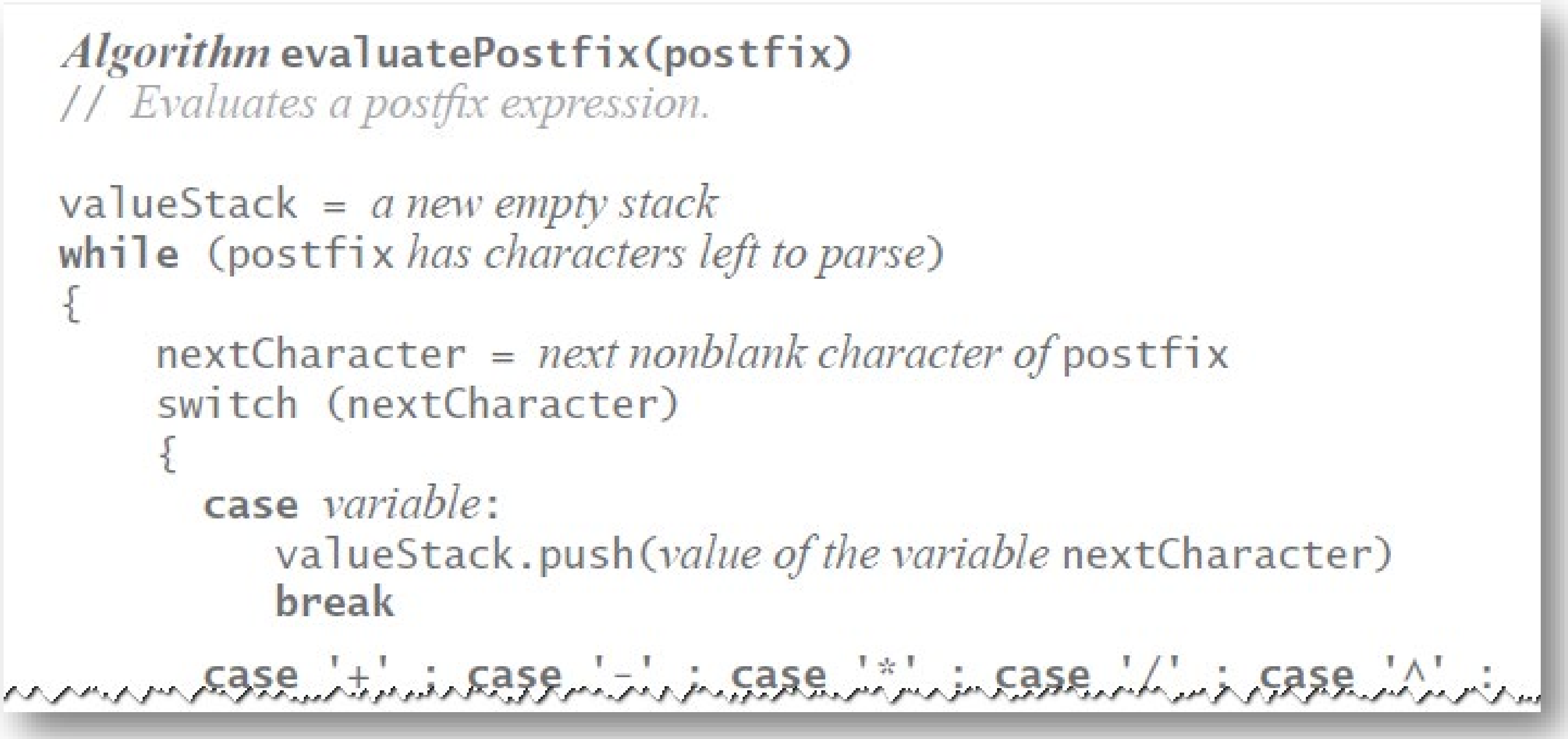
Evaluating Postfix Expressions: Example 2

The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3



Evaluating Postfix Expressions

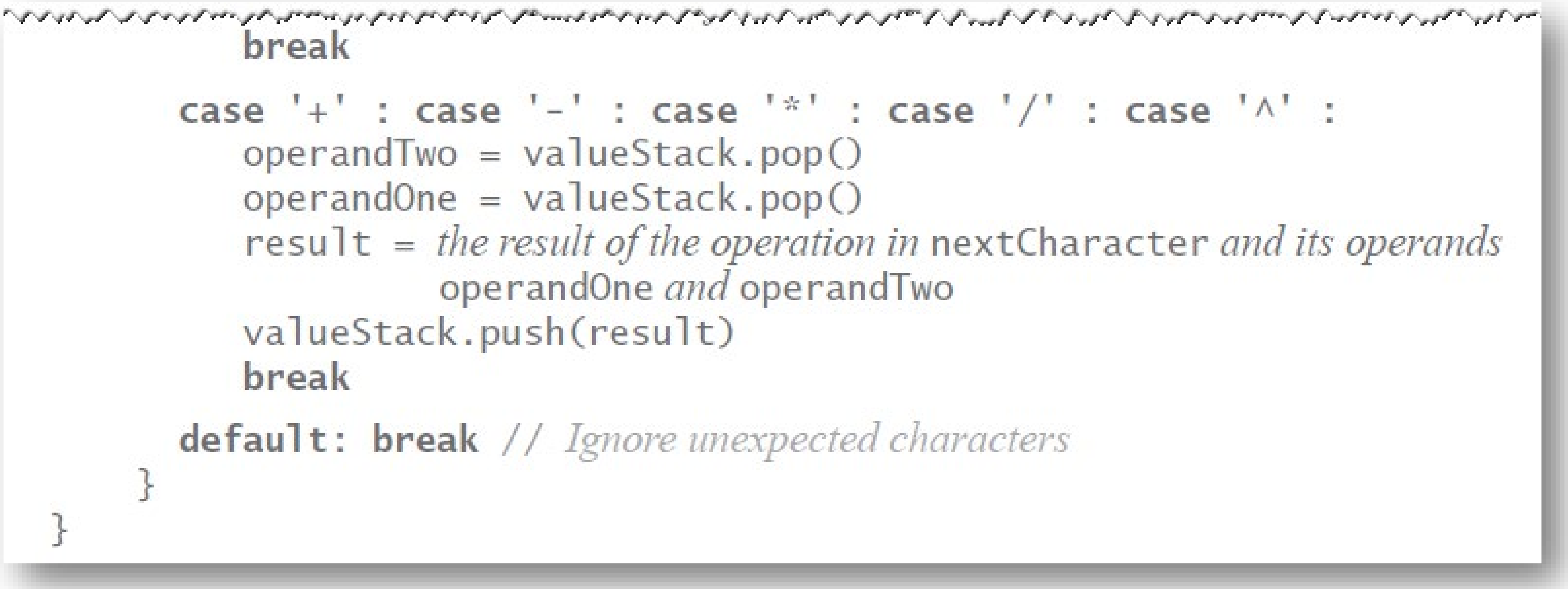
- Algorithm for evaluating postfix expressions.



```
Algorithm evaluatePostfix(postfix)
// Evaluates a postfix expression.

valueStack = a new empty stack
while (postfix has characters left to parse)
{
     nextCharacter = next nonblank character of postfix
     switch (nextCharacter)
     {
       case variable:
          valueStack.push(value of the variable nextCharacter)
          break
       case '+' : case '-' : case '*' : case '/' : case '^' :
```

Evaluating Postfix Expressions

- Algorithm for evaluating postfix expressions.



```
            break

        case '+' : case '-' : case '*' : case '/' : case '^' :
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in nextCharacter and its operands
                     operandOne and operandTwo
            valueStack.push(result)
            break

        default: break // Ignore unexpected characters
    }
}
```

What is the running time?

- in terms of *n,* the length of the input prefix string
- Check balance
    - how many times does each character get pushed?
        - at most 1
    - how many times does each character get poped?
        - at most 1
    - What is the runtime of push and pop?
        - O(1)
    - O(n)
- Convert infix to postfix: O(n)
- Evaluate postfix: O(n)
- Total: O(3n) = O(n)
- Three passes!
- Can we do better?
- Yes! We can use two passes only
    - Expect to require more space
    - space-time tradeoff

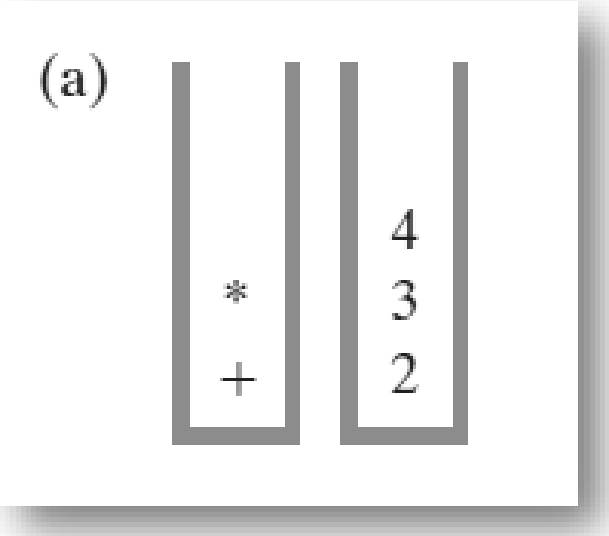
Evaluating Infix Expressions with 2 passes only

- We will use two stacks
  - Operator Stack
  - Operand stack
- Scan the expression once:
  - follow the steps of infix conversion to postfix,
  - <u>**except**</u>
    - instead of appending to postfix output, push to operand stack
    - when popping an operator, pop second then first operands, apply operator, push result to operand stack
- While operator stack not empty
  - pop an operator
  - pop second operand then first operand
  - apply the operator and push result to operand stack
- Result is the remaining value in the operand stack

Evaluating Infix Expressions with 2 passes only

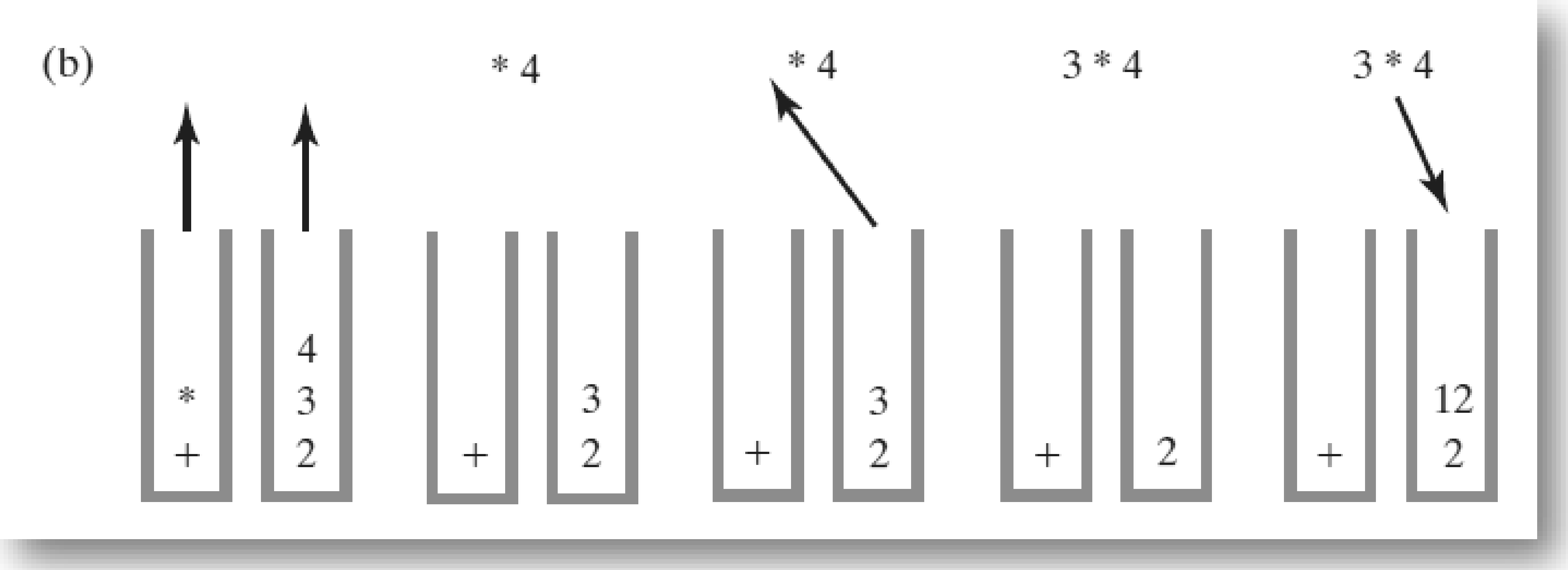
Two stacks during the evaluation of
a + b * c when a is 2, b is 3, and c is 4:

- after reaching the end of the expression;



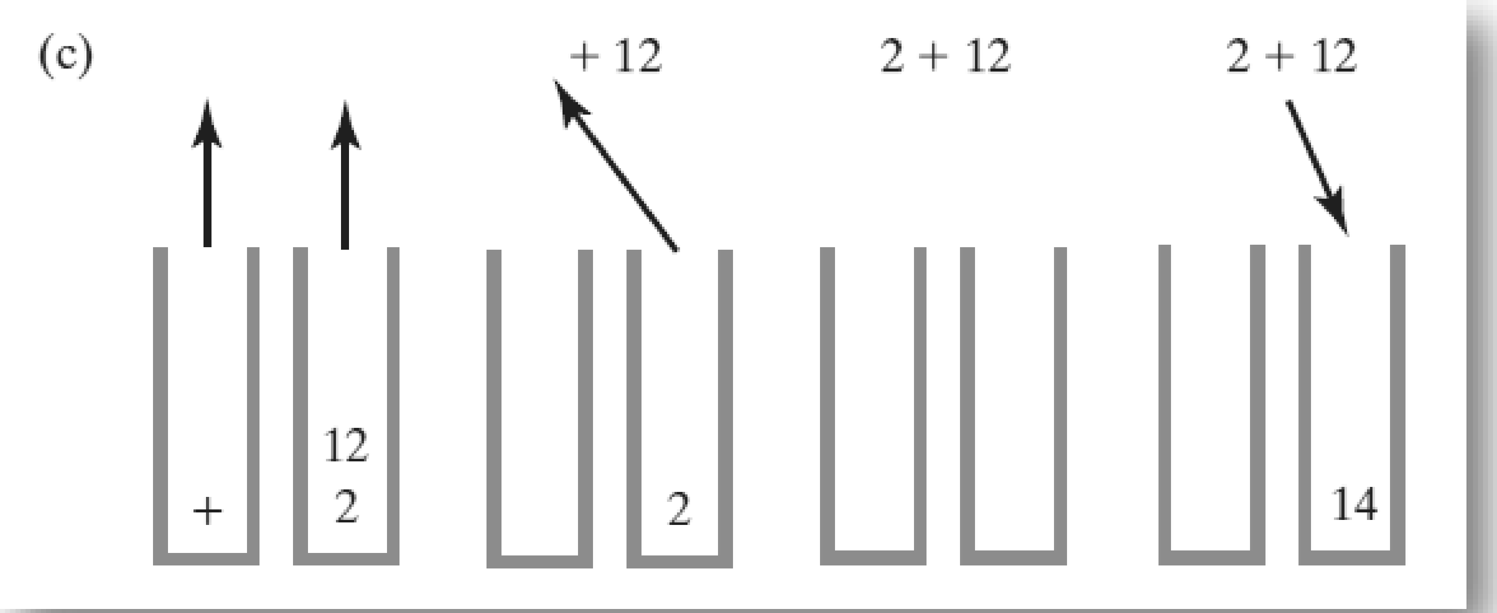
Evaluating Infix Expressions

Two stacks during the evaluation of
a + b * c when a is 2, b is 3, and c is 4:
while performing the multiplication;



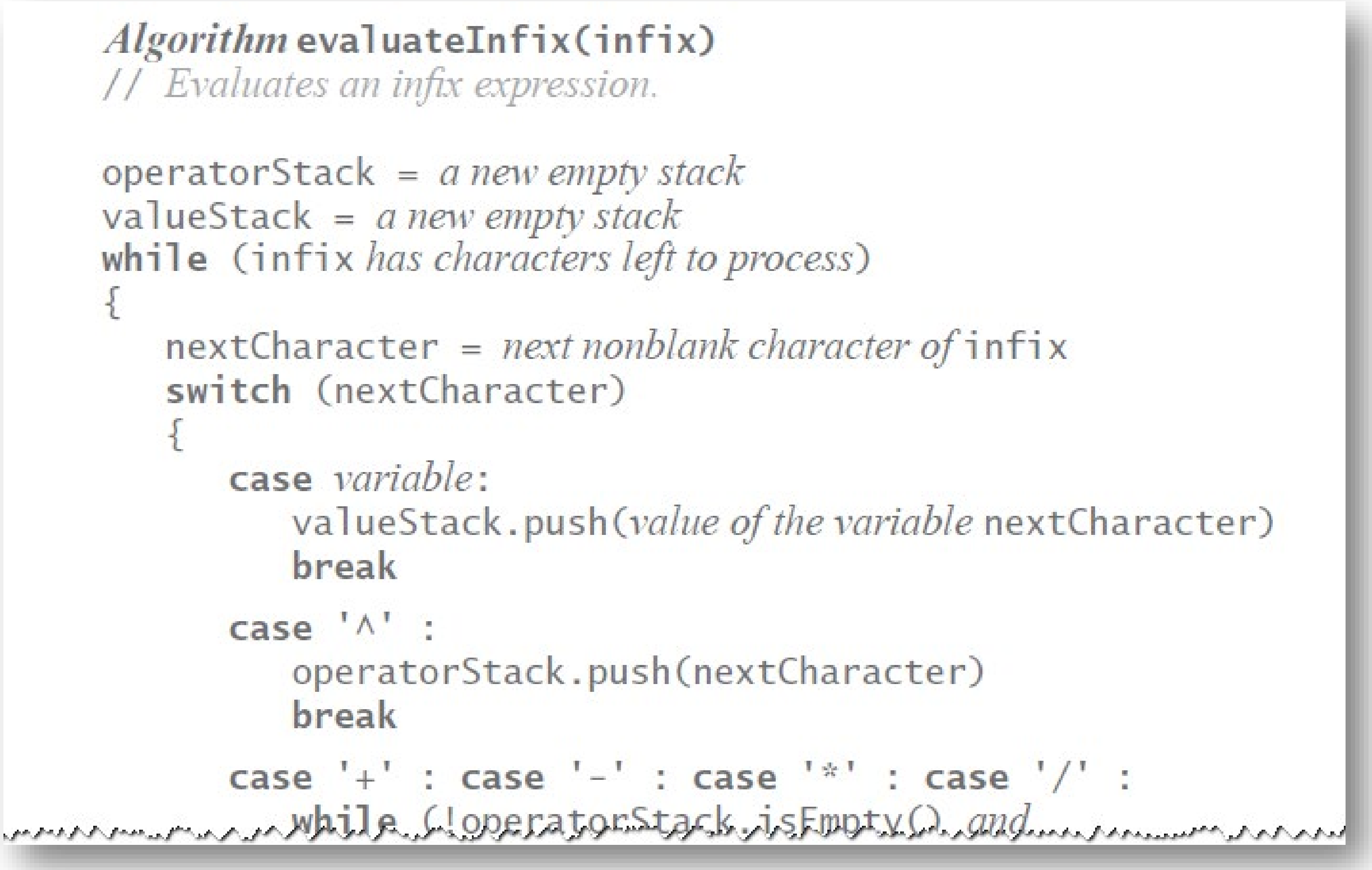
Evaluating Infix Expressions

Two stacks during the evaluation of
a + b * c when a is 2, b is 3, and c is 4:
(c) while performing the addition



Evaluating Infix Expressions

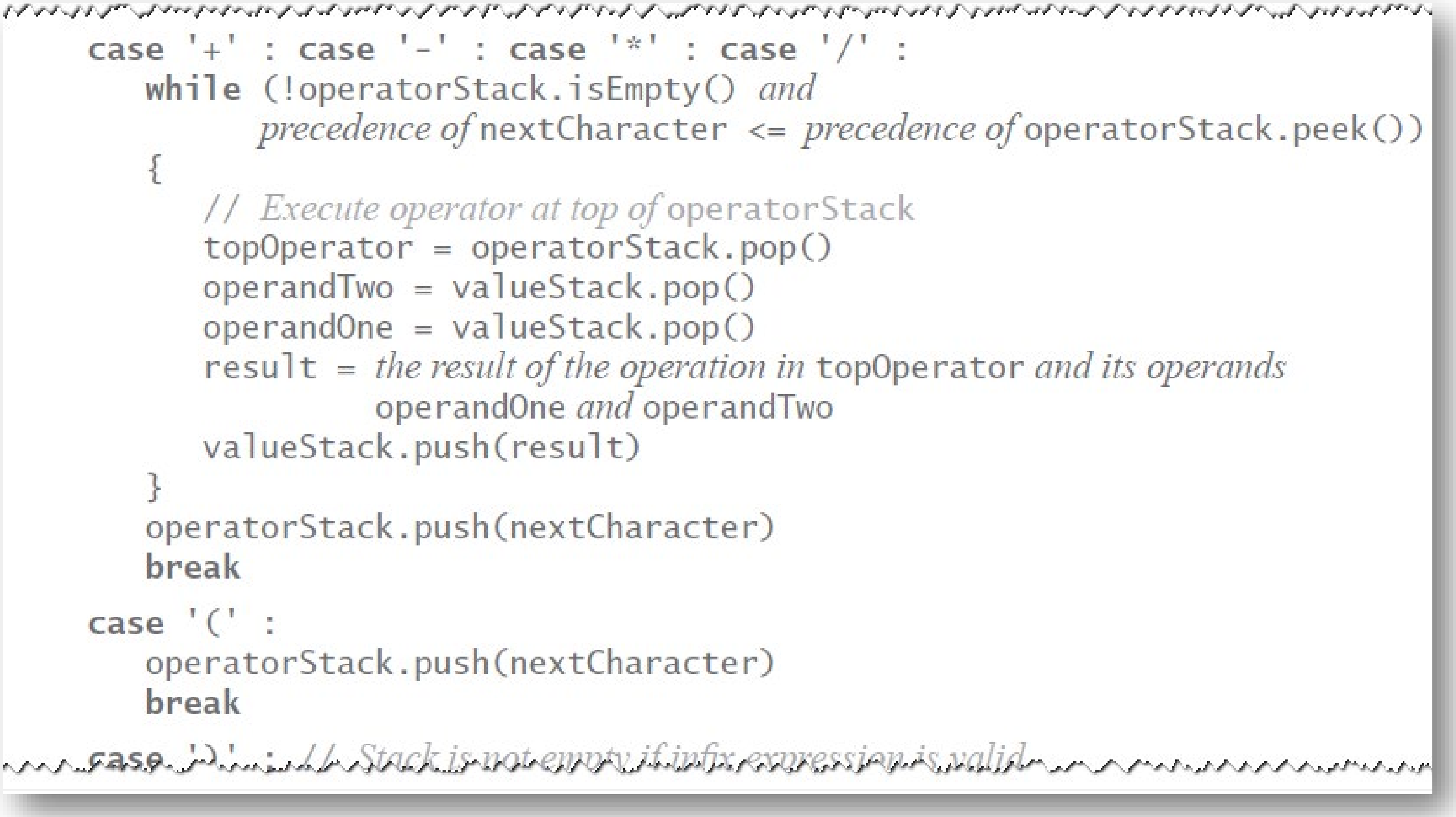
- Algorithm to evaluate infix expression.



```
Algorithm evaluateInfix(infix)
// Evaluates an infix expression.

operatorStack = a new empty stack
valueStack = a new empty stack
while (infix has characters left to process)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break

      case '^' :
         operatorStack.push(nextCharacter)
         break

      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
```

Evaluating Infix Expressions

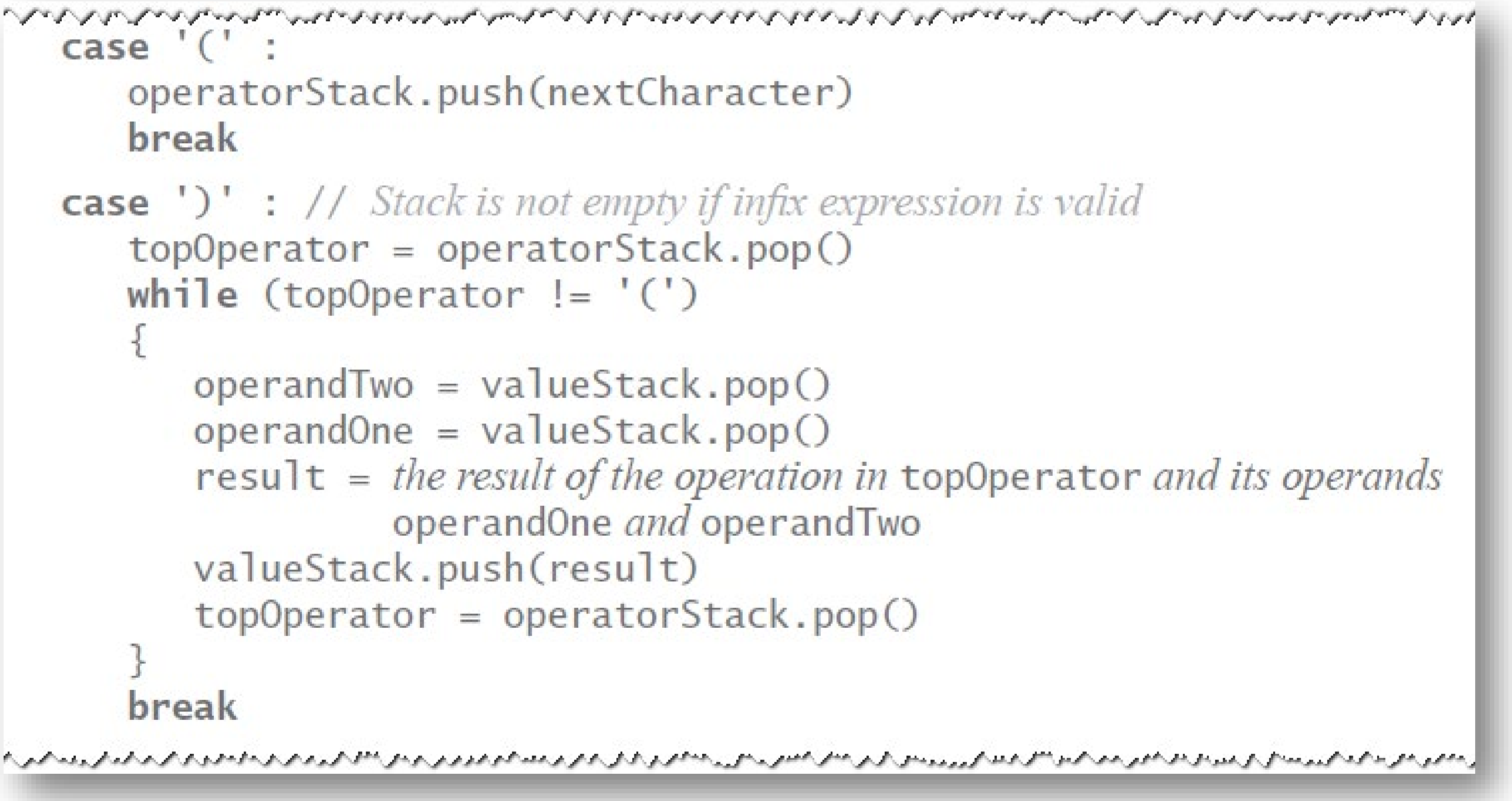
- Algorithm to evaluate infix expression.



```
case '+' : case '-' : case '*' : case '/' :
   while (!operatorStack.isEmpty() and
         precedence of nextCharacter <= precedence of operatorStack.peek())
   {
      // Execute operator at top of operatorStack
      topOperator = operatorStack.pop()
      operandTwo = valueStack.pop()
      operandOne = valueStack.pop()
      result = the result of the operation in topOperator and its operands
               operandOne and operandTwo
      valueStack.push(result)
   }
   operatorStack.push(nextCharacter)
   break
case '(' :
   operatorStack.push(nextCharacter)
   break
case ')' : // Stack is not empty if infix expression is valid
```

