University of Pittsburgh

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

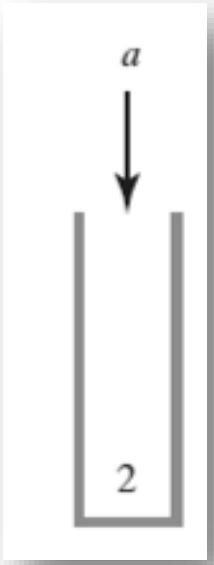
Step 3: Evaluating Postfix Expressions

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

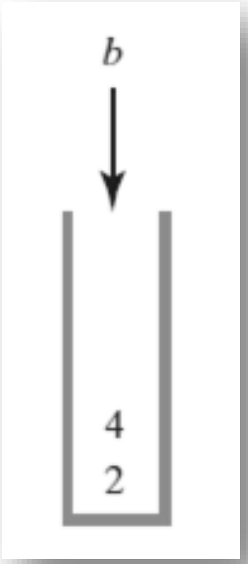
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

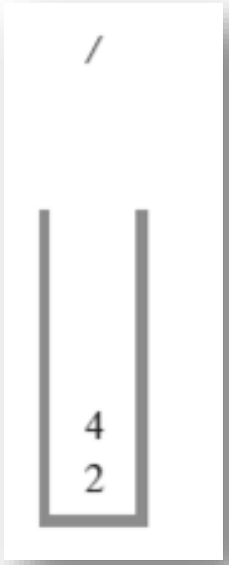
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

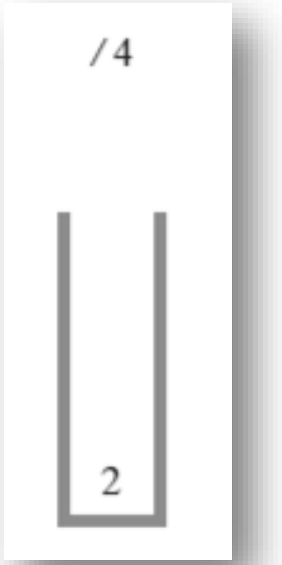
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

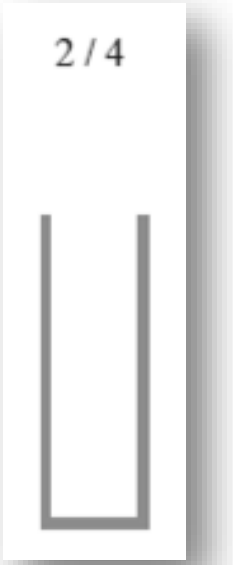
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

The stack during the evaluation of the postfix expression

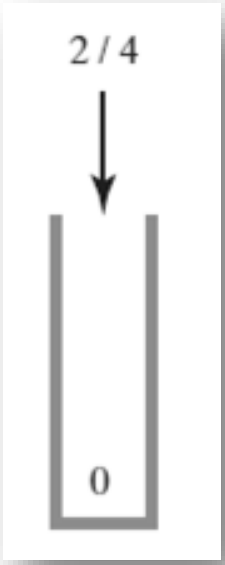
a b / when *a* is 2 and b is 4



Step 3: Evaluating Postfix Expressions

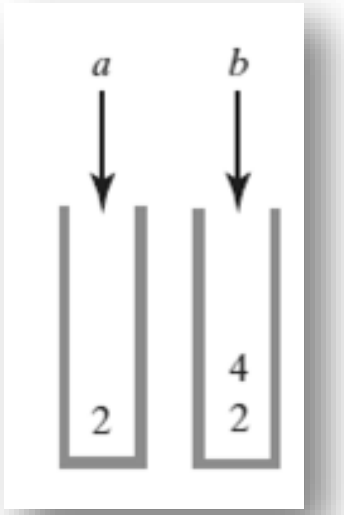
The stack during the evaluation of the postfix expression

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