Evaluating Infix Expressions

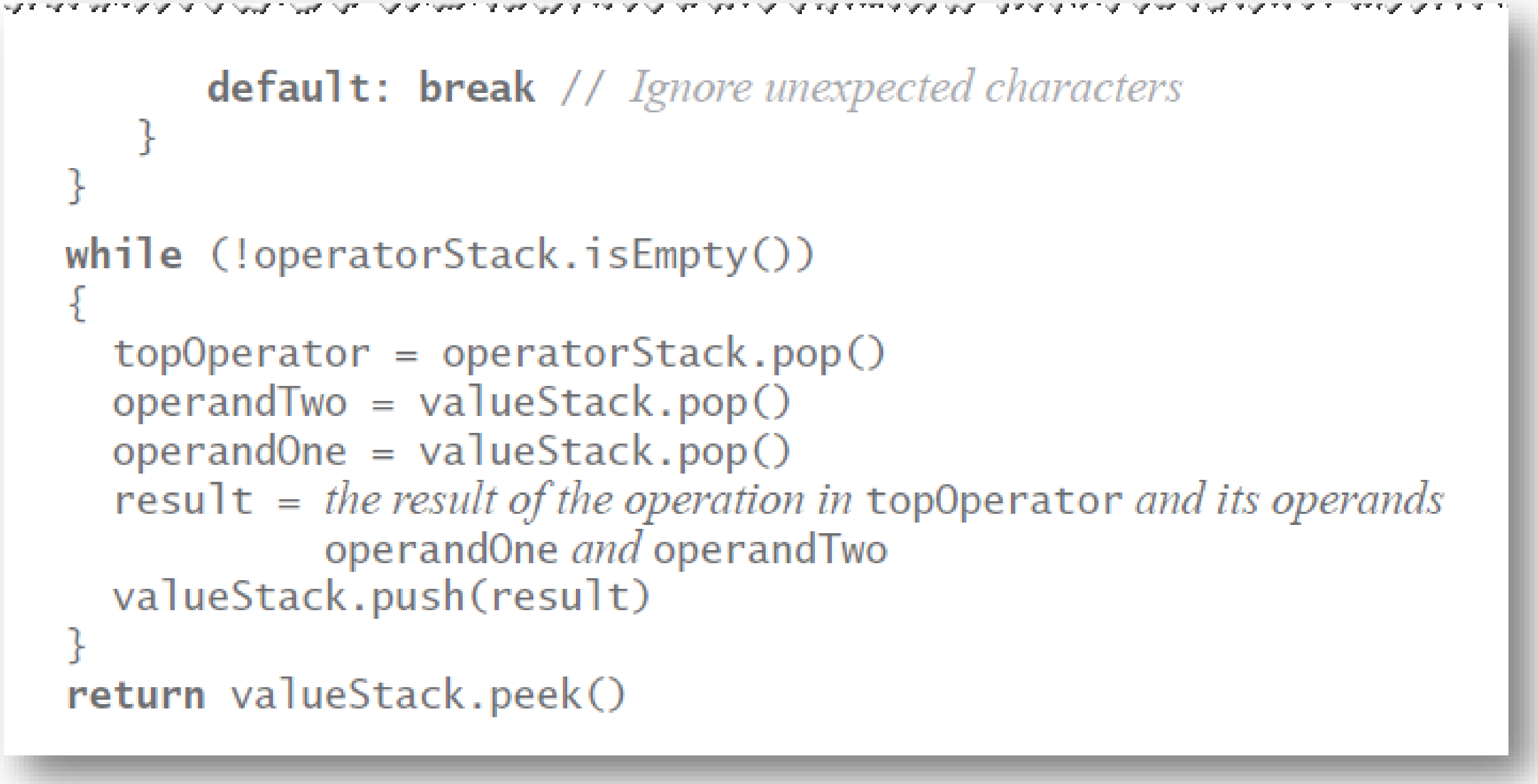
- Algorithm to evaluate infix expression.



```
case '(' :
   operatorStack.push(nextCharacter)
   break

case ')' : // Stack is not empty if infix expression is valid
   topOperator = operatorStack.pop()
   while (topOperator != '(')
   {
      operandTwo = valueStack.pop()
      operandOne = valueStack.pop()
      result = the result of the operation in topOperator and its operands
               operandOne and operandTwo
      valueStack.push(result)
      topOperator = operatorStack.pop()
   }
   break
```

Evaluating Infix Expressions

- Algorithm to evaluate infix expression.



```
      default: break // Ignore unexpected characters
   }
}
while (!operatorStack.isEmpty())
{
  topOperator = operatorStack.pop()
  operandTwo = valueStack.pop()
  operandOne = valueStack.pop()
  result = the result of the operation in topOperator and its operands
           operandOne and operandTwo
  valueStack.push(result)
}
return valueStack.peek()
```

The Runtime Stack (aka program stack)

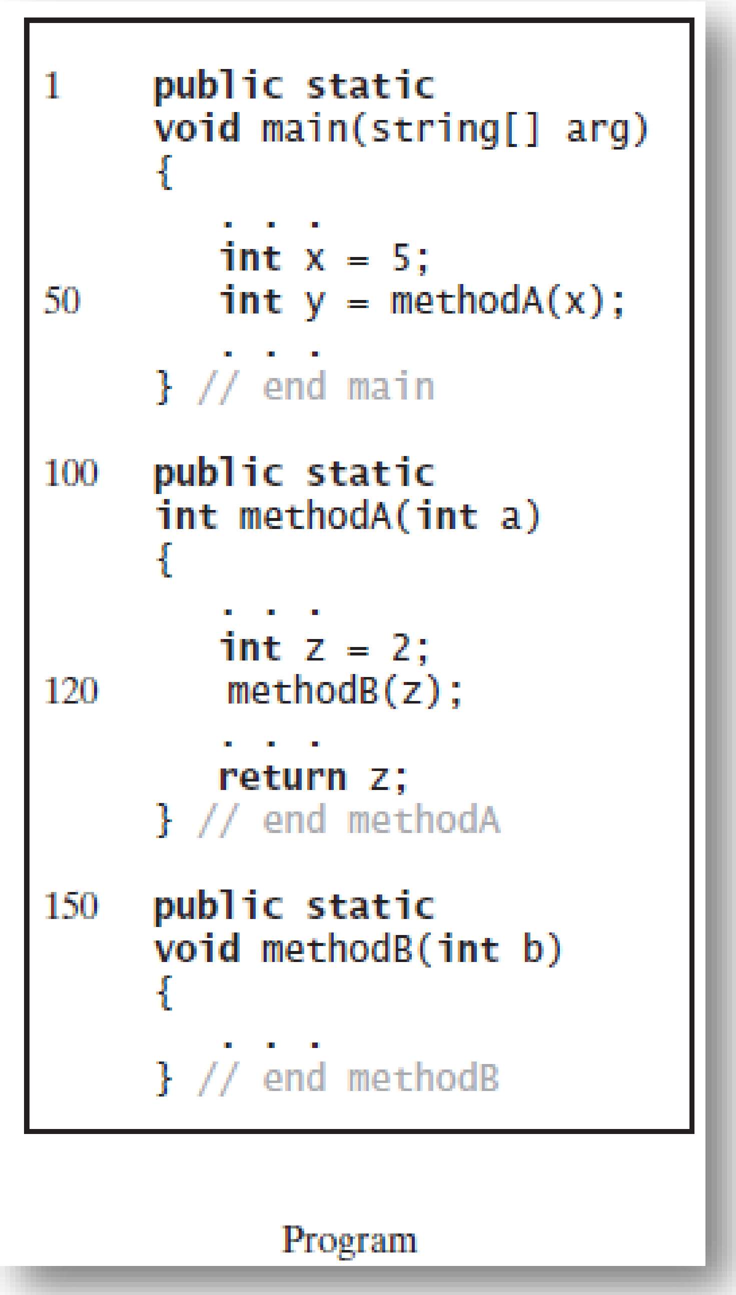
Under the hood/behind the scenes alert!

The Runtime Stack (aka program stack)

- Under the hood/behind the scenes alert!
- A stack is created for each running program
  - called **runtime stack**
- The stack is used to hold data for each method call
  - in an **activation record** (aka activation frame or just frame)
- Activation record stores:
  - method parameters
  - local (method) variables
  - address of return point (i.e., next statement to execute after returning from call)
- When a method is called, its activation record is *pushed* to the runtime stack
- When a method returns, the top activation record is *popped*

Example

- The following code has three methods:
  - main
  - methodA
  - method
- main calls methodA
- methodA calls method
- Side note: methodA and method must be static
  - because they are called from main, which must static
- What are the local variables of each method?
- What are the parameters of each method?

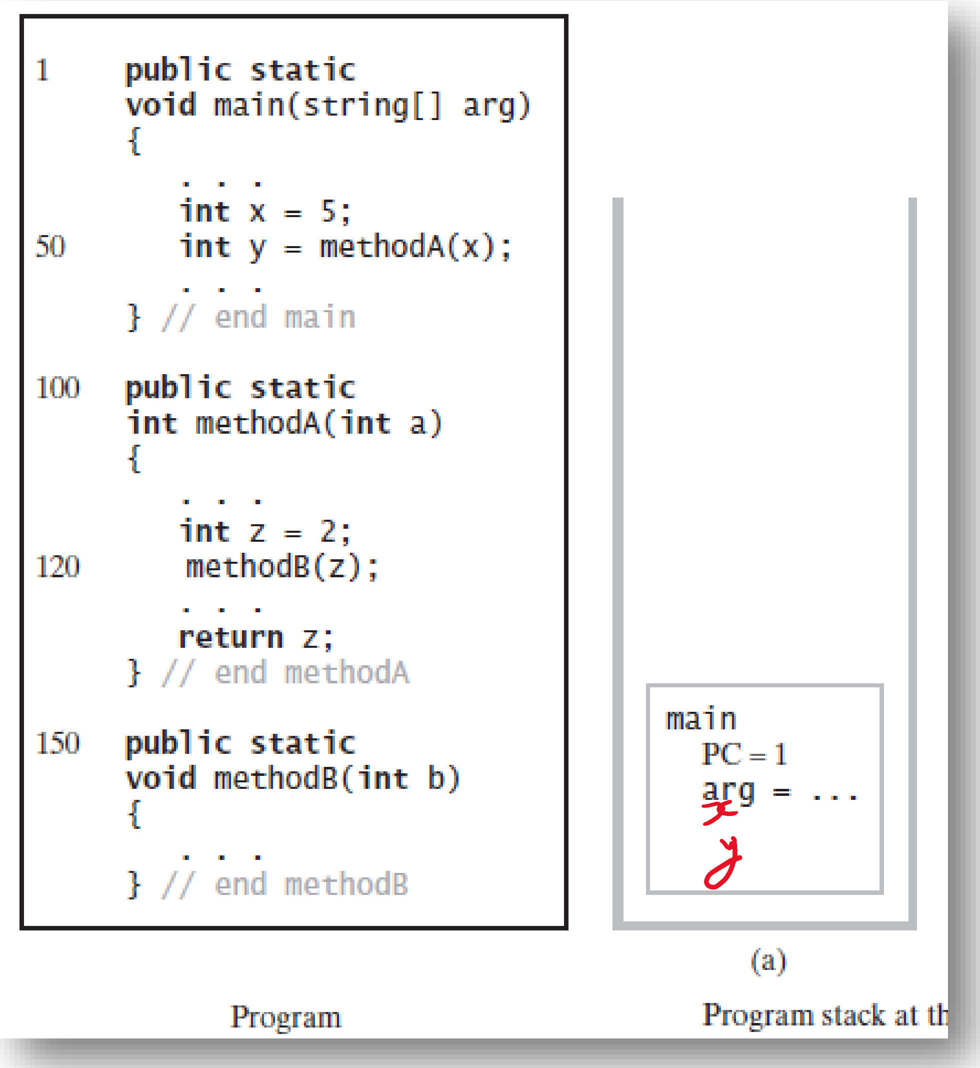


```
1      public static
       void main(string[] arg)
       {
          . . .
          int x = 5;
50        int y = methodA(x);
          . . .
       } // end main

100    public static
       int methodA(int a)
       {
          . . .
          int z = 2;
120       methodB(z);
          . . .
          return z;
       } // end methodA

150    public static
       void methodB(int b)
       {
          . . .
       } // end methodB
```

Program

The Program Stack

- The program stack when main begins execution
- PC is the Program Counter CPU register
  - it keeps track of the address of the next instruction to execute



```
1       public static
        void main(string[] arg)
        {
           . . .
           int x = 5;
50         int y = methodA(x);
           . . .
        } // end main

100     public static
        int methodA(int a)
        {
           . . .
           int z = 2;
120        methodB(z);
           . . .
           return z;
        } // end methodA

150     public static
        void methodB(int b)
        {
           . . .
        } // end methodB
```

```
main
  PC = 1
  arg = ...
  x
  y
```

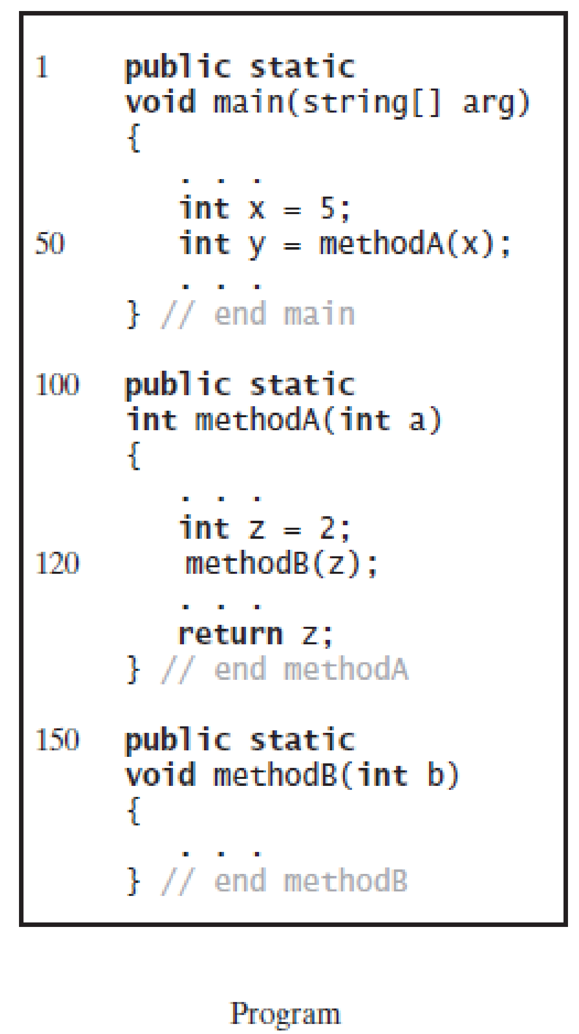
(a)

Program

Program stack at th

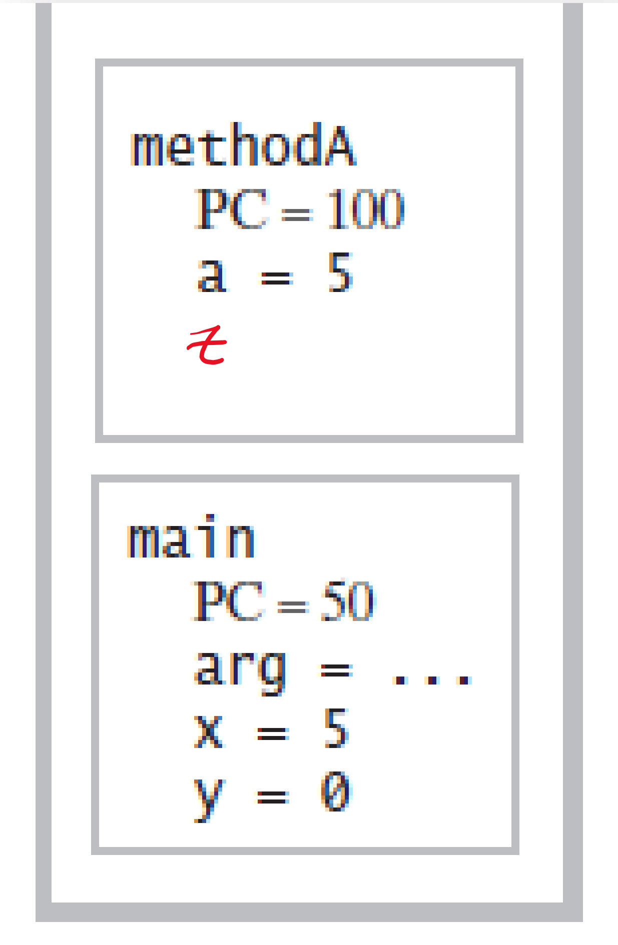
The Program Stack

- The program stack when methodA begins execution
- Before methodA starts, the value of PC is stored in the activation record of main



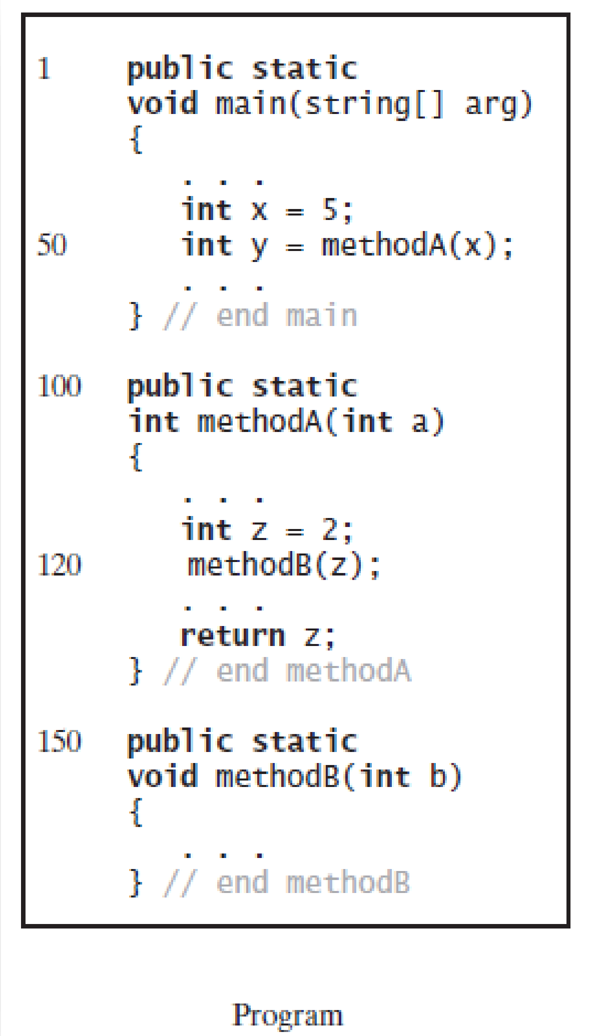
```
1     public static
      void main(string[] arg)
      {
         . . .
         int x = 5;
50       int y = methodA(x);
         . . .
      } // end main

100   public static
      int methodA(int a)
      {
         . . .
         int z = 2;
120      methodB(z);
         . . .
         return z;
      } // end methodA

150   public static
      void methodB(int b)
      {
         . . .
      } // end methodB
```

Program



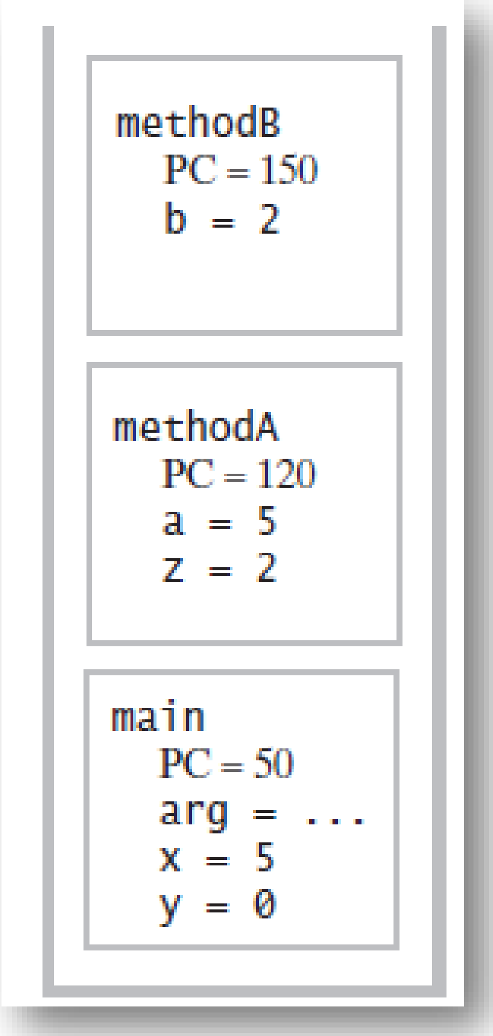
The Program Stack

- The program stack when methodB begins execution
- Before methodB starts, the value of PC is stored in the activation record of methodA



```
1     public static
      void main(string[] arg)
      {
         . . .
         int x = 5;
50       int y = methodA(x);
         . . .
      } // end main

100   public static
      int methodA(int a)
      {
         . . .
         int z = 2;
120       methodB(z);
         . . .
         return z;
      } // end methodA

150   public static
      void methodB(int b)
      {
         . . .
      } // end methodB
```

Program



```
methodB
   PC = 150
   b = 2

methodA
   PC = 120
   a = 5
   z = 2

main
   PC = 50
   arg = ...
   x = 5
   y = 0
```

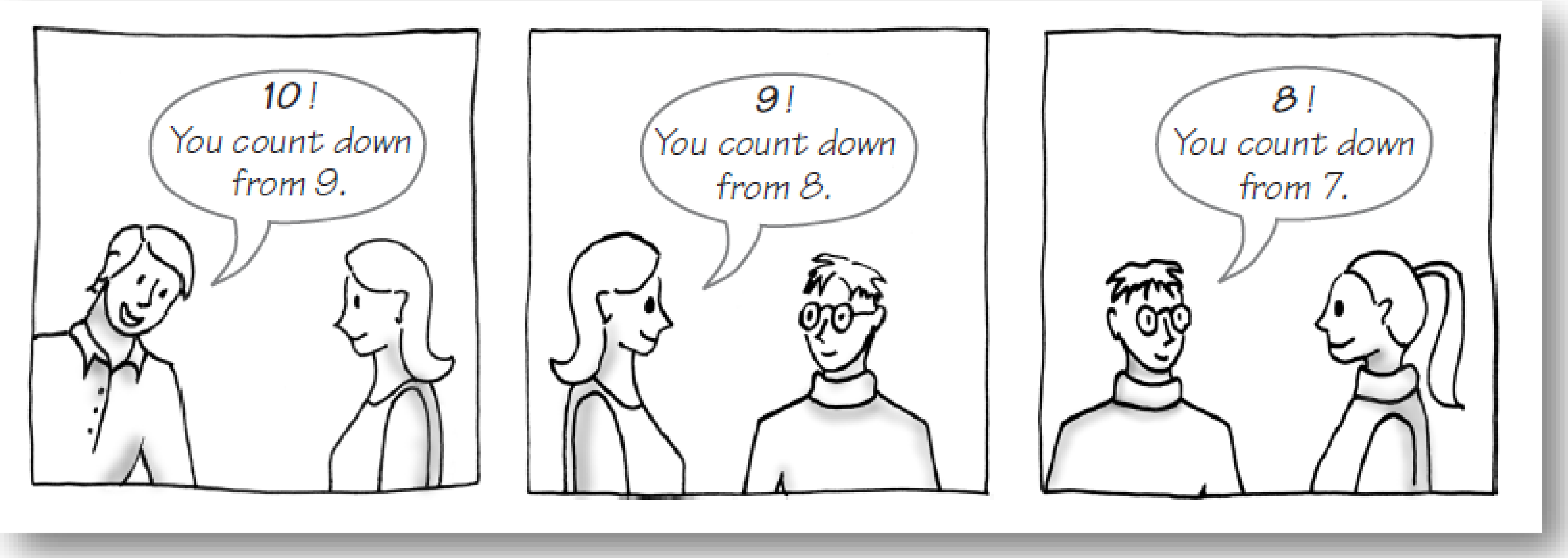
Recursion

What Is Recursion?

- Consider hiring a contractor to build
  - He hires a subcontractor for a portion of the job
  - That subcontractor hires a sub-subcontractor to do a smaller portion of job
- The last sub-sub- … subcontractor finishes
  - Each one finishes and reports “done” up the line

Example: The Countdown

Counting down from 10


10 ! You count down from 9.
9 ! You count down from 8.
8 ! You count down from 7.

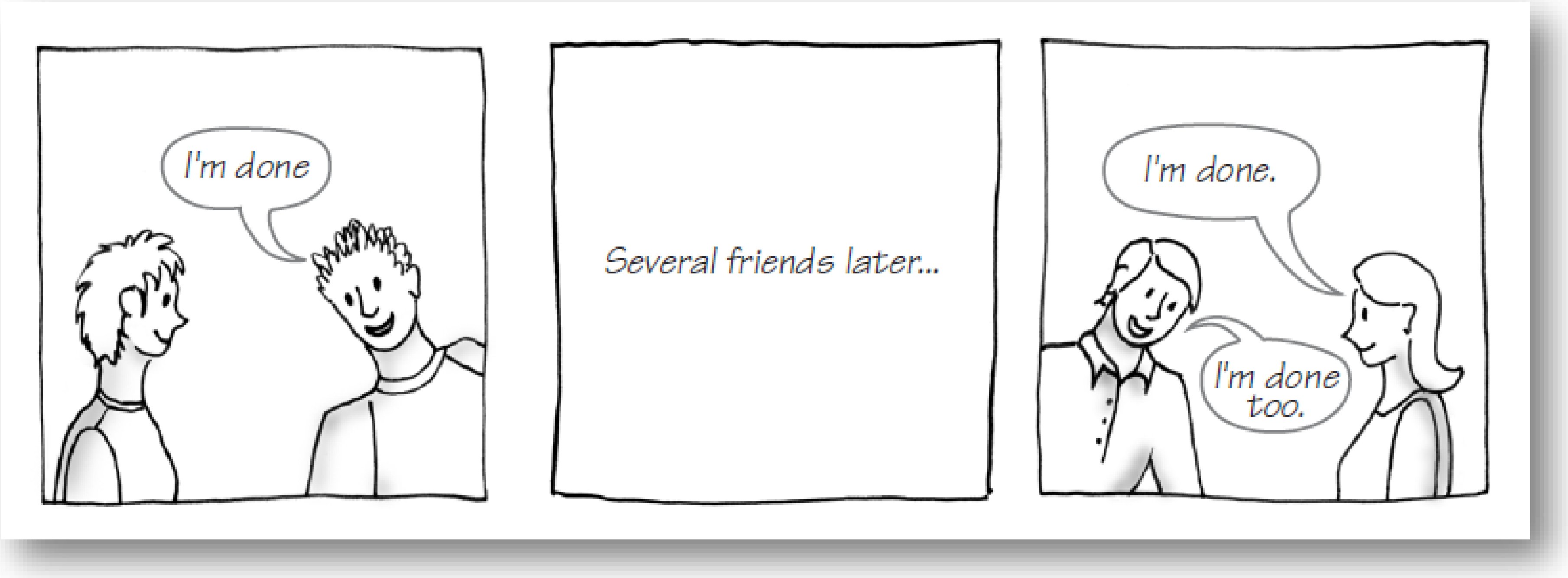
Example: The Countdown

Counting down from 10

Several friends later...
2 ! You count down from 1.
1 ! I'm done

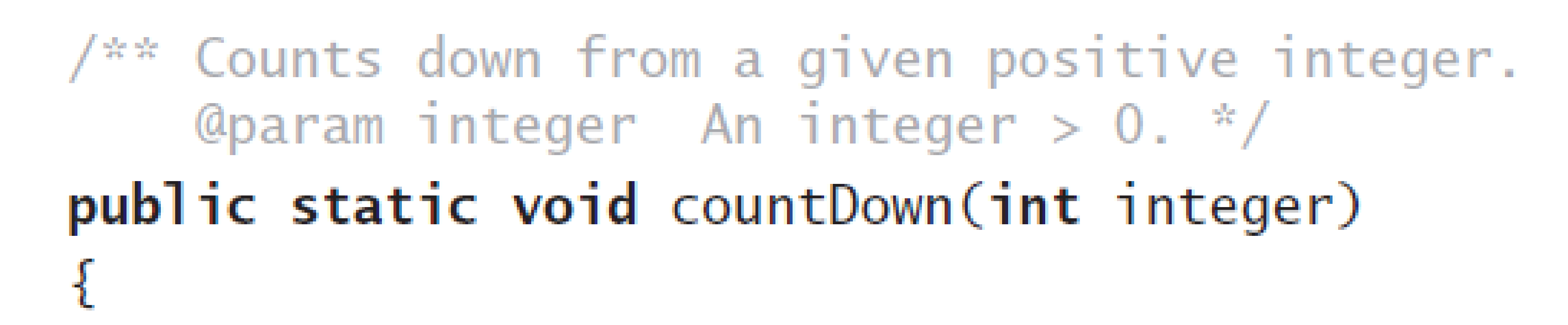
Example: The Countdown

Counting down from 10



Example: The Countdown

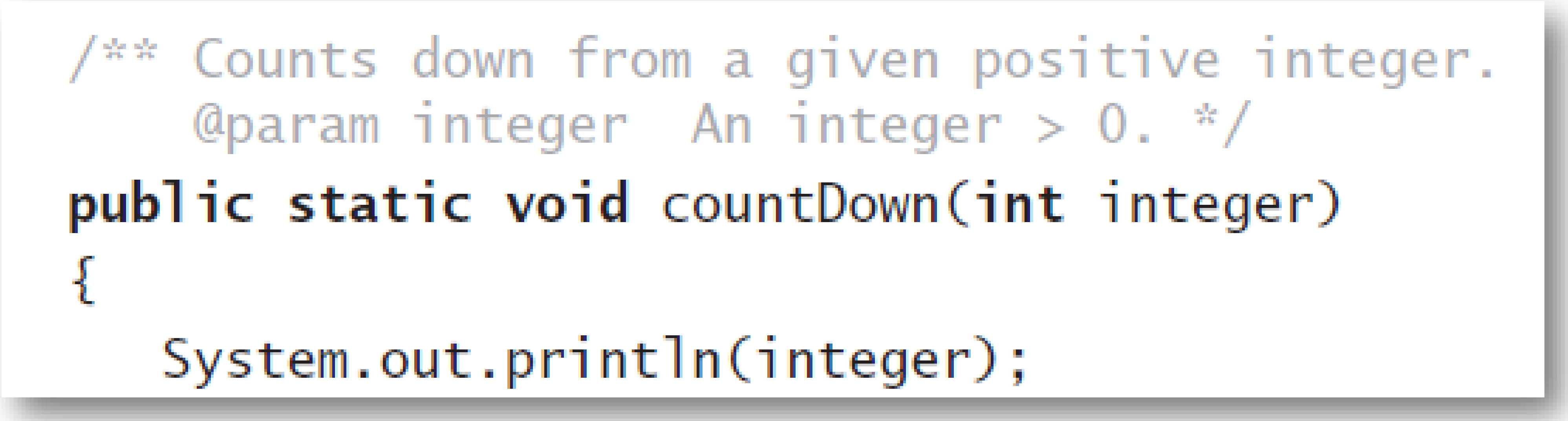
- Recursive Java method to do countdown.



```
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
```

Example: The Countdown

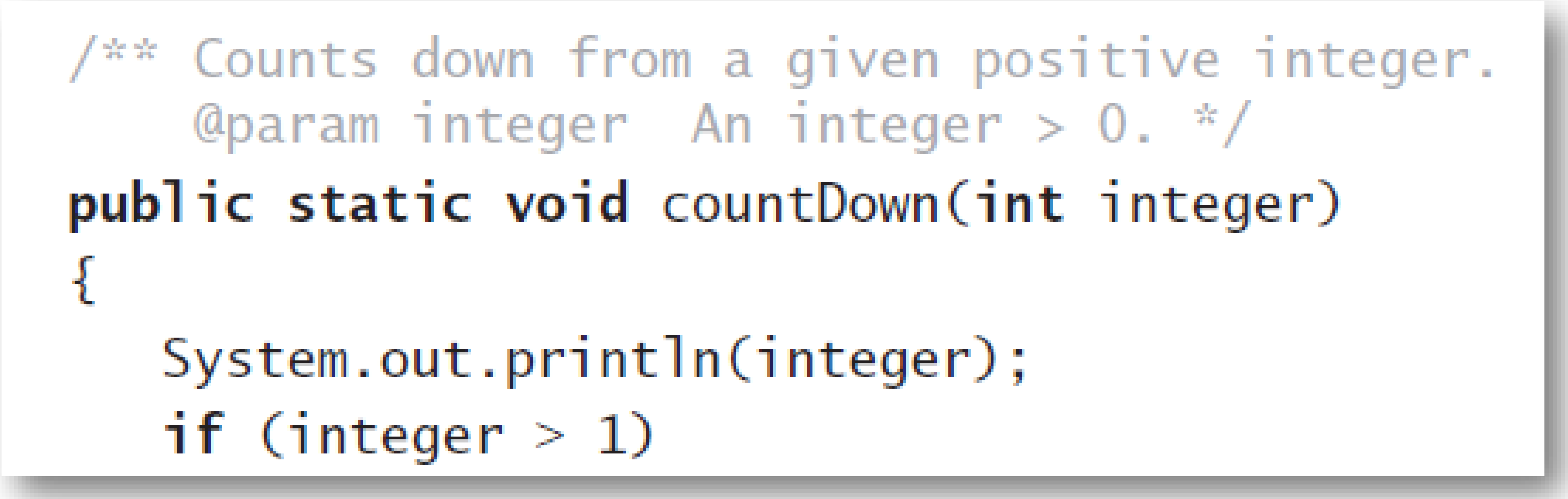
- Recursive Java method to do countdown.



```
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
   System.out.println(integer);
```

Example: The Countdown

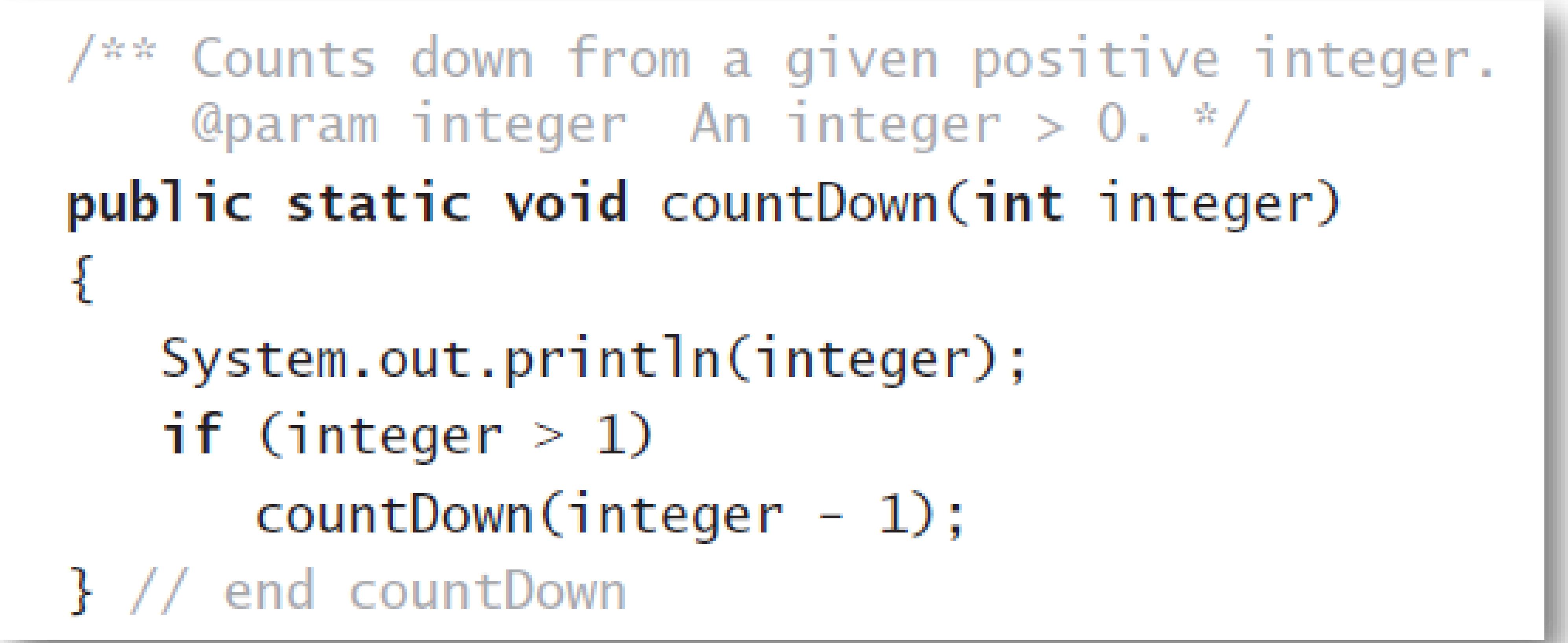
- Recursive Java method to do countdown.



```
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
   System.out.println(integer);
   if (integer > 1)
```

Example: The Countdown

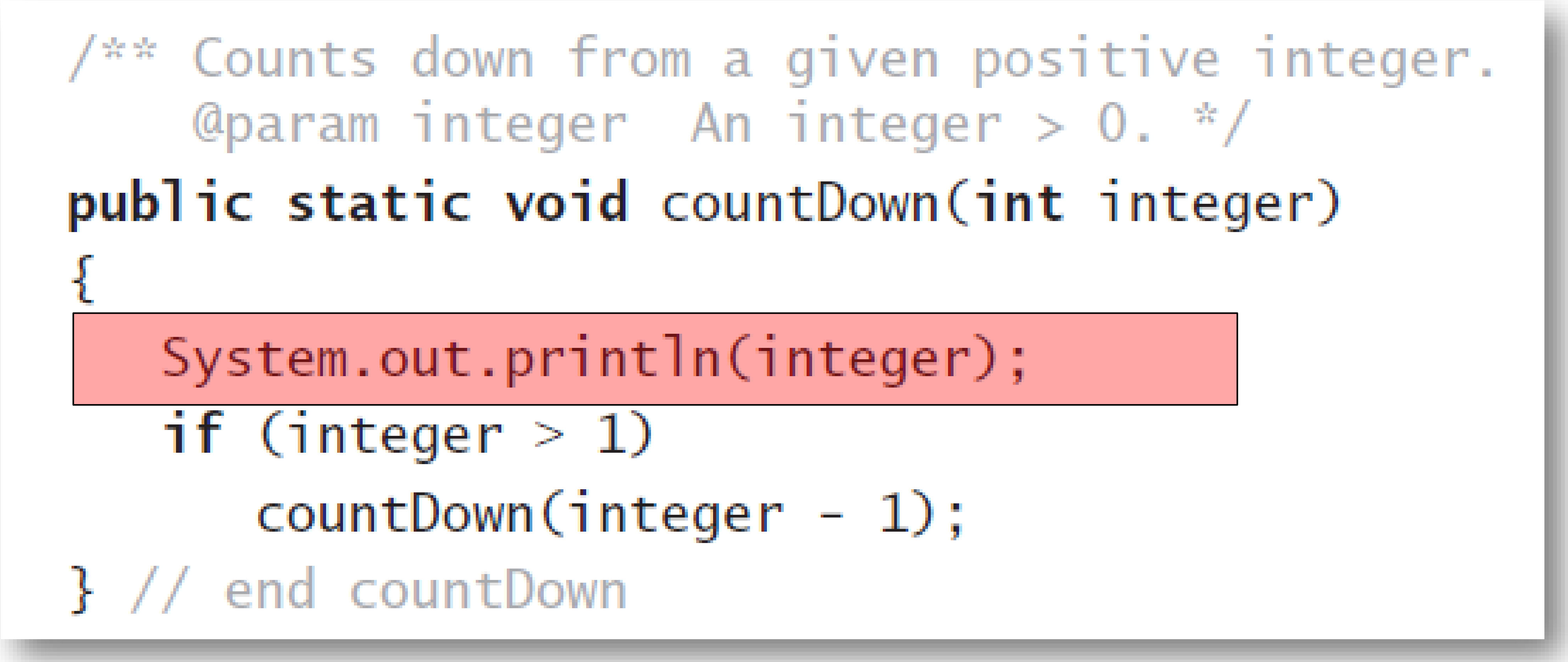
- Recursive Java method to do countdown.



```
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
   System.out.println(integer);
   if (integer > 1)
      countDown(integer - 1);
} // end countDown
```

Example: The Countdown

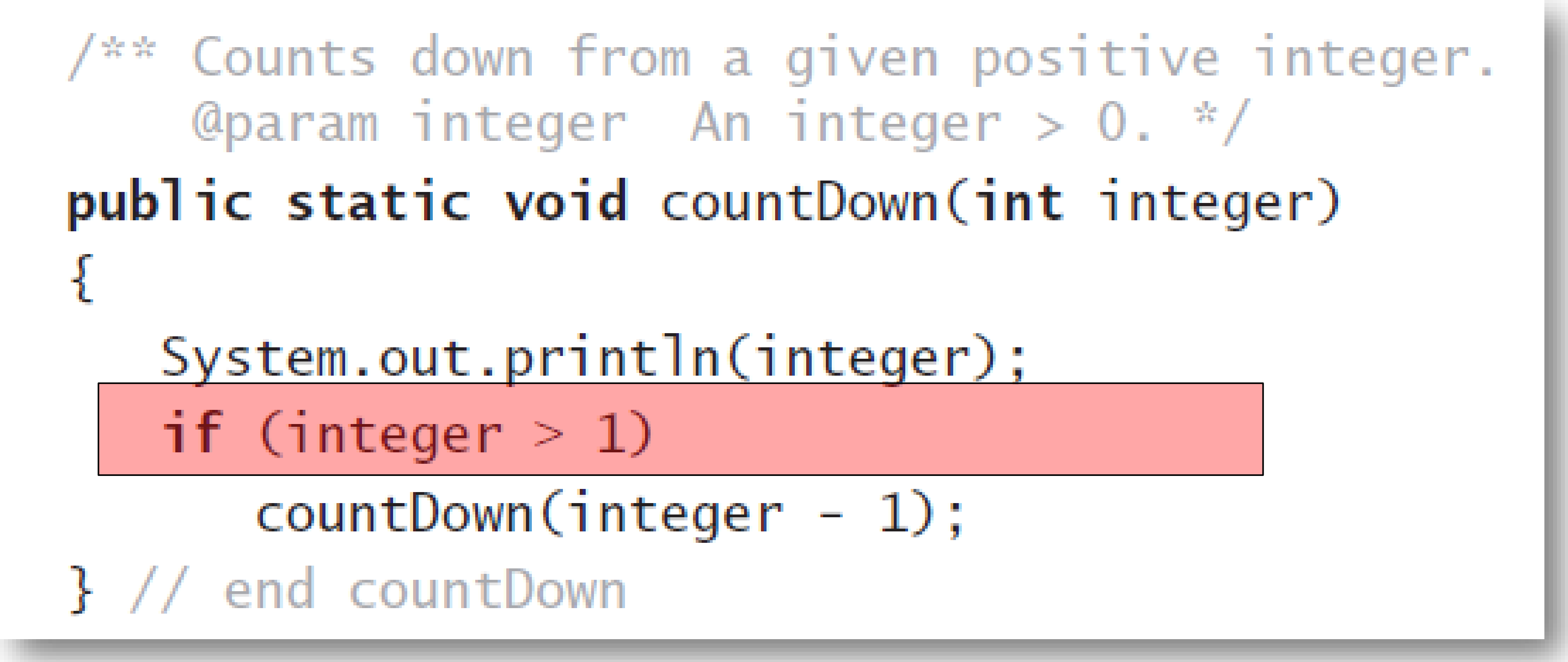
- Each call to the countdown method corresponds to one person
  - what did each person do?
  - say a number



```java
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
   System.out.println(integer);
   if (integer > 1)
      countDown(integer - 1);
} // end countDown
```

Example: The Countdown

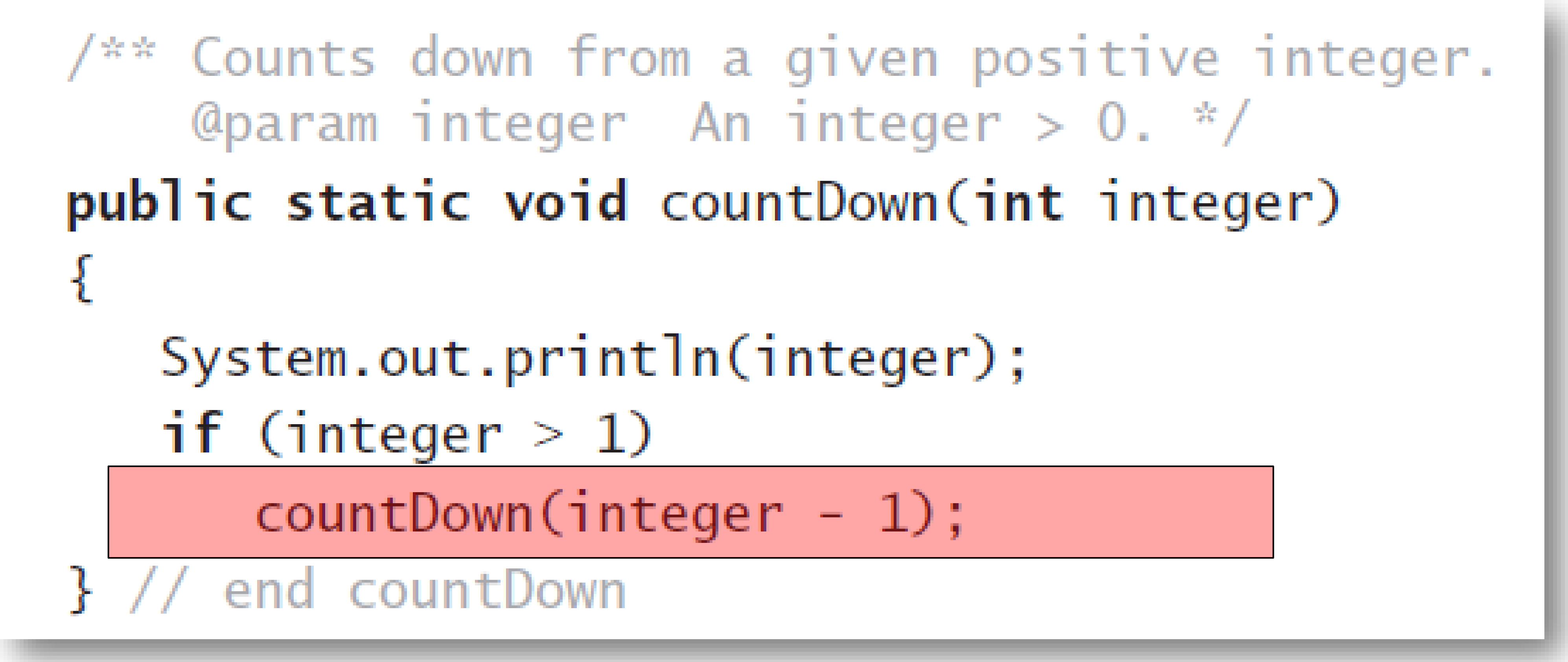
- Each call to the countdown method corresponds to one person
  - what did each person do?
  - say a number
  - check if not done



```
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
   System.out.println(integer);
   if (integer > 1)
      countDown(integer - 1);
} // end countDown
```

Example: The Countdown

- Each call to the countdown method corresponds to one person
  - what did each person do?
  - say a number
  - check if not done
  - ask a classmate to count down starting from the number before



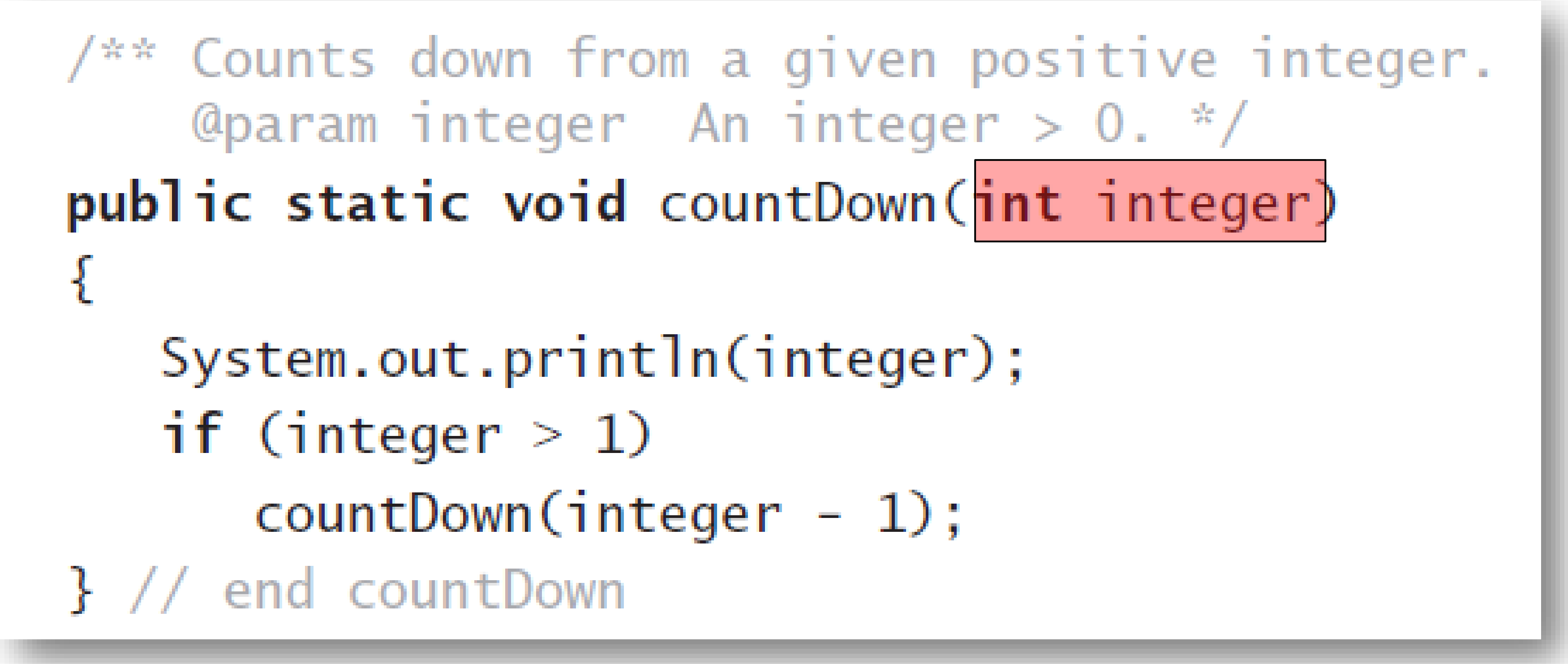
```java
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
   System.out.println(integer);
   if (integer > 1)
      countDown(integer - 1);
} // end countDown
```

Definition

- Recursion is a problem-solving process
  - Breaks a problem into identical but smaller problems.
- A method that calls itself is a **recursive method.**
  - The invocation is a **recursive call** or **recursive invocation**.

Design Guidelines

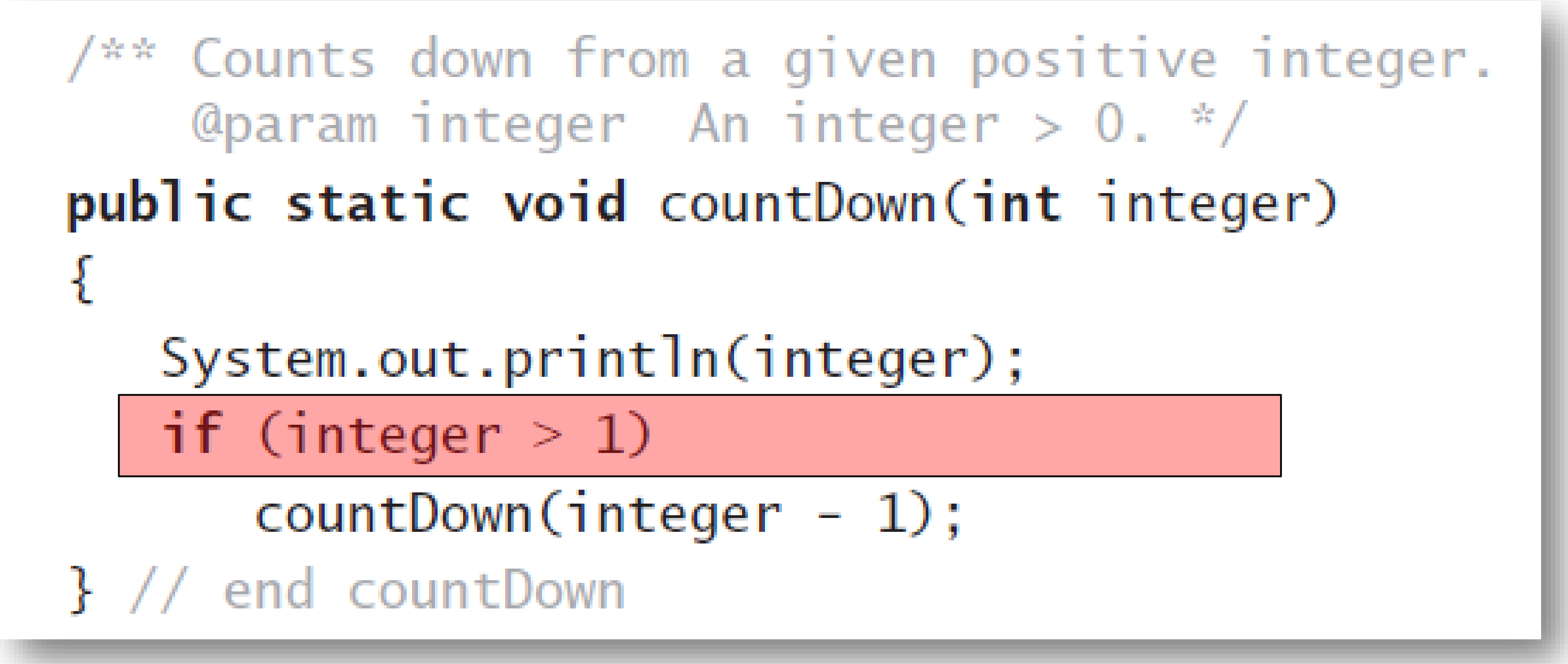
- Method must be given an input value



```
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
   System.out.println(integer);
   if (integer > 1)
      countDown(integer - 1);
} // end countDown
```

Design Guidelines

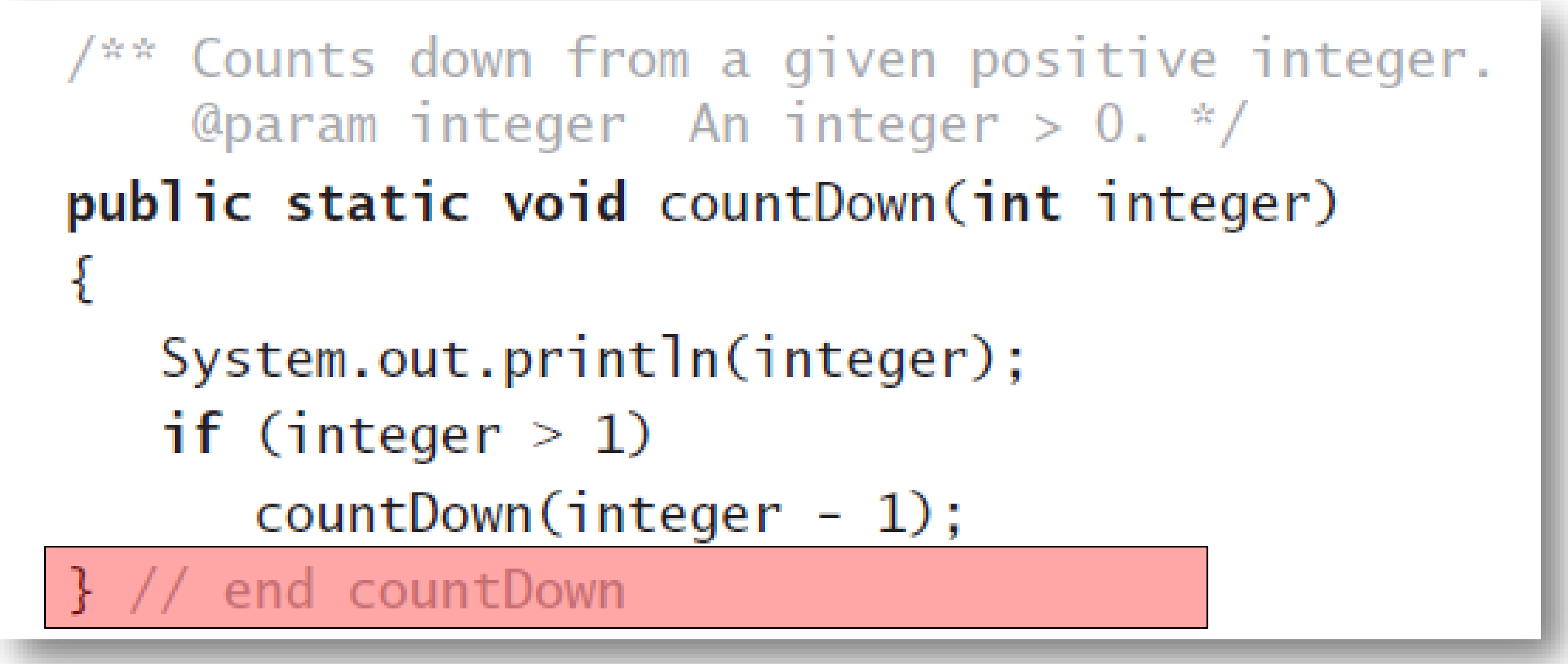
- Method must be given an input value
- Method definition must contain logic that involves this input, leads to different cases



```java
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
   System.out.println(integer);
   if (integer > 1)
      countDown(integer - 1);
} // end countDown
```

Design Guidelines

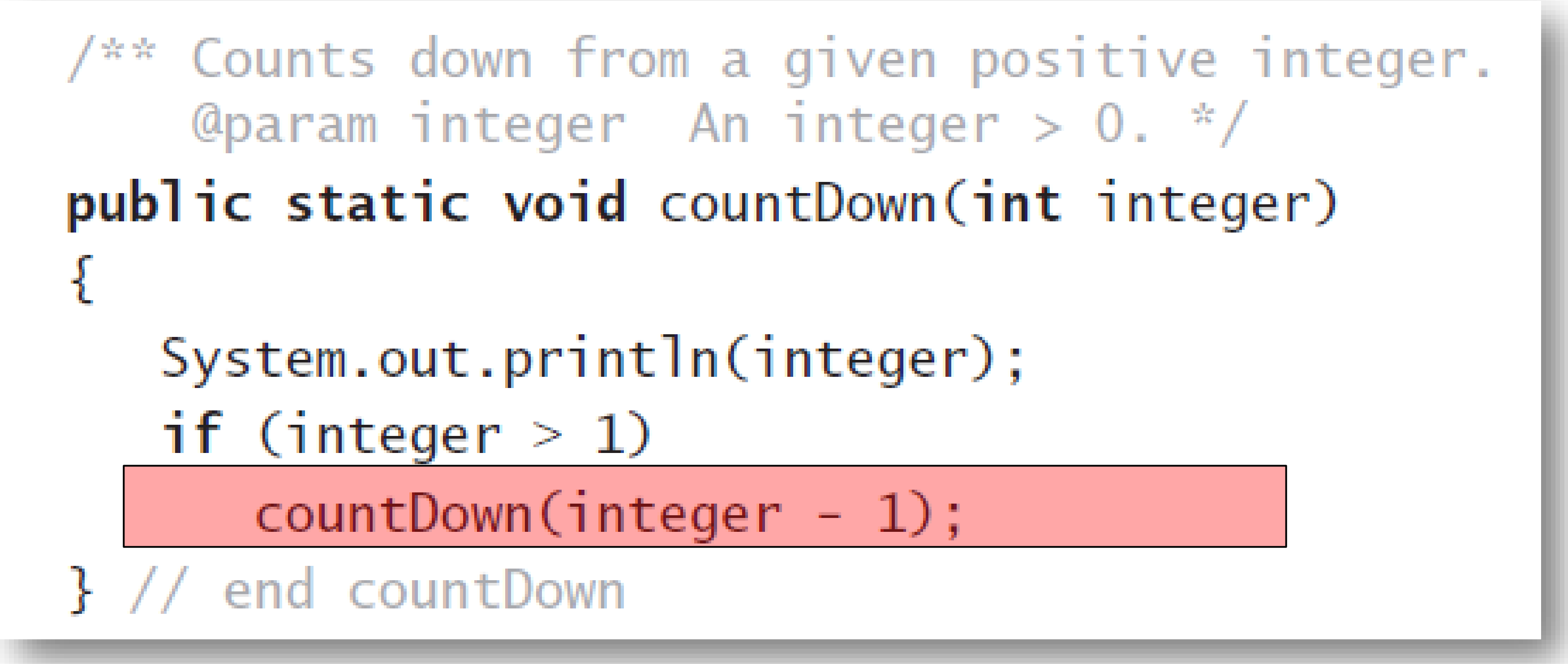
- Method must be given an input value
- Method definition must contain logic that involves this input, leads to different cases
- One or more cases should provide solution that does not require recursion
  - otherwise, infinite recursion
  - if integer <= 1 → return



```
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
   System.out.println(integer);
   if (integer > 1)
      countDown(integer - 1);
} // end countDown
```

Design Guidelines

- Method must be given an input value
- Method definition must contain logic that involves this input, leads to different cases
- One or more cases should provide solution that does not require recursion
- One or more cases must include a recursive invocation



```
/** Counts down from a given positive integer.
    @param integer  An integer > 0. */
public static void countDown(int integer)
{
   System.out.println(integer);
   if (integer > 1)
      countDown(integer - 1);
} // end countDown
```

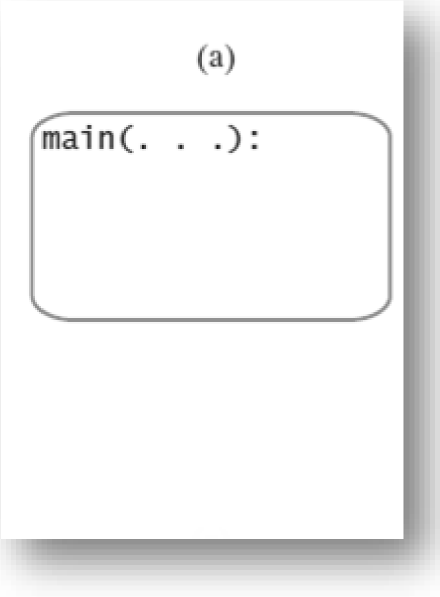
Programming Tip

- Iterative method contains a loop
- Recursive method calls itself
- Some recursive methods contain a loop and call themselves
  - If the recursive method with loop uses **`while`**, make sure you did not mean to use an **`if`** statement

Tracing a Recursive Method

The stack of activation records during the execution of the call **`countDown(3)`**

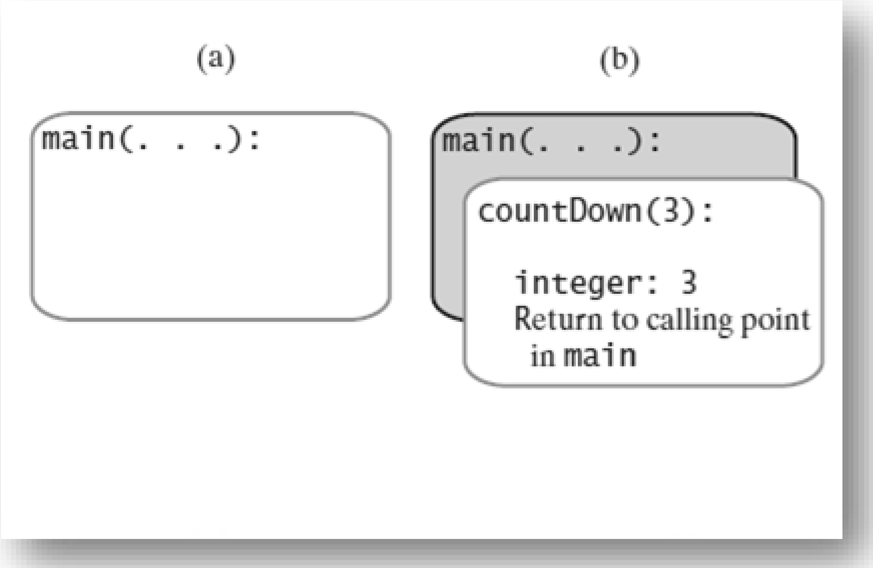
pushing → activation records pile up



Tracing a Recursive Method

The stack of activation records during the execution of the call `countDown(3)`

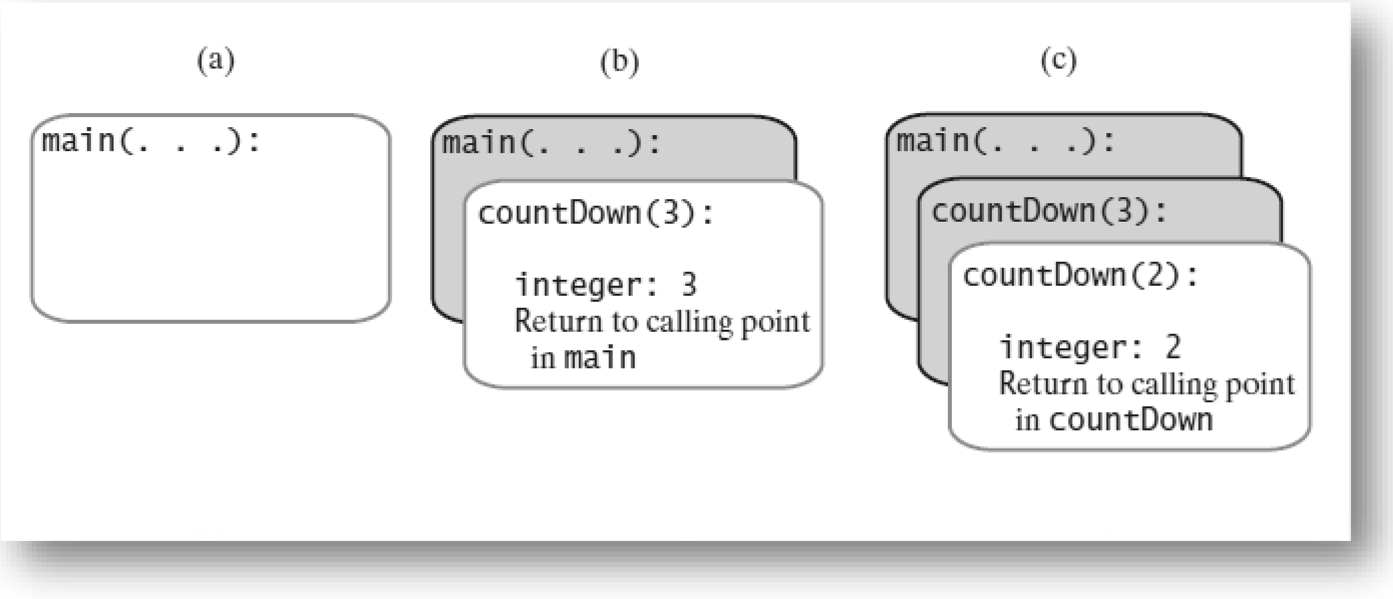
pushing → activation records pile up



Tracing a Recursive Method

The stack of activation records during the execution of the call **`countDown(3)`**

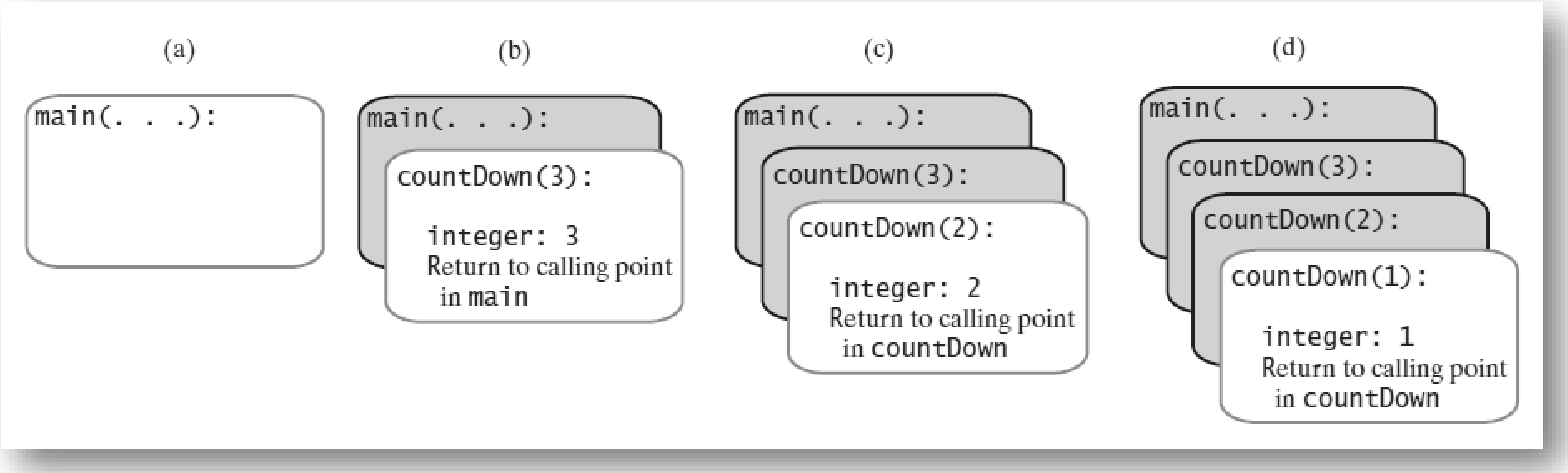
pushing → activation records pile up



Tracing a Recursive Method

The stack of activation records during the execution of the call `countDown(3)`

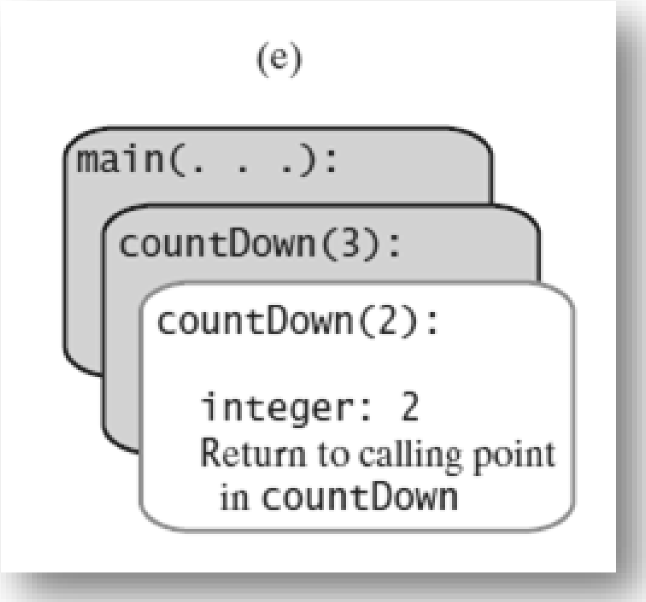
pushing → activation records pile up



Tracing a Recursive Method

The stack of activation records during the execution of the call **`countDown(3)`**

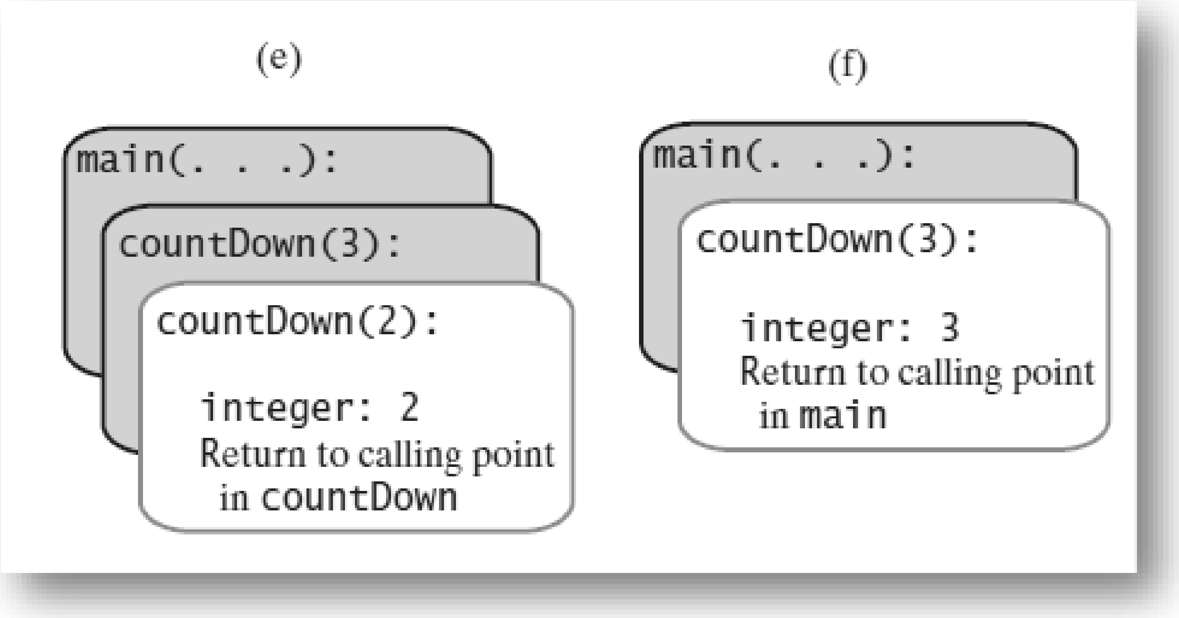
popping → activation records tear down



Tracing a Recursive Method

The stack of activation records during the execution of the call **`countDown(3)`**

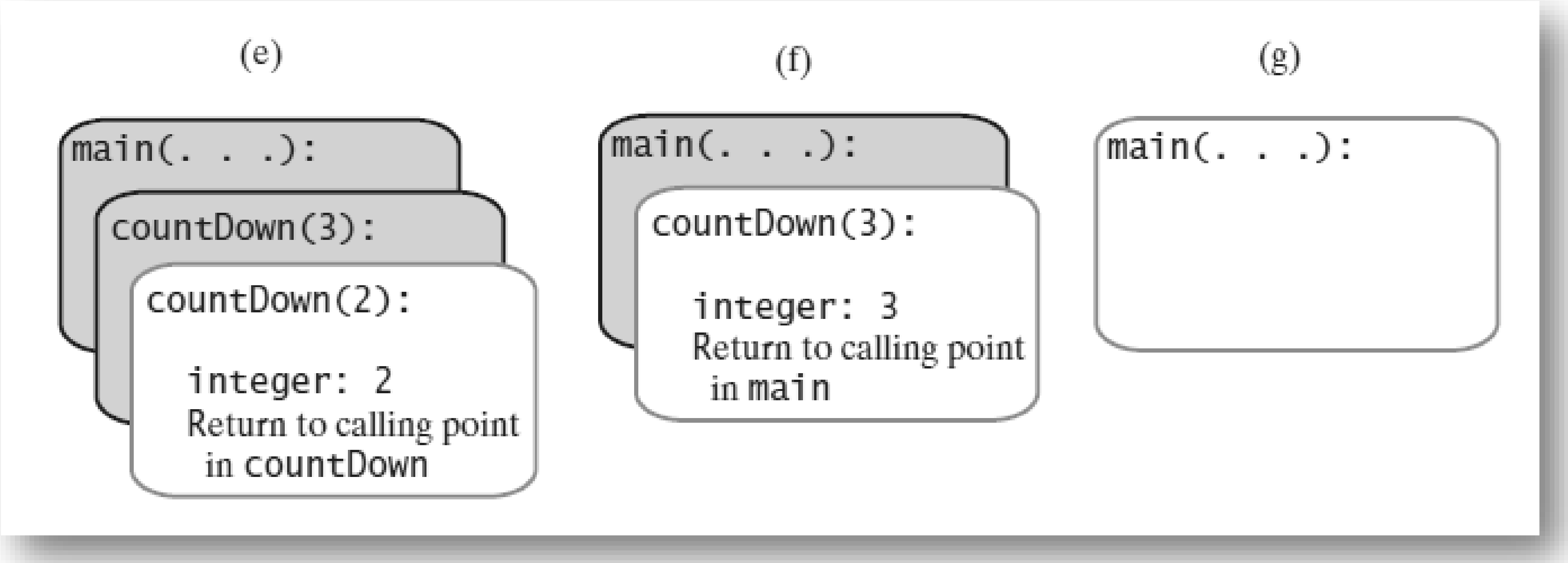
popping → activation records tear down



Tracing a Recursive Method

The stack of activation records during the execution of the call **`countDown(3)`**

popping → activation records tear down



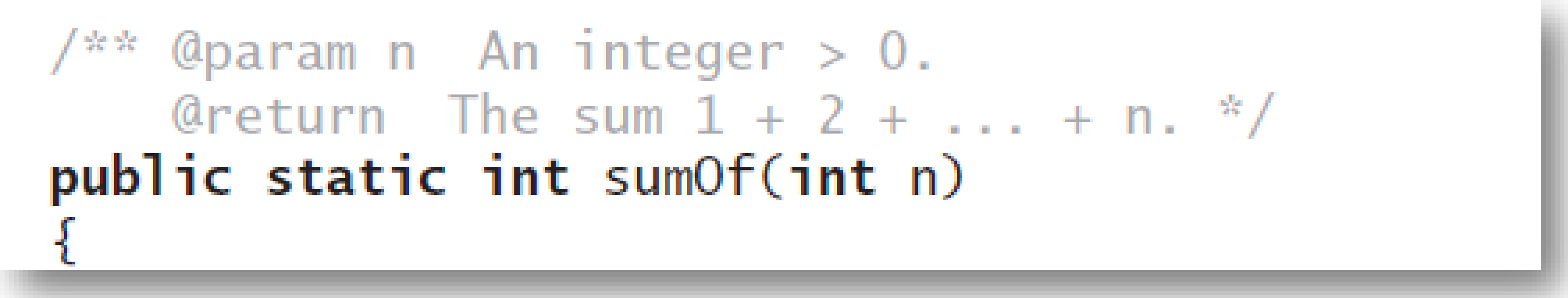
Stack of Activation Records

- Each call to a method generates an activation record
- Recursive method uses more memory than an iterative method
  - Each recursive call generates an activation record
- If recursive call generates too many activation records, could cause stack overflow

Recursive Methods That Return a Value

- Recursive method to calculate

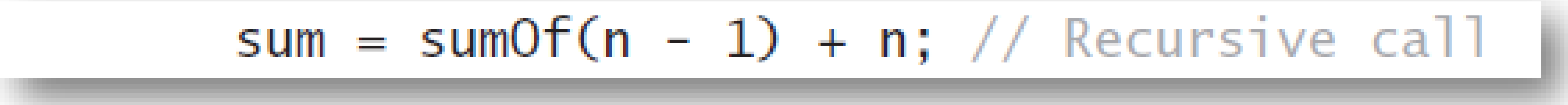
$$\sum_{i=1}^{n} i$$



```
/** @param n  An integer > 0.
    @return  The sum 1 + 2 + ... + n. */
public static int sumOf(int n)
{
```

Recursive Methods That Return a Value

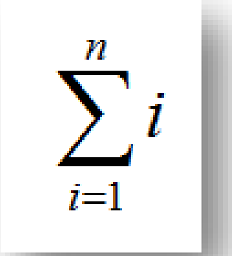
- Recursive method to calculate $\sum_{i=1}^{n} i$



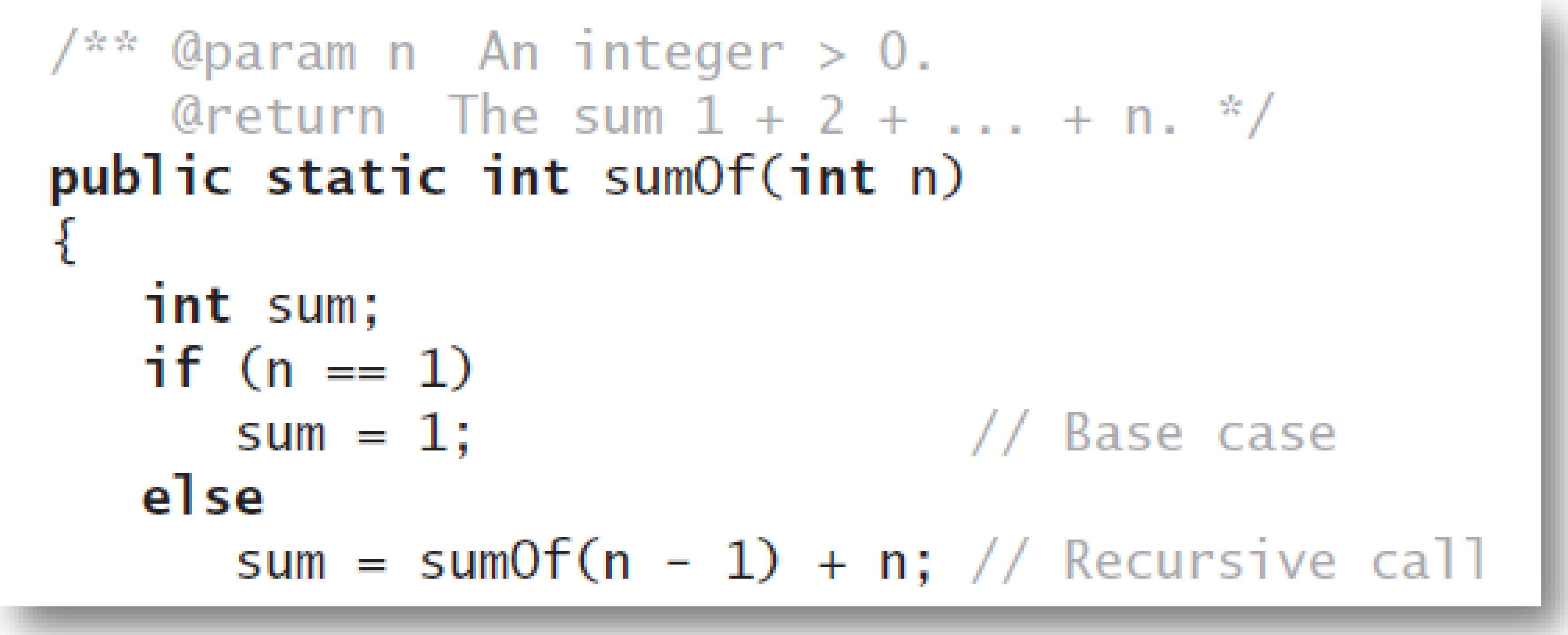
```
        sum = sumOf(n - 1) + n; // Recursive call
```

Recursive Methods That Return a Value

- Recursive method to calculate



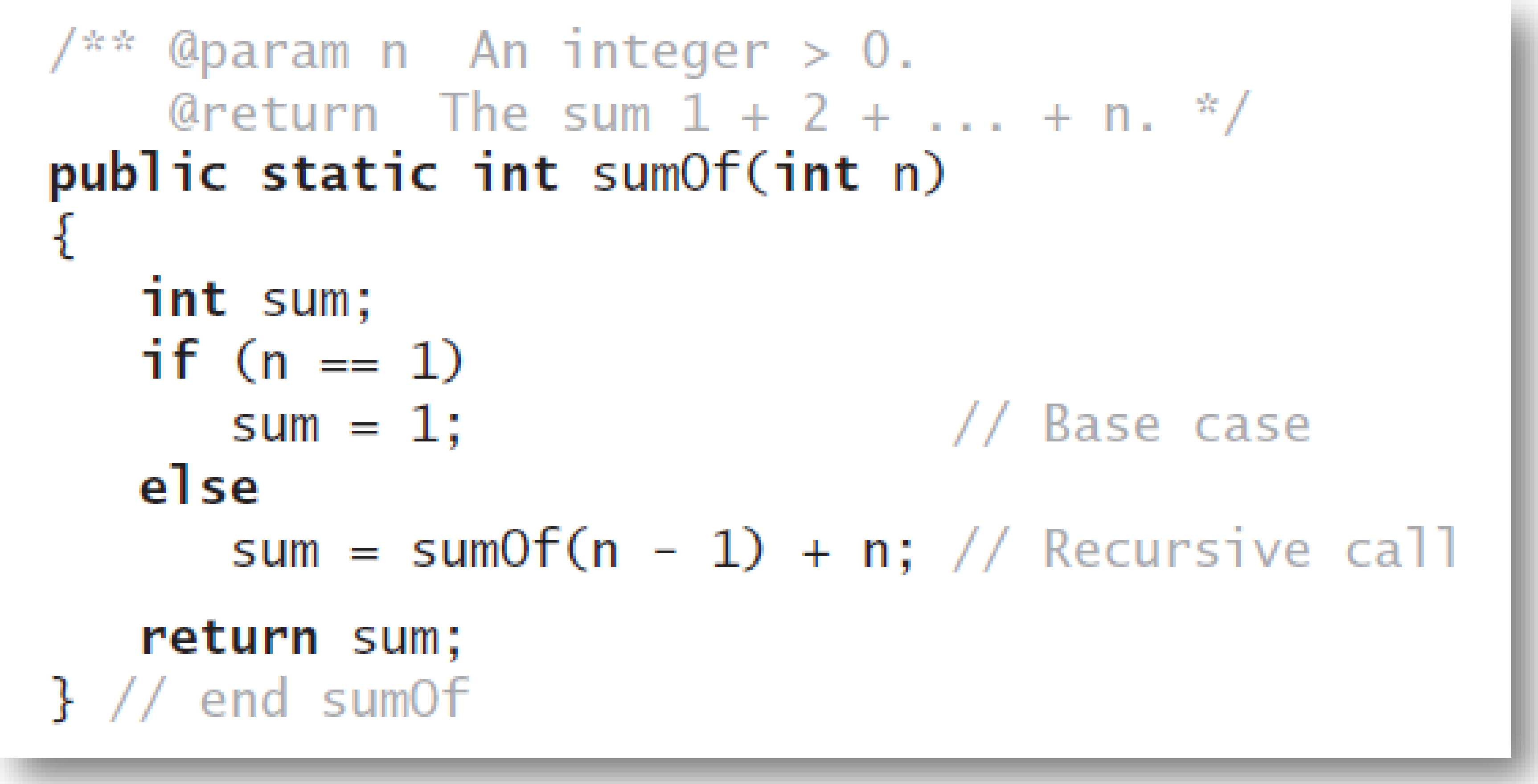
$$\sum_{i=1}^{n} i$$



```
/** @param n  An integer > 0.
    @return  The sum 1 + 2 + ... + n. */
public static int sumOf(int n)
{
   int sum;
   if (n == 1)
      sum = 1;                  // Base case
   else
      sum = sumOf(n - 1) + n; // Recursive call
```

Recursive Methods That Return a Value

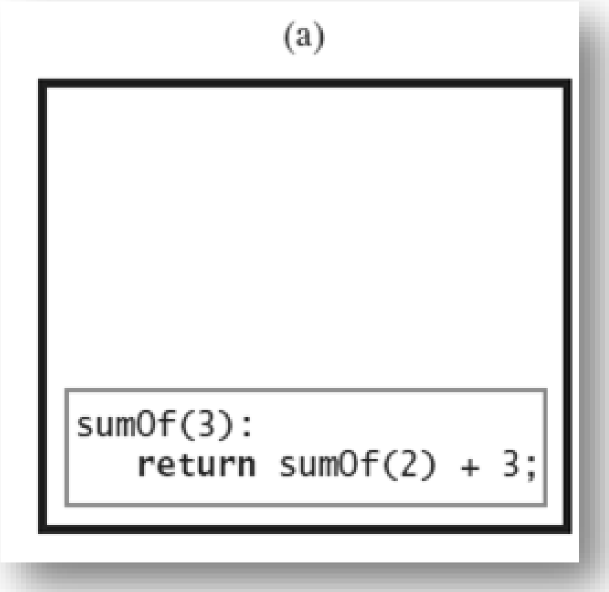
- Recursive method to calculate $\sum_{i=1}^{n} i$



```
/** @param n  An integer > 0.
    @return  The sum 1 + 2 + . . . + n. */
public static int sumOf(int n)
{
   int sum;
   if (n == 1)
      sum = 1;                // Base case
   else
      sum = sumOf(n - 1) + n; // Recursive call

   return sum;
} // end sumOf
```

Tracing a Recursive Method

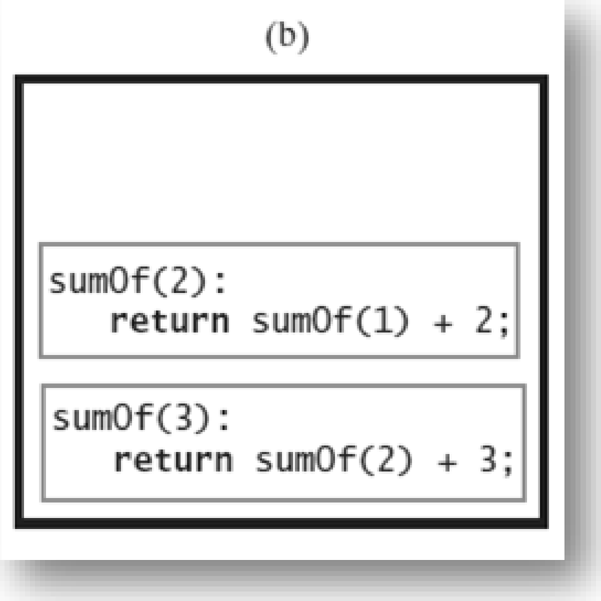
Tracing the execution of `sumOf(3)`



```
sumOf(3):
   return sumOf(2) + 3;
```

Tracing a Recursive Method

Tracing the execution of `sumOf(3)`



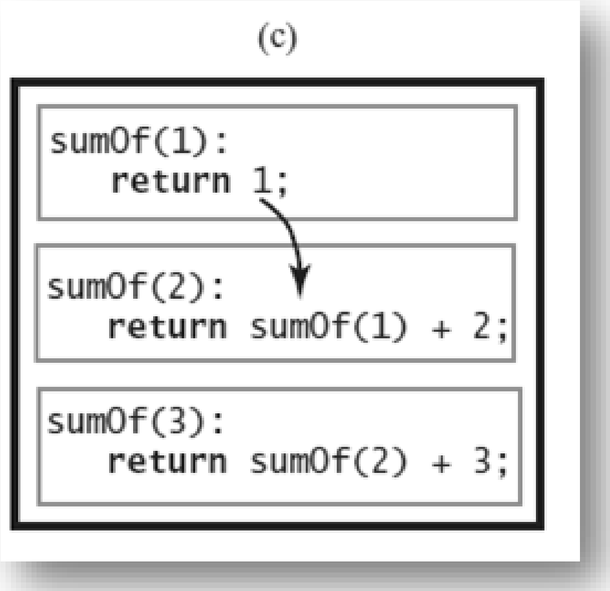
(b)

```
sumOf(2):
   return sumOf(1) + 2;
```

```
sumOf(3):
   return sumOf(2) + 3;
```

Tracing a Recursive Method

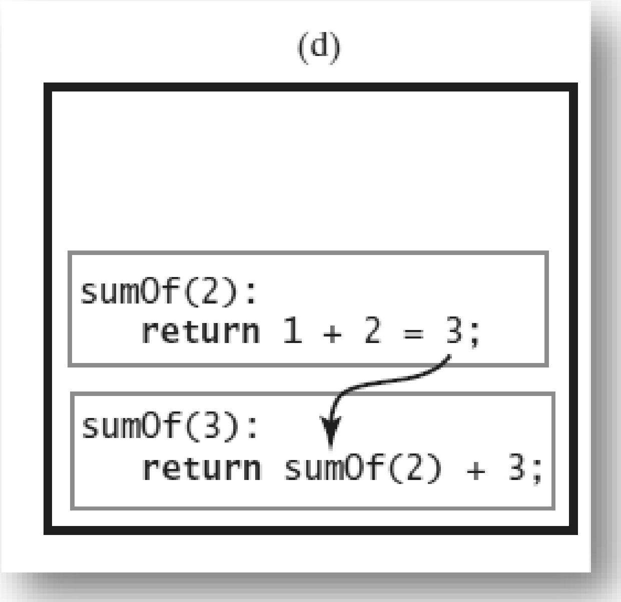
Tracing the execution of `sumOf(3)`



```
sumOf(1):
   return 1;

sumOf(2):
   return sumOf(1) + 2;

sumOf(3):
   return sumOf(2) + 3;
```

Tracing a Recursive Method

Tracing the execution of `sumOf(3)`

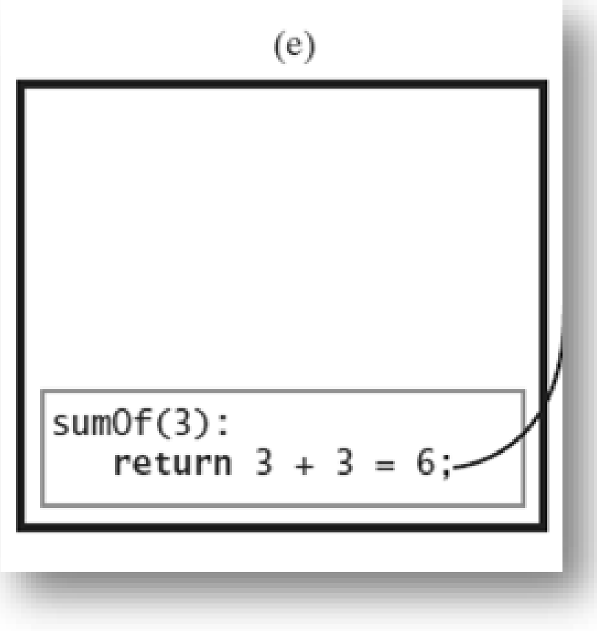


```
sumOf(2):
   return 1 + 2 = 3;
```

```
sumOf(3):
   return sumOf(2) + 3;
```

Tracing a Recursive Method

Tracing the execution of `sumOf(3)`

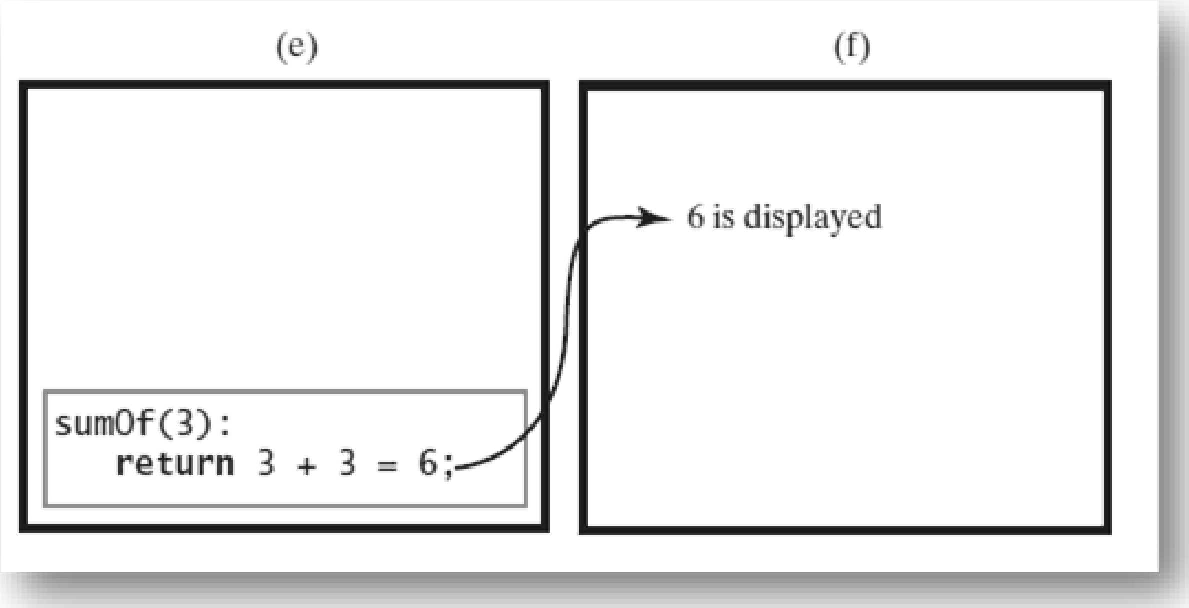


(e)

```
sumOf(3):
   return 3 + 3 = 6;
```

Tracing a Recursive Method

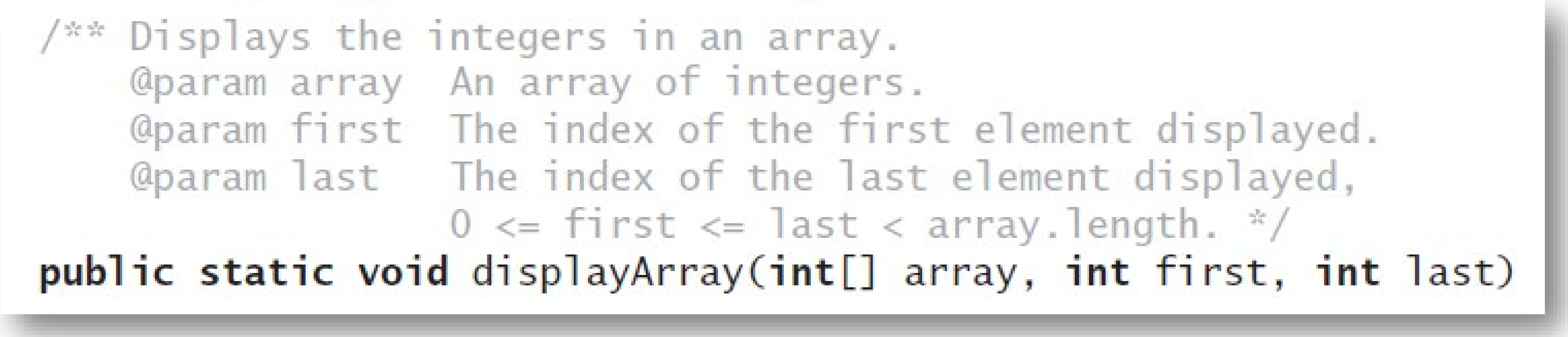
Tracing the execution of `sumOf(3)`



```
sumOf(3):
   return 3 + 3 = 6;
```

Recursively Processing an Array

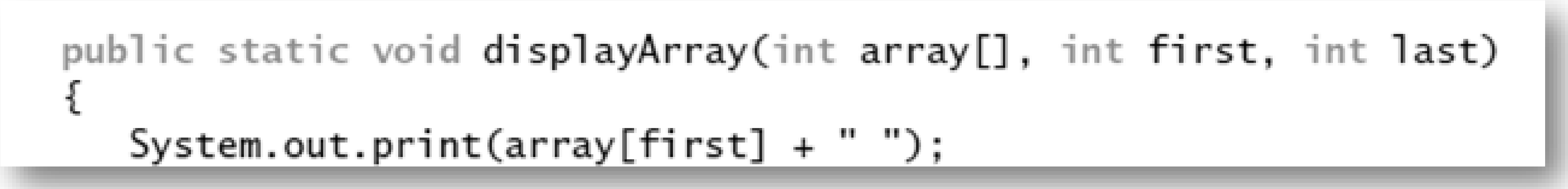
A recursive method to display array.



```
/** Displays the integers in an array.
    @param array  An array of integers.
    @param first  The index of the first element displayed.
    @param last   The index of the last element displayed,
                  0 <= first <= last < array.length. */
public static void displayArray(int[] array, int first, int last)
```

Recursively Processing an Array

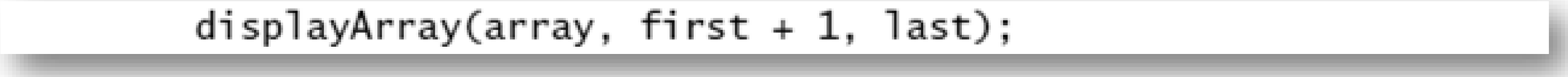
Starting with `array[first]`



```
public static void displayArray(int array[], int first, int last)
{
   System.out.print(array[first] + " ");
```

Recursively Processing an Array

Starting with `array[first]`

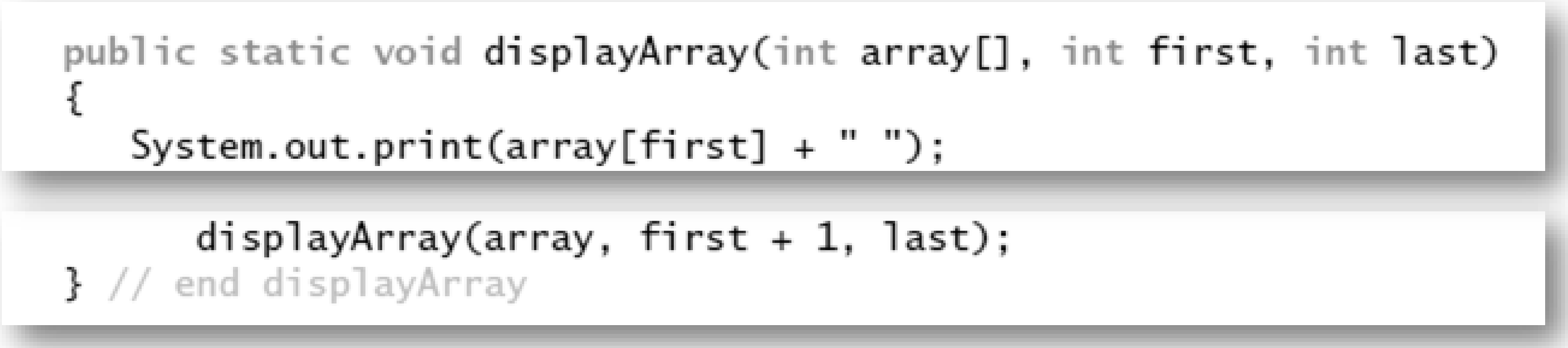


```
        displayArray(array, first + 1, last);
```

Recursively Processing an Array

Starting with `array[first]`

What is wrong with this method?



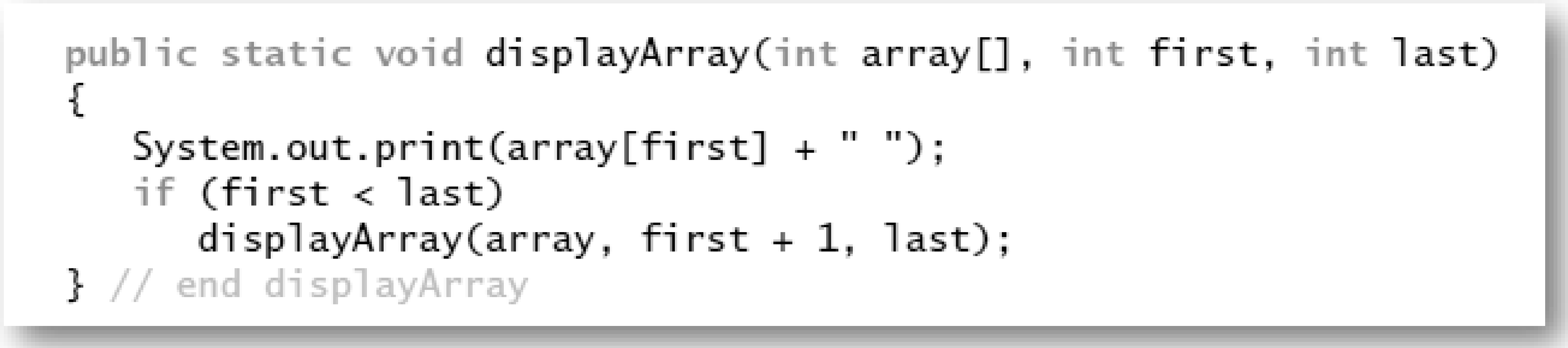
```
public static void displayArray(int array[], int first, int last)
{
   System.out.print(array[first] + " ");

      displayArray(array, first + 1, last);
} // end displayArray
```

Recursively Processing an Array

Starting with `array[first]`

We need a base (non-recursive) case!

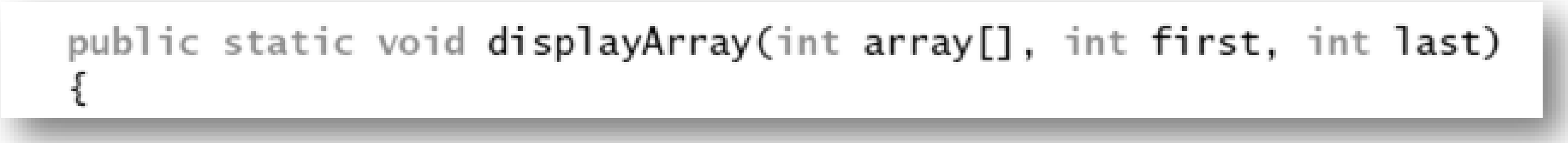
- ask for help only when there is at least one array entry to display
- otherwise, return



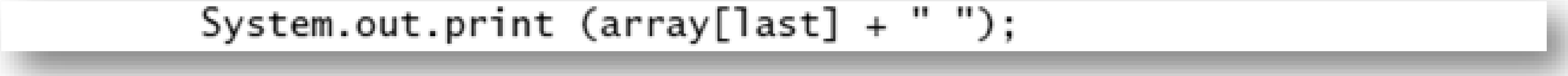
```
public static void displayArray(int array[], int first, int last)
{
   System.out.print(array[first] + " ");
   if (first < last)
      displayArray(array, first + 1, last);
} // end displayArray
```

Recursively Processing an Array

Alternatively, …



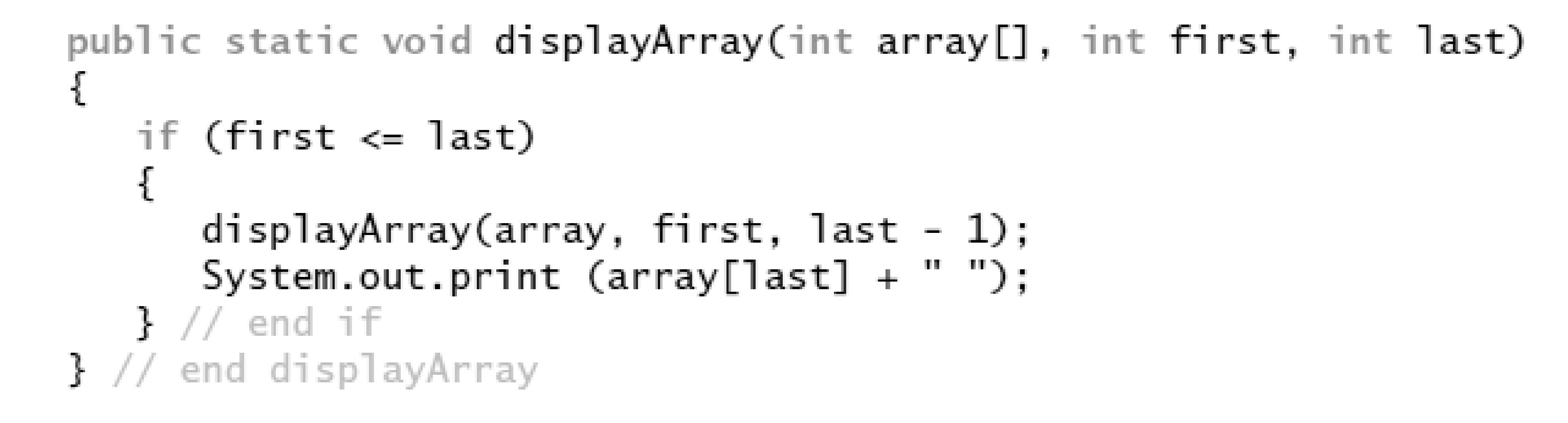
```
public static void displayArray(int array[], int first, int last)
{
```



```
      System.out.print (array[last] + " ");
```

Recursively Processing an Array

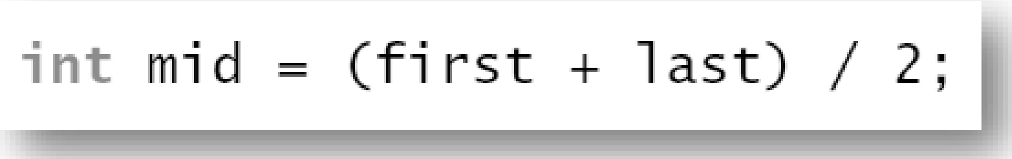
Alternatively, …



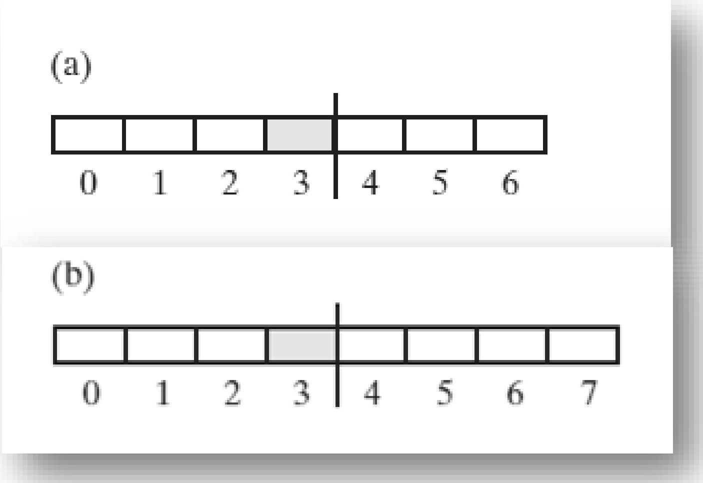
```
public static void displayArray(int array[], int first, int last)
{
   if (first <= last)
   {
      displayArray(array, first, last - 1);
      System.out.print (array[last] + " ");
   } // end if
} // end displayArray
```

Recursively Processing an Array

How can we find the middle entry given first and last?

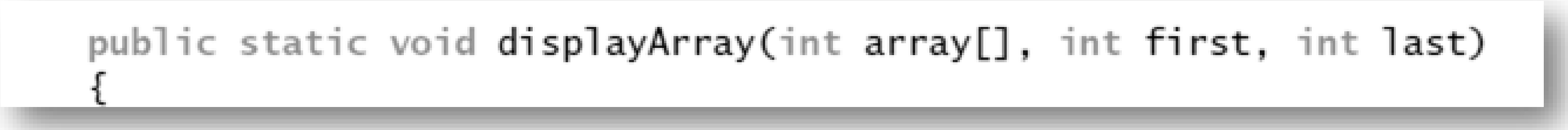


```
int mid = (first + last) / 2;
```

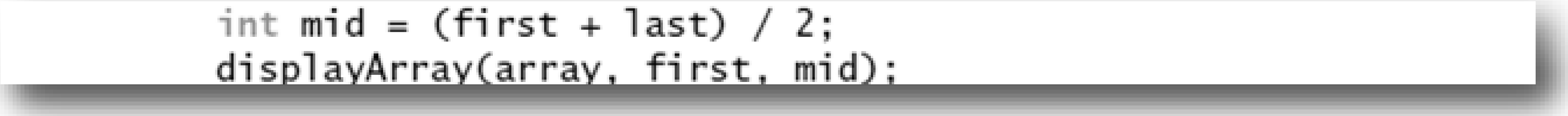


Recursively Processing an Array

- Processing array from middle.



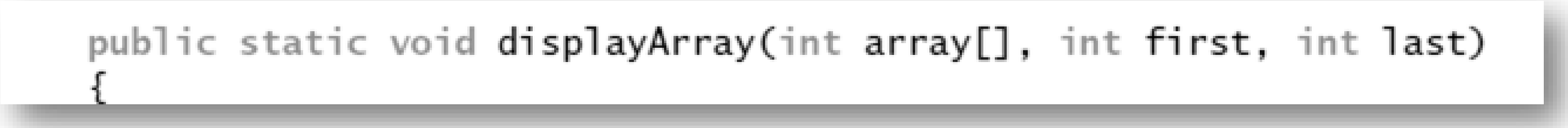
```
public static void displayArray(int array[], int first, int last)
{
```



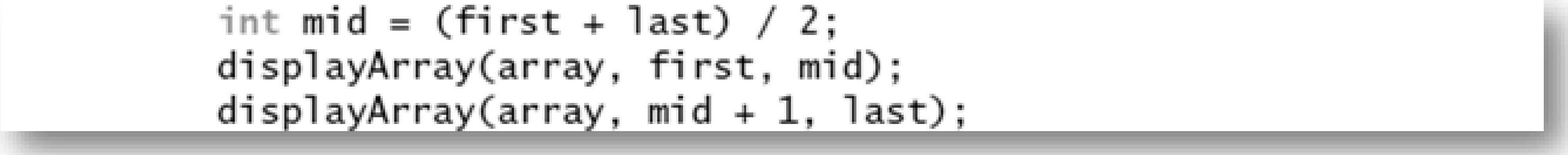
```
        int mid = (first + last) / 2;
        displayArray(array, first, mid);
```

Recursively Processing an Array

- Processing array from middle.



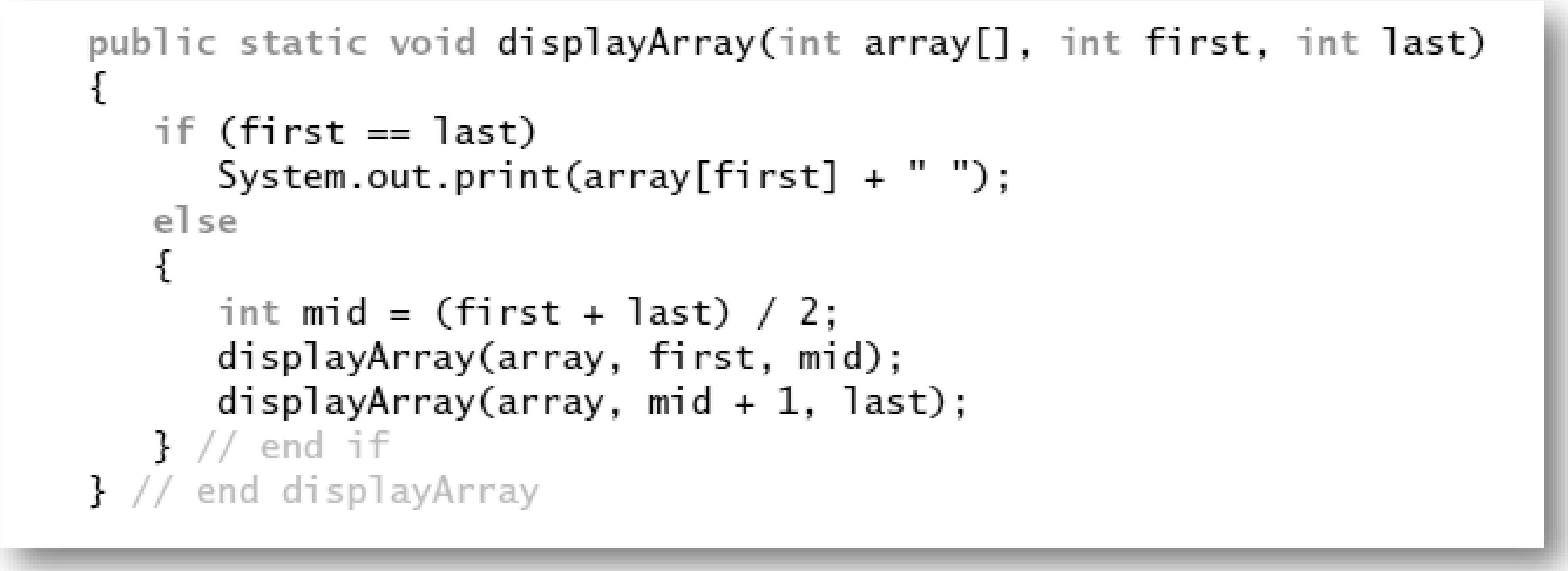
```
public static void displayArray(int array[], int first, int last)
{
```



```
      int mid = (first + last) / 2;
      displayArray(array, first, mid);
      displayArray(array, mid + 1, last);
```

Recursively Processing an Array

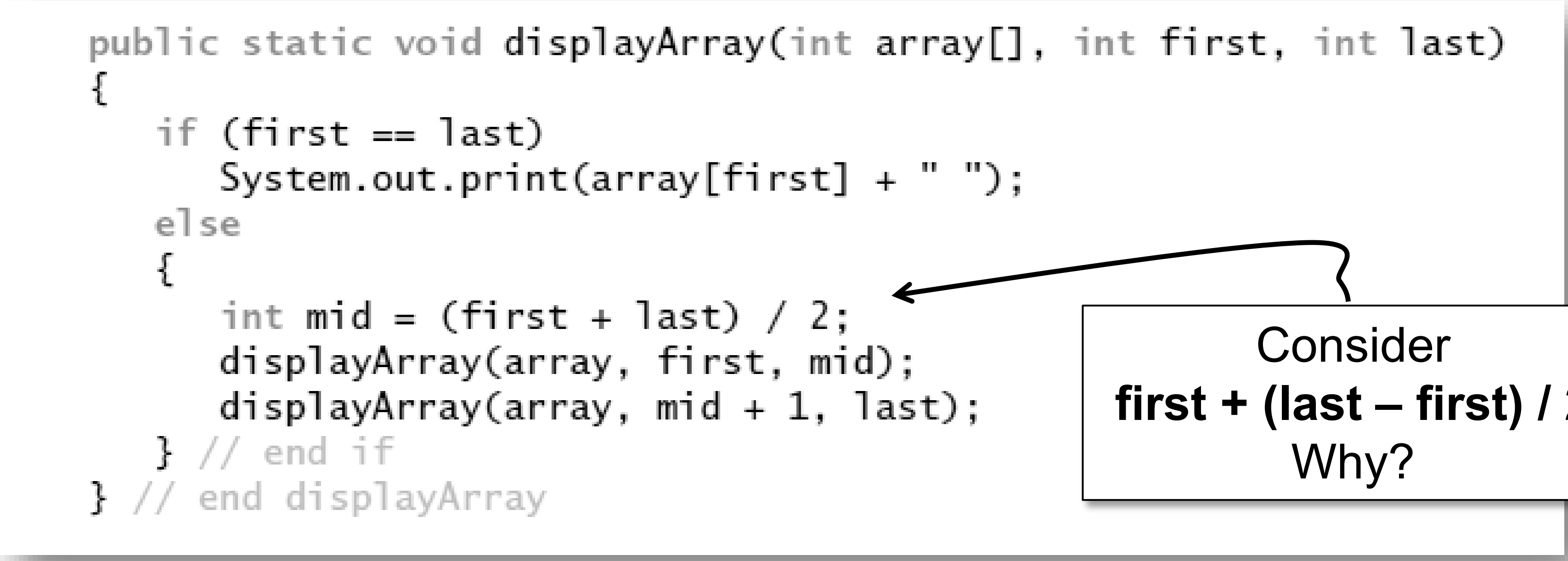
- Processing array from middle.



```
public static void displayArray(int array[], int first, int last)
{
   if (first == last)
      System.out.print(array[first] + " ");
   else
   {
      int mid = (first + last) / 2;
      displayArray(array, first, mid);
      displayArray(array, mid + 1, last);
   } // end if
} // end displayArray
```

Recursively Processing an Array

- Processing array from middle.

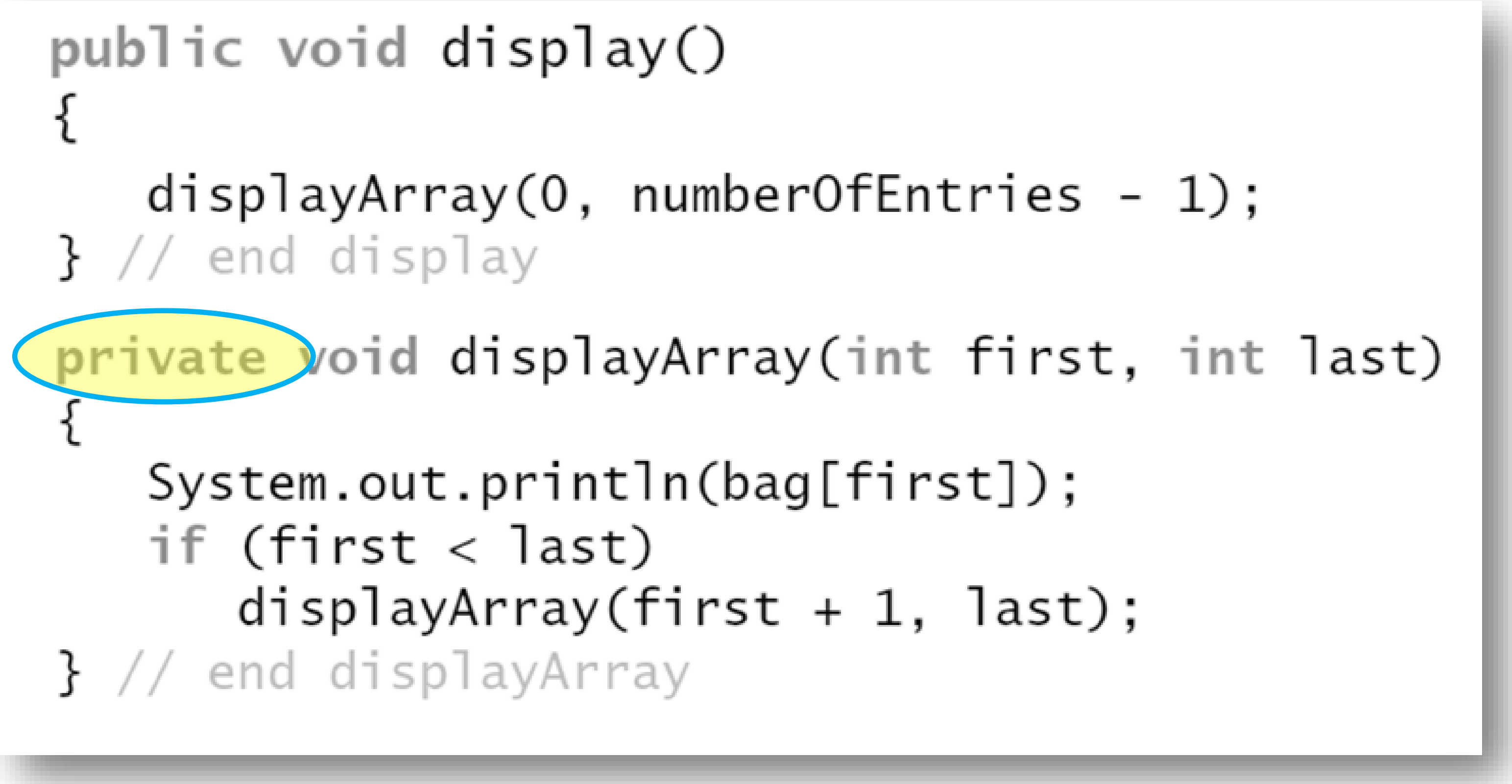


```
public static void displayArray(int array[], int first, int last)
{
   if (first == last)
      System.out.print(array[first] + " ");
   else
   {
      int mid = (first + last) / 2;
      displayArray(array, first, mid);
      displayArray(array, mid + 1, last);
   } // end if
} // end displayArray
```

Consider
first + (last – first) / 2
Why?

Displaying a Bag

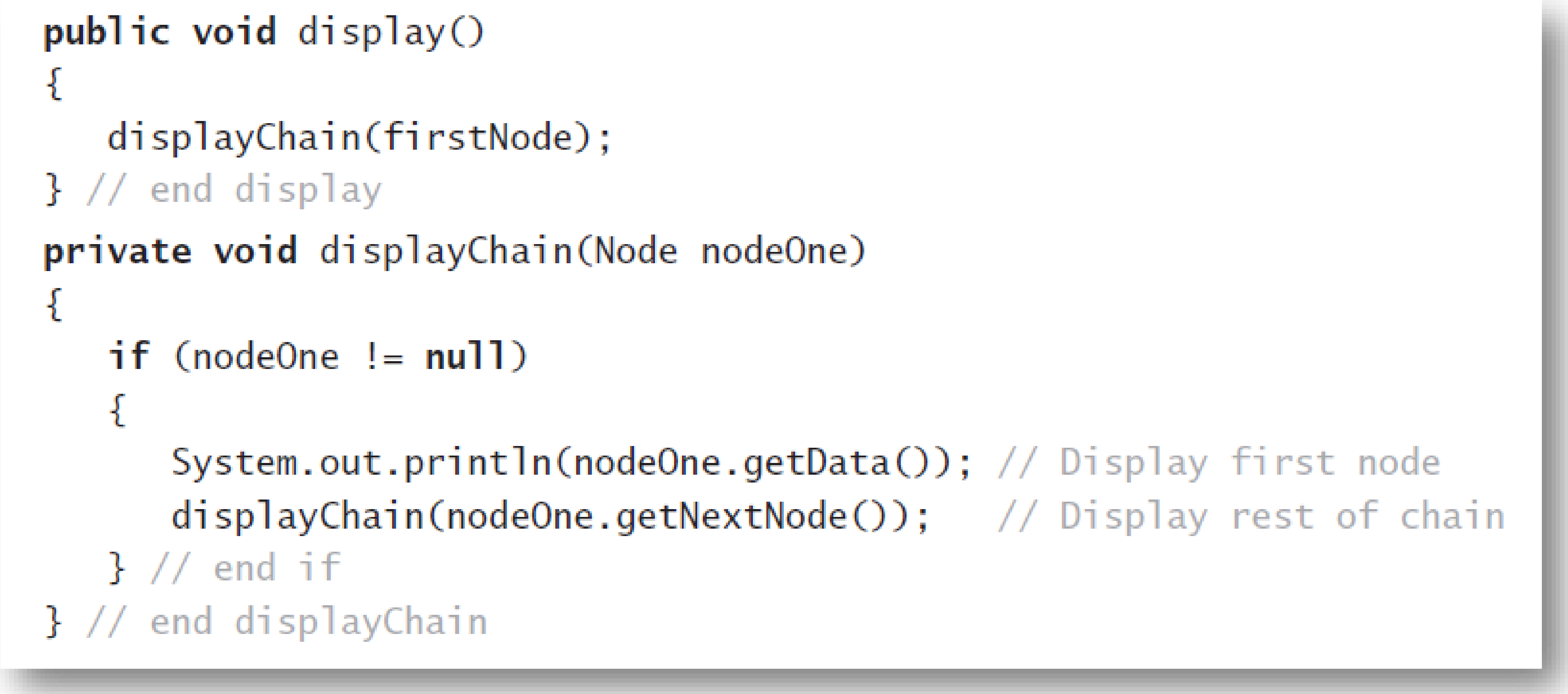
- Recursive method that is part of an implementation of an ADT is private



```
public void display()
{
   displayArray(0, numberOfEntries - 1);
} // end display

private void displayArray(int first, int last)
{
   System.out.println(bag[first]);
   if (first < last)
      displayArray(first + 1, last);
} // end displayArray
```

Recursively Processing a Linked Chain

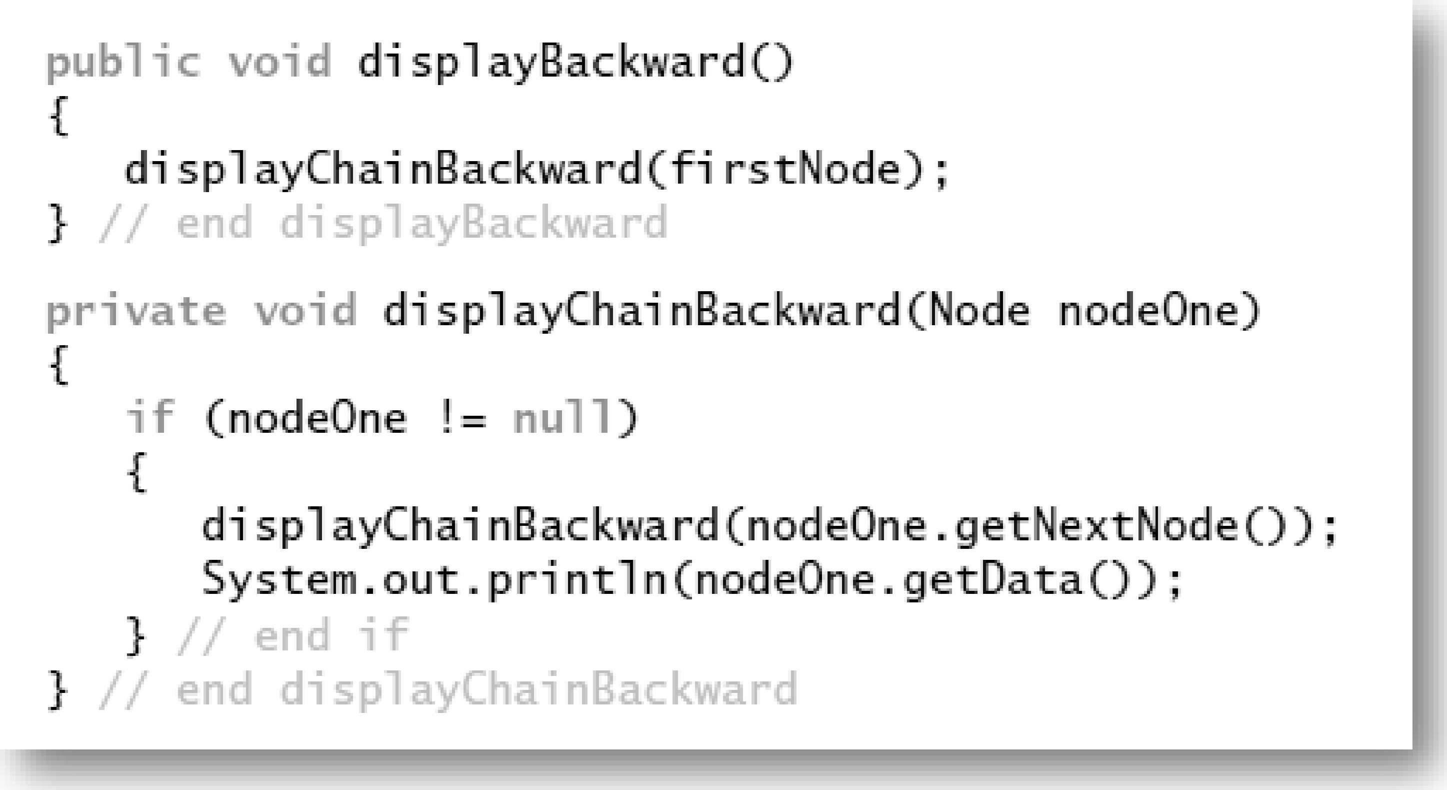
- Display data in first node and recursively display data in rest of chain.



```
public void display()
{
   displayChain(firstNode);
} // end display
private void displayChain(Node nodeOne)
{
   if (nodeOne != null)
   {
      System.out.println(nodeOne.getData()); // Display first node
      displayChain(nodeOne.getNextNode());   // Display rest of chain
   } // end if
} // end displayChain
```

Recursively Processing a Linked Chain

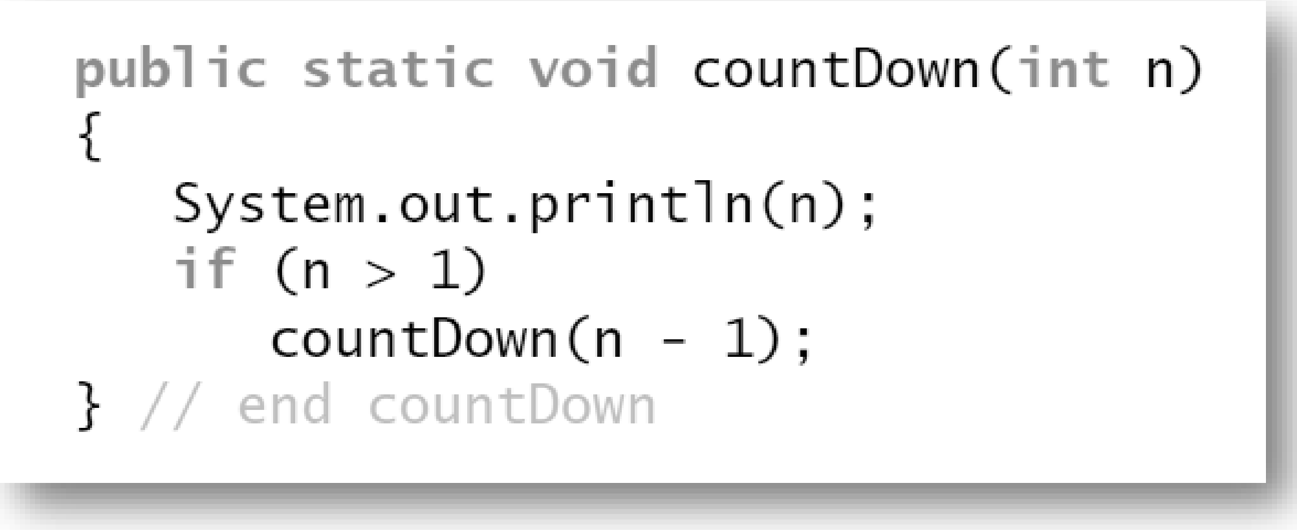
- Displaying a chain backwards. Traversing chain of linked nodes in reverse order easier when done recursively.



```
public void displayBackward()
{
   displayChainBackward(firstNode);
} // end displayBackward

private void displayChainBackward(Node nodeOne)
{
   if (nodeOne != null)
   {
      displayChainBackward(nodeOne.getNextNode());
      System.out.println(nodeOne.getData());
   } // end if
} // end displayChainBackward
```

Recursively Processing a Linked Chain

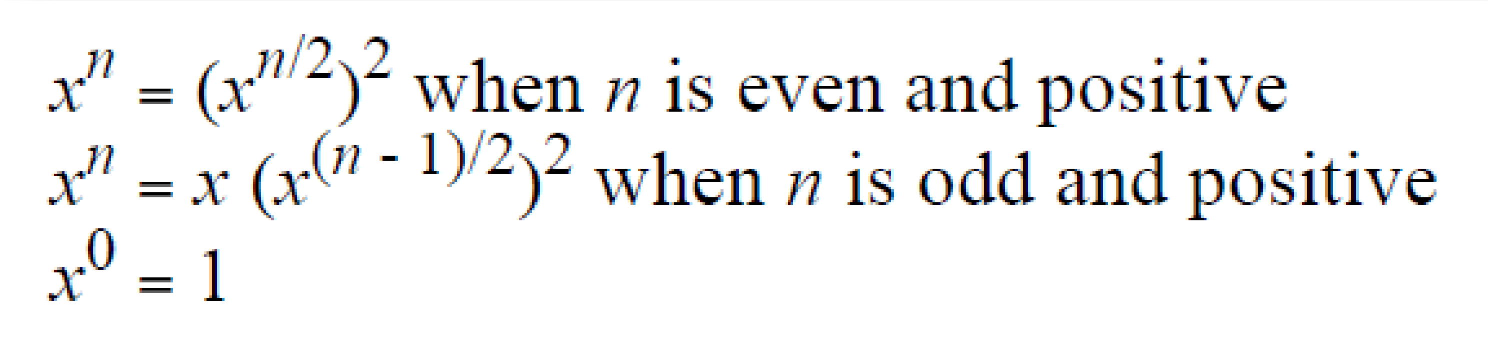
- Using proof by induction, we conclude method is *O(n).*



```
public static void countDown(int n)
{
   System.out.println(n);
   if (n > 1)
      countDown(n - 1);
} // end countDown
```

Time Efficiency of Computing x^n

- Efficiency of algorithm is *O(log n)*



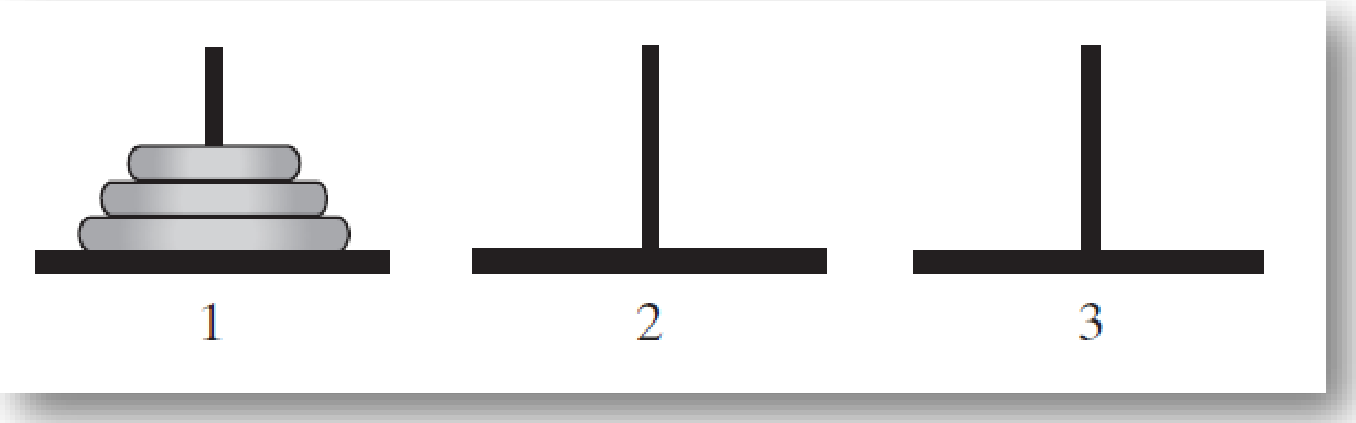
$x^n = (x^{n/2})^2$ when n is even and positive

$x^n = x\ (x^{(n - 1)/2})^2$ when n is odd and positive

$x^0 = 1$

Simple Solution to a Difficult Problem

The initial configuration of the Towers of Hanoi for three disks.



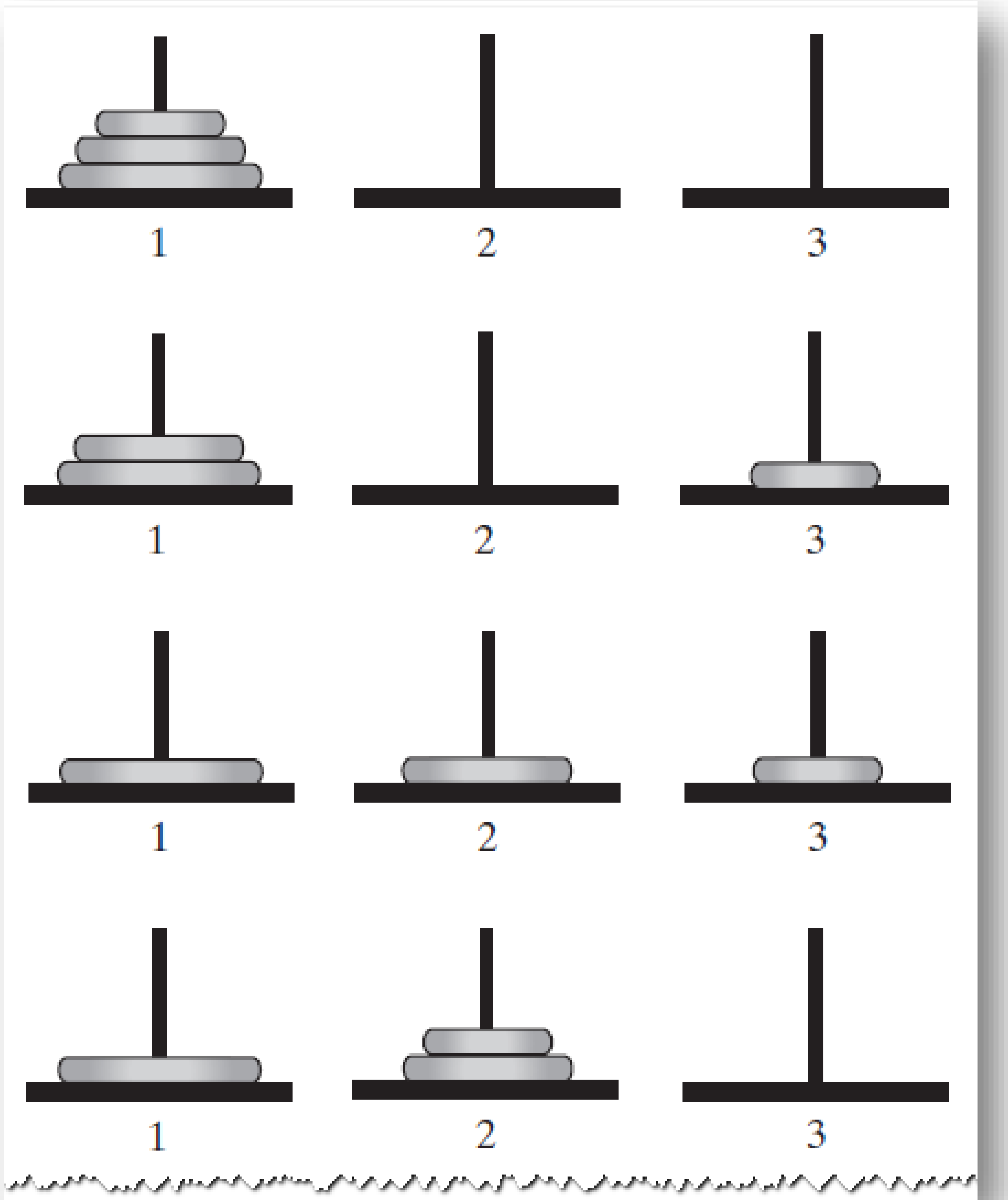
Simple Solution to a Difficult Problem

- Rules:

1. Move one disk at a time. Each disk moved must be topmost disk.
2. No disk may rest on top of a disk smaller than itself.
3. You can store disks on the second pole temporarily, as long as you observe the previous two rules.

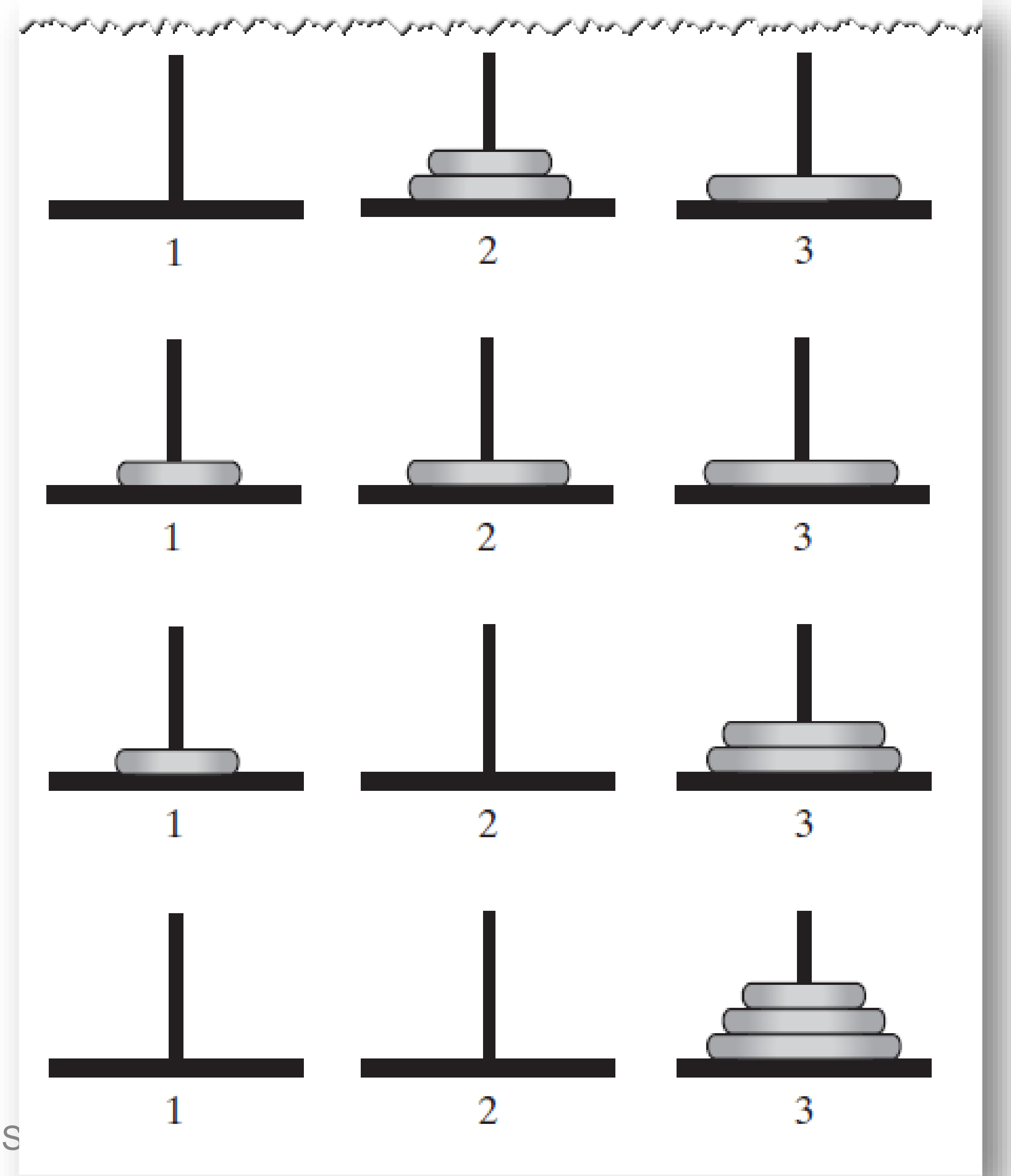
Solutions

The sequence of moves for solving the Towers of Hanoi problem with three disks



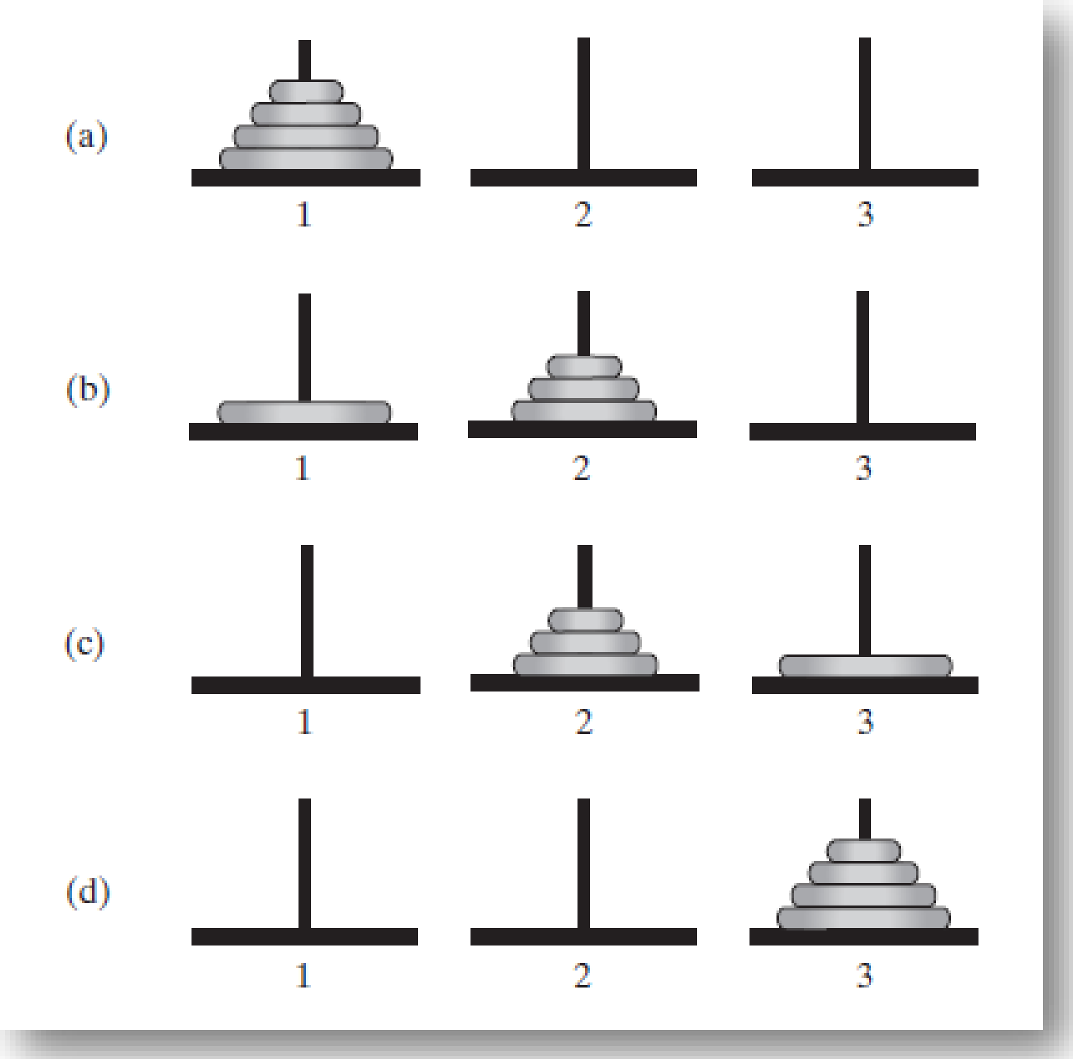
Solutions

The sequence of moves for solving the Towers of Hanoi problem with three disks



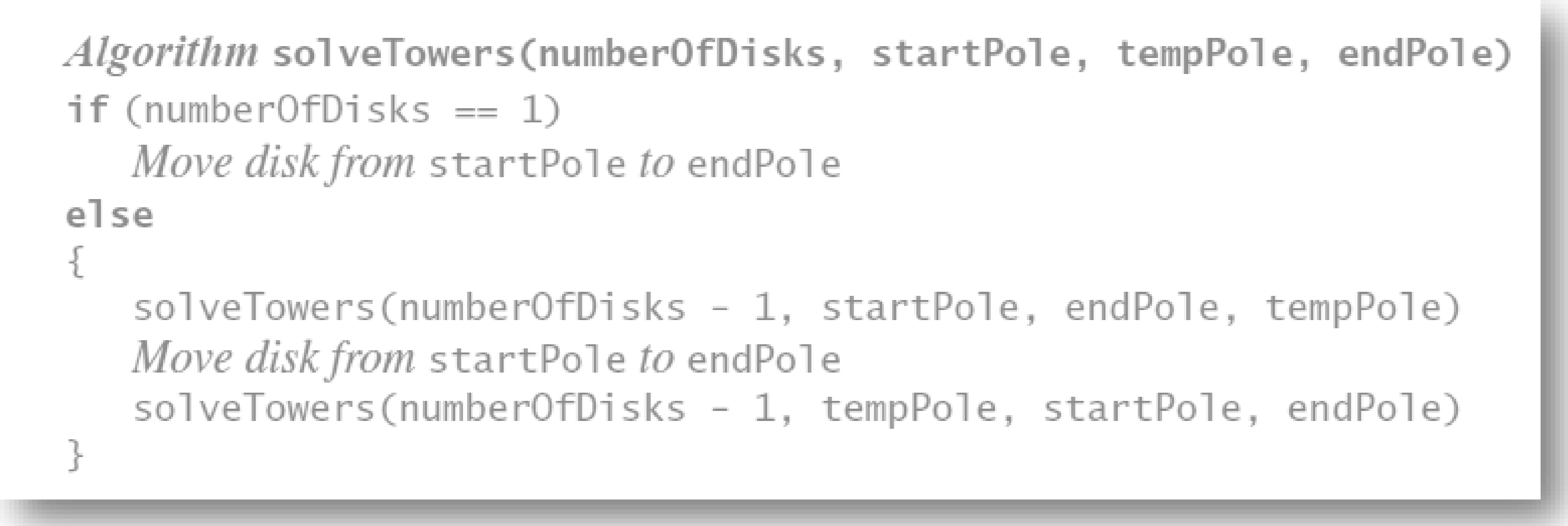
Solutions

The smaller problems in a recursive solution for four disks



Solutions

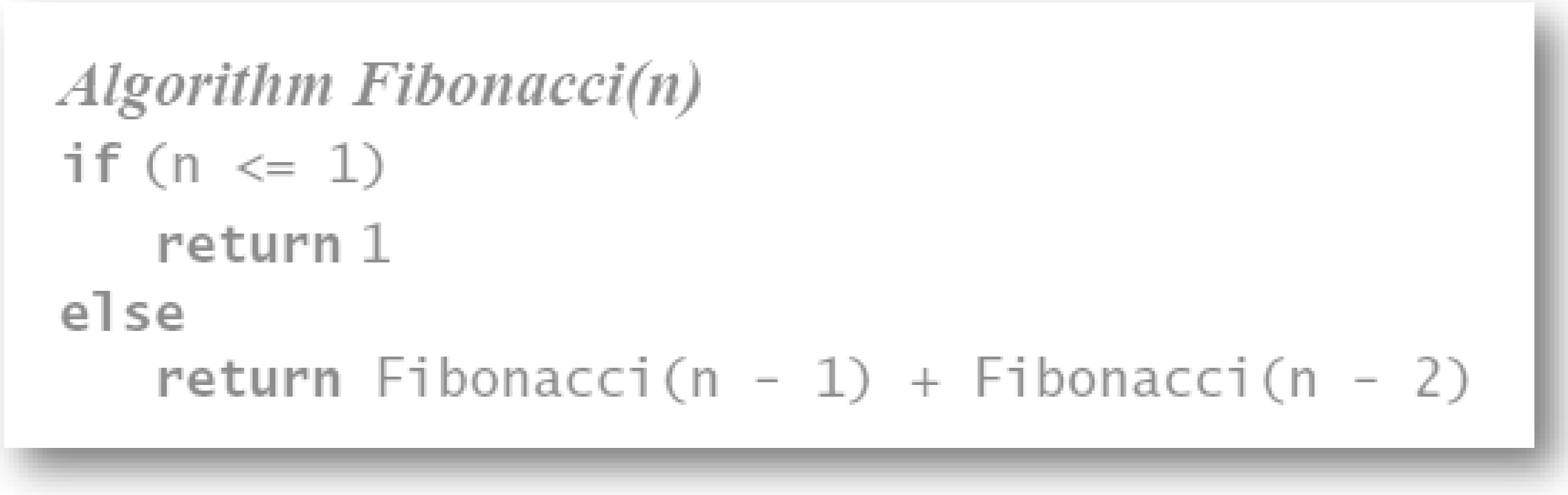
- Recursive algorithm to solve any number of disks.
  Note: for *n* disks, solution will be $2^n – 1$ moves



```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
    Move disk from startPole to endPole
else
{
    solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
    Move disk from startPole to endPole
    solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

Poor Solution to a Simple Problem

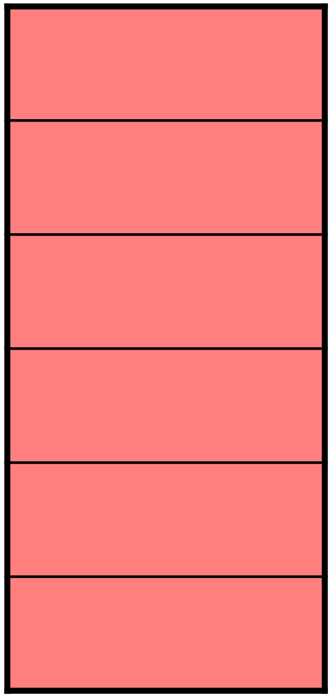
- Algorithm to generate Fibonacci numbers.
- Why is this inefficient?



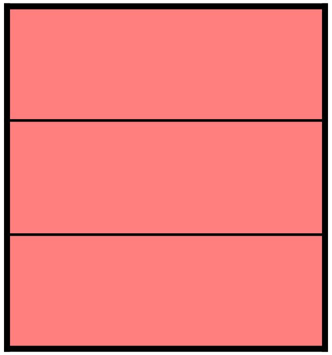
```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

Single recursion

- A recursive algorithm with a single recursive call still provides a linear chain of calls



Calls build run-time stack



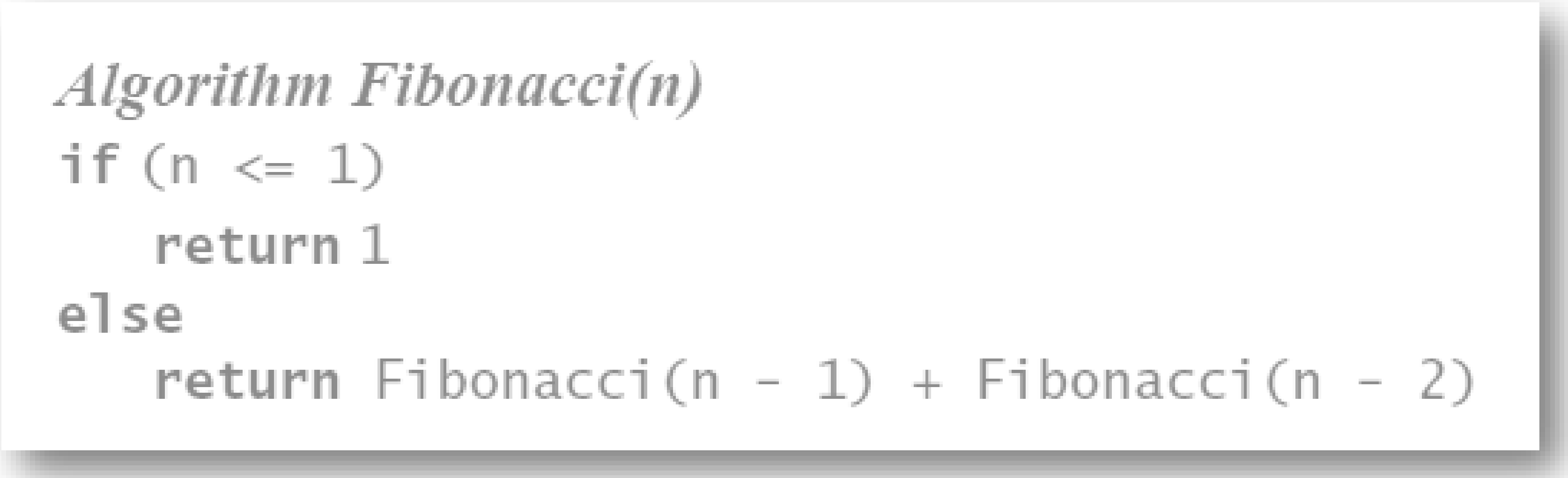
Stack shrinks as calls finish

Double recursion

- When a recursive algorithm has 2 calls, the execution trace is now a binary tree, as we saw with the trace on the board
  - This is execution is more difficult to do without recursion
    - To do it, programmer must create and maintain his/her own stack to keep all of the various data values
    - This increases the likelihood of errors / bugs in the code
- Later we will see some other classic recursive algorithms with multiple calls
  - Ex: MergeSort, QuickSort

Poor Solution to a Simple Problem

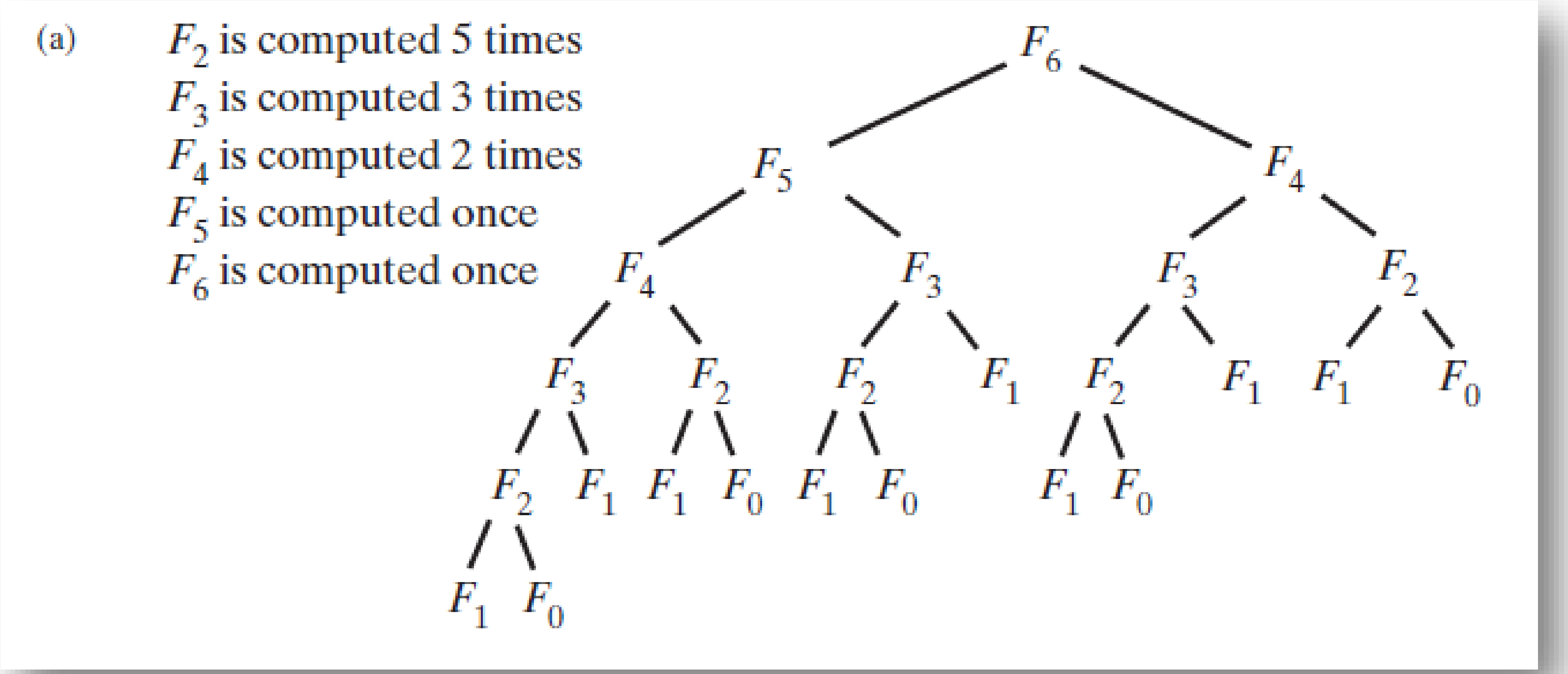
- Algorithm to generate Fibonacci numbers.
- Why is this inefficient?



```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

Poor Solution to a Simple Problem

- The computation of the Fibonacci number F_6 using (a) recursion … $F_n = \Omega(2^n)$



(a)

F_2 is computed 5 times
F_3 is computed 3 times
F_4 is computed 2 times
F_5 is computed once
F_6 is computed once

Converting Recursion into Iteration

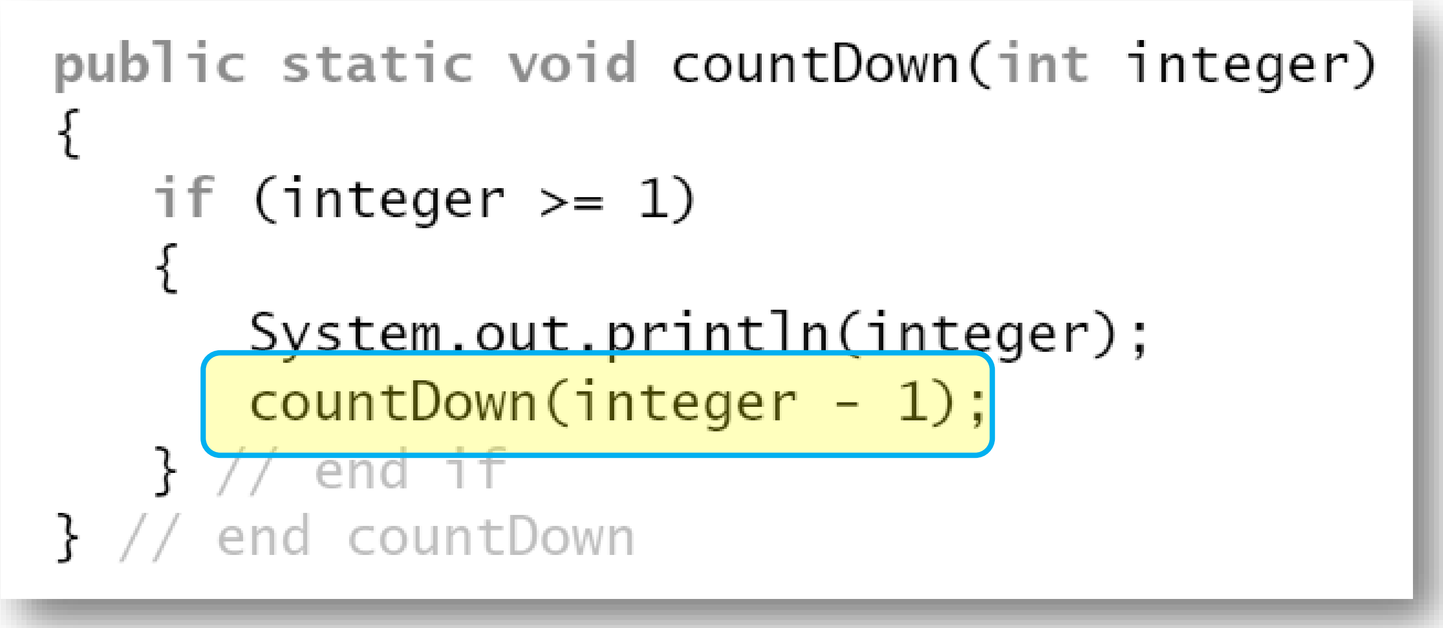
- Can we tell if a recursive algorithm can be easily done in an iterative way?
  - Yes – any recursive algorithm that is exclusively tail recursive can be done simply using iteration without recursion
  - Most algorithms we have seen so far are exclusively tail recursive

Tail Recursion

- So what is tail recursion?
  - Recursive algorithm in which the recursive call is the LAST statement in a call of the method
- What are the implications of tail recursion?
  - Any tail recursive algorithm can be converted into an iterative algorithm in a methodical way
    - In fact some compilers do this automatically

Tail Recursion

- When the last action performed by a recursive method is a recursive call.



```
public static void countDown(int integer)
{
    if (integer >= 1)
    {
        System.out.println(integer);
        countDown(integer - 1);
    } // end if
} // end countDown
```

Tail Recursion

- In a tail-recursive method, the last action is a recursive call
- This call performs a repetition that can be done by using iteration.
- Converting a tail-recursive method to an iterative one is usually a straightforward process.

Converting to tail-recursion

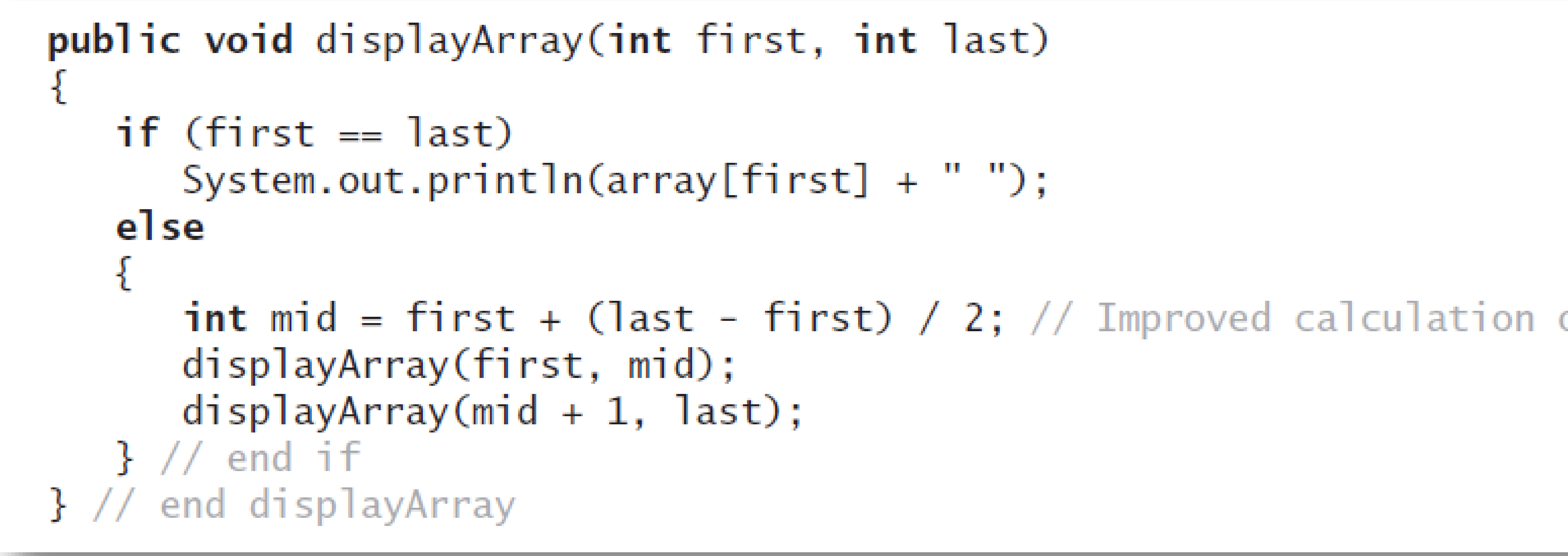
- Examples (Done on board)
  - Power
  - Fibonacci
  - Towers of Hanoi

Converting tail-recursion into iteration

- Examples (Done on board)
    - CountDown
    - Power
    - Fibonacci
    - Towers of Hanoi

Using a Stack Instead of Recursion

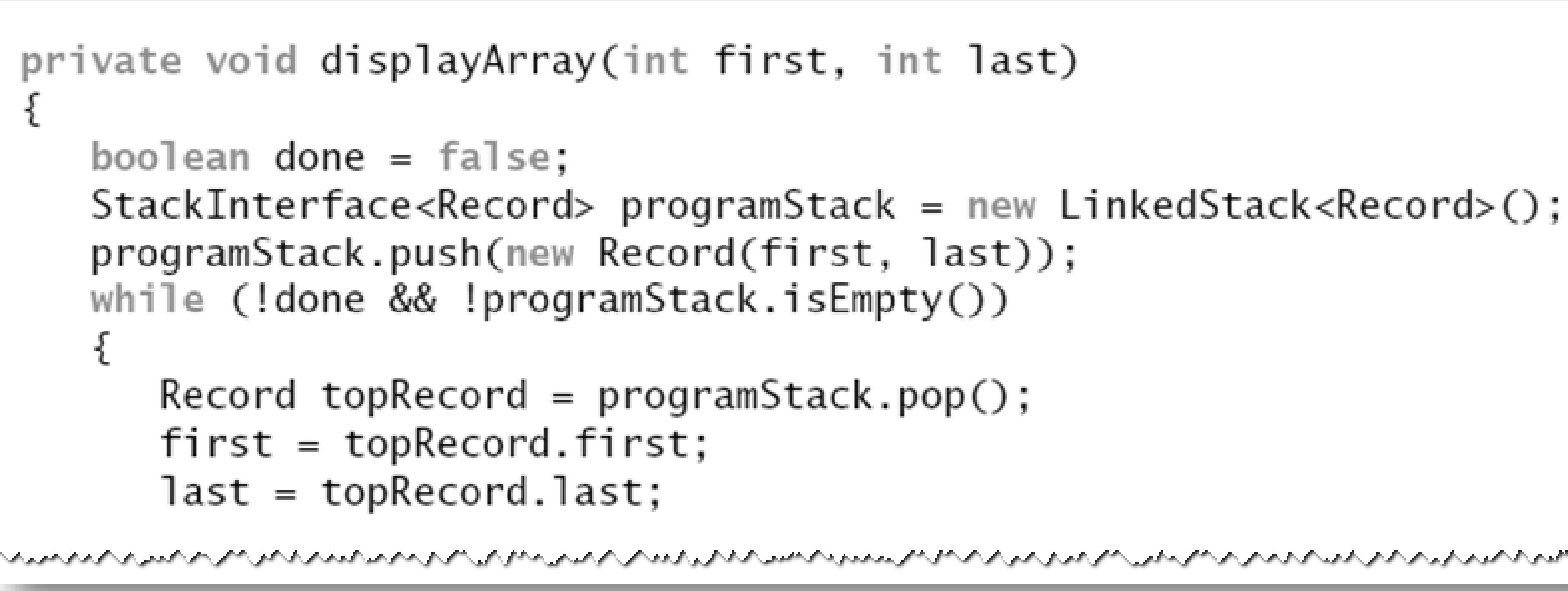
- An example of converting a recursive method to an iterative one



```
public void displayArray(int first, int last)
{
   if (first == last)
      System.out.println(array[first] + " ");
   else
   {
      int mid = first + (last - first) / 2; // Improved calculation
      displayArray(first, mid);
      displayArray(mid + 1, last);
   } // end if
} // end displayArray
```

Using a Stack Instead of Recursion

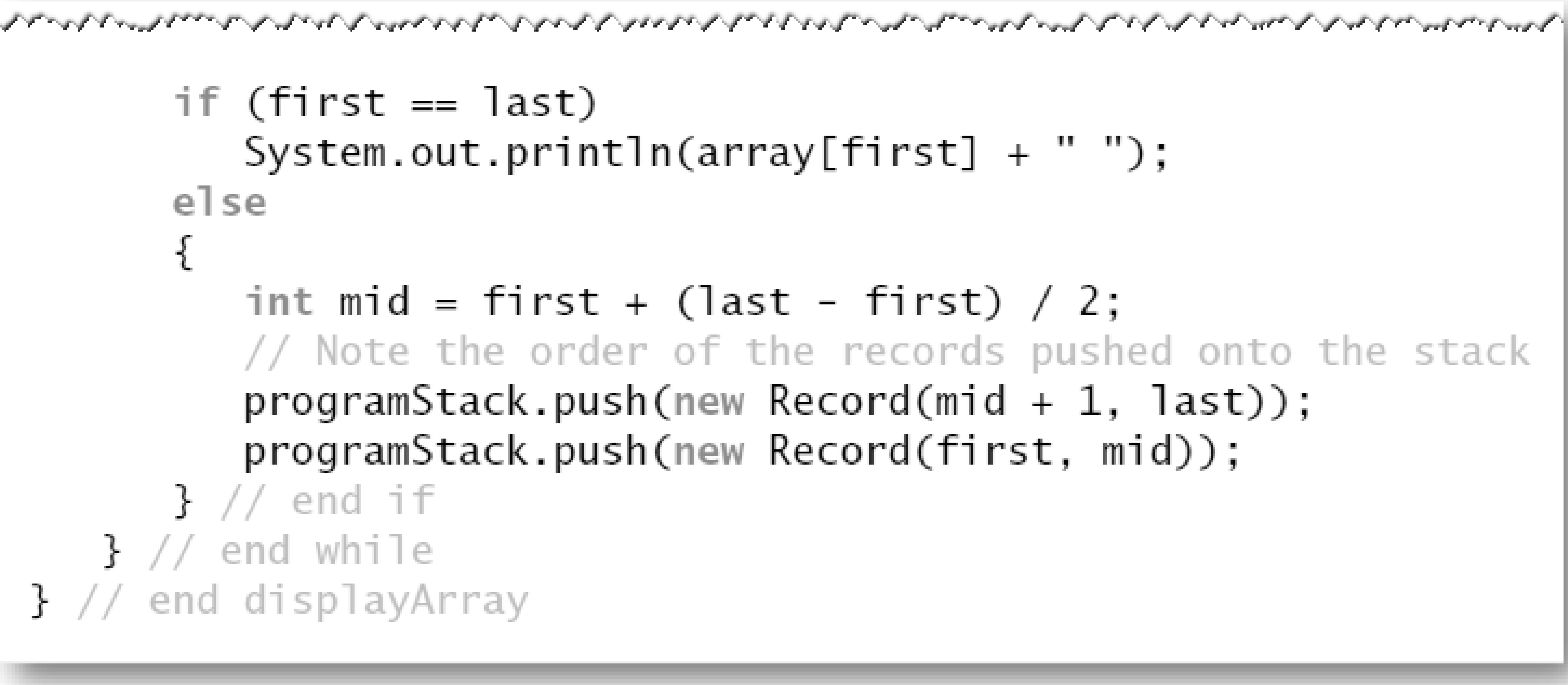
- An iterative **displayArray** to maintain its own stack



```
private void displayArray(int first, int last)
{
    boolean done = false;
    StackInterface<Record> programStack = new LinkedStack<Record>();
    programStack.push(new Record(first, last));
    while (!done && !programStack.isEmpty())
    {
        Record topRecord = programStack.pop();
        first = topRecord.first;
        last = topRecord.last;
```

Using a Stack Instead of Recursion

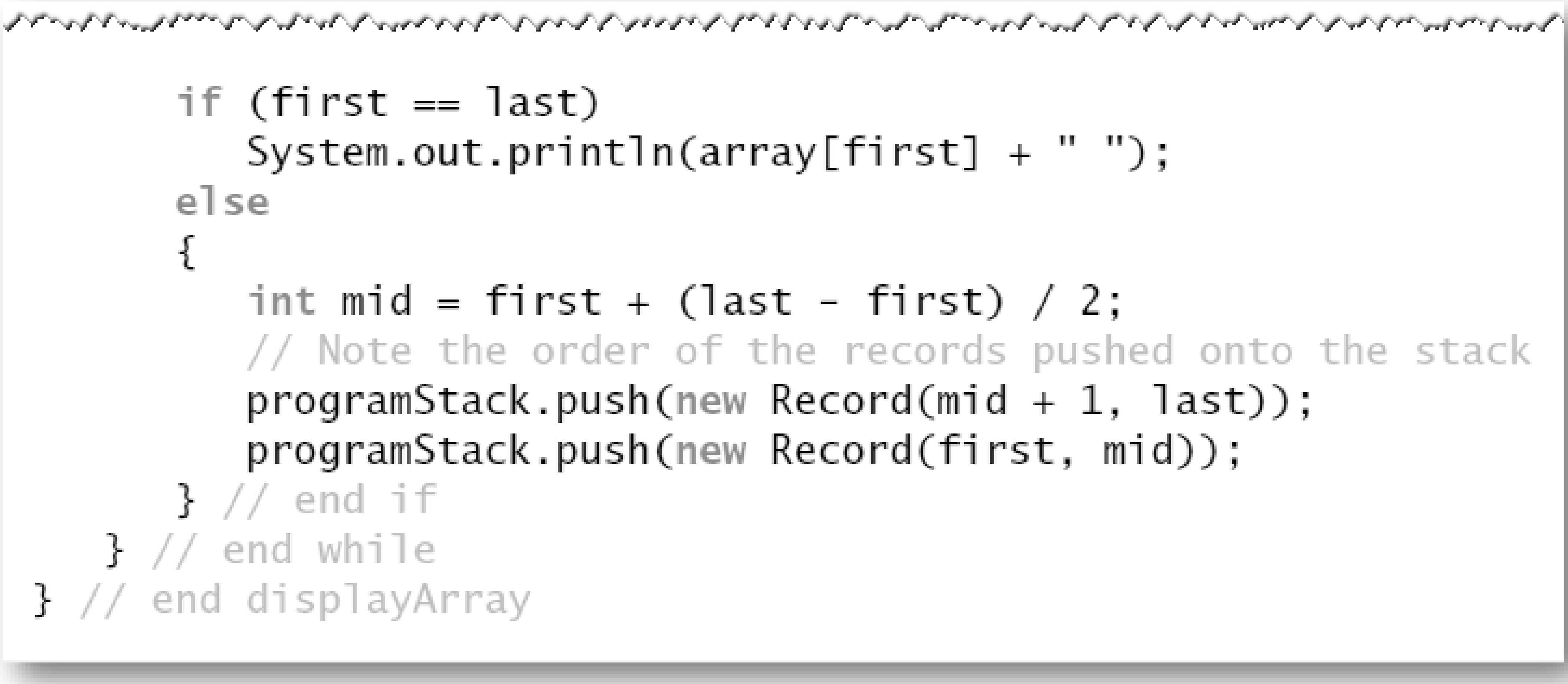
- An iterative **`displayArray`** to maintain its own stack



```
      if (first == last)
         System.out.println(array[first] + " ");
      else
      {
         int mid = first + (last - first) / 2;
         // Note the order of the records pushed onto the stack
         programStack.push(new Record(mid + 1, last));
         programStack.push(new Record(first, mid));
      } // end if
   } // end while
} // end displayArray
```

Using a Stack Instead of Recursion

- An iterative **displayArray** to maintain its own stack



```
      if (first == last)
         System.out.println(array[first] + " ");
      else
      {
         int mid = first + (last - first) / 2;
         // Note the order of the records pushed onto the stack
         programStack.push(new Record(mid + 1, last));
         programStack.push(new Record(first, mid));
      } // end if
   } // end while
} // end displayArray
```

Another example

- Towers of Hanoi
    - Check “Recursion to Iteration” handout

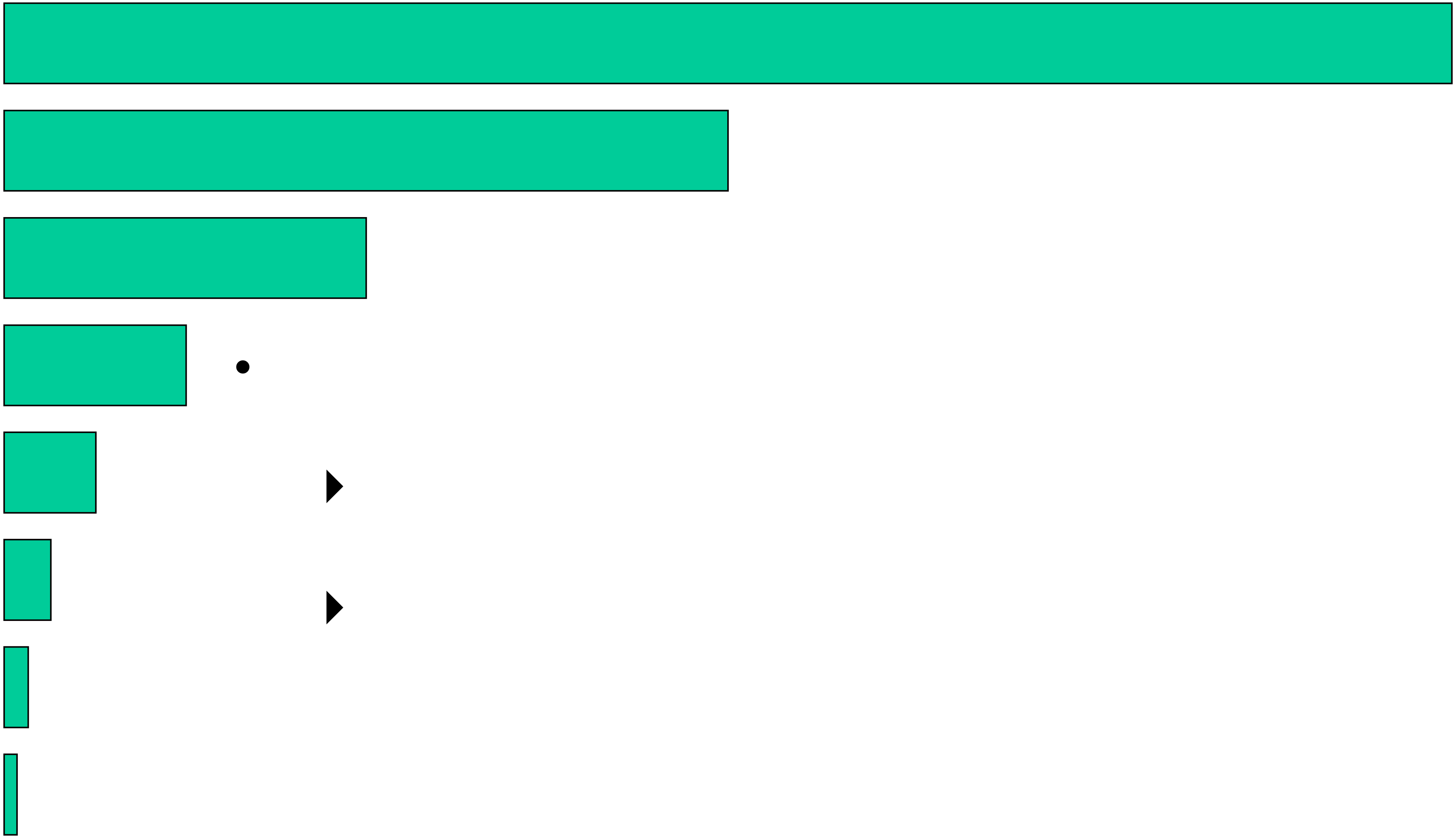
Overhead of Recursion

- Why do we care?
  - Recursive algorithms have overhead associated with them
    - Space: each activation record (AR) takes up memory in the run-time stack (RTS)
      - If too many calls "stack up" memory can be a problem
    - Time: generating ARs and manipulating the RTS takes time
      - A recursive algorithm will always run more slowly than an equivalent iterative version

Recursion and Divide and Conquer

- Divide and Conquer
  - The idea is that a problem can be solved by breaking it down to one or more "smaller" problems in a systematic way
    - Usually the subproblem(s) are a fraction of the size of the original problem
    - Usually the subproblems(s) are identical in nature to the original problem
    - It is fairly clear why these algorithms can typically be solved quite nicely using recursion

Recursion and Divide and Conquer



Recursion and Divide and Conquer

- How can we apply this to the Power fn?
  - We typically need to consider two important things:
    1) How do we break up or "divide" the problem into subproblems?
       - In other words, what do we do to the data to process it before making our recursive call(s)?
    2) How do we use the solutions of the subproblems to generate the solution of the original problem?
       - In other words, after the recursive calls complete, what do we do with the results?
  - For X^N the problem "size" is the exponent, N
    - So a subproblem would be the same problem with a smaller N

Recursion and Divide and Conquer

- Let's try cutting N in half – use N/2

1) We want to define X^N somehow in terms of $X^{N/2}$

   - We can't forget the base case

2) We need to determine how the original problem is solved in terms of the solution $X^{N/2}$

   - Done on board (and see notes below)

- Will this be an improvement over the other version of the function?
  - It seems like it since the problem is being cut in half each time
  - Informal analysis shows we only need $O(\log_2 N)$ multiplications in this case (see text)

Overhead of Recursion

- So what else is recursion good for?

1) For some problems, a recursive approach is more natural and simpler to understand than an iterative approach
    - Once the algorithm is developed, if it is tail recursive, we can always convert it into a faster iterative version
2) For some problems, it is very difficult to even conceive an iterative approach, especially if multiple recursive calls are required in the recursive solution

- Example: Backtracking problems

Recursion and Backtracking

- Idea of backtracking:
  - Proceed forward to a solution until it becomes apparent that no solution can be achieved along the current path
    - At that point UNDO the solution (backtrack) to a point where we can again proceed forward
  - Example: 8 Queens Problem
    - How can I place 8 queens on a chessboard such that no queen can take any other in the next move?
      - Recall that queens can move horizontally, vertically or diagonally for multiple spaces

8 Queens Problem

- How can we solve this with recursion and backtracking?
  - We note that all queens must be in different rows and different columns, so each row and each column must have exactly one queen when we are finished
    - Complicating it a bit is the fact that queens can move diagonally
  - So, thinking recursively, we see the following
    - To place 8 queens on the board we need to
      - Place a queen in a legal (row, column)
      - Recursively place 7 queens on the rest of the board
  - Where does backtracking come in?
    - Our initial choices may not lead to a solution – we need a way to undo a choice and try another one

8 Queens Problem

- Using this approach we come up with the solution as shown in 8-Queens handout
    - 8Queens.java
- Idea of solution:
    - Each recursive call attempts to place a queen in a specific column
        - A loop is used, since there are 8 squares in the column
    - For a given call, the state of the board from previous placements is known (i.e. where are the other queens?)
        - This is used to determine if a square is legal or not
    - If a placement within the column does not lead to a solution, the queen is removed and moved "down" the column

8 Queens Problem

    - When all rows in a column have been tried, the call terminates and backtracks to the previous call (in the previous column)
    - If a queen cannot be placed into column i, do not even try to place one onto column i+1 – rather, backtrack to column i-1 and move the queen that had been placed there
    - See handout for code details

- Why is this difficult to do iteratively?
  - We need to store a lot of state information as we try (and un-try) many locations on the board
    - For each column so far, where has a queen been placed?

8 Queens Problem

- The run-time stack does this automatically for us via activation records
  - Without recursion, we would need to store / update this information ourselves
  - This can be done (using our own Stack rather than the run-time stack), but since the mechanism is already built into recursive programming, why not utilize it?
- There are many other famous backtracking problems
  - http://en.wikipedia.org/wiki/Backtracking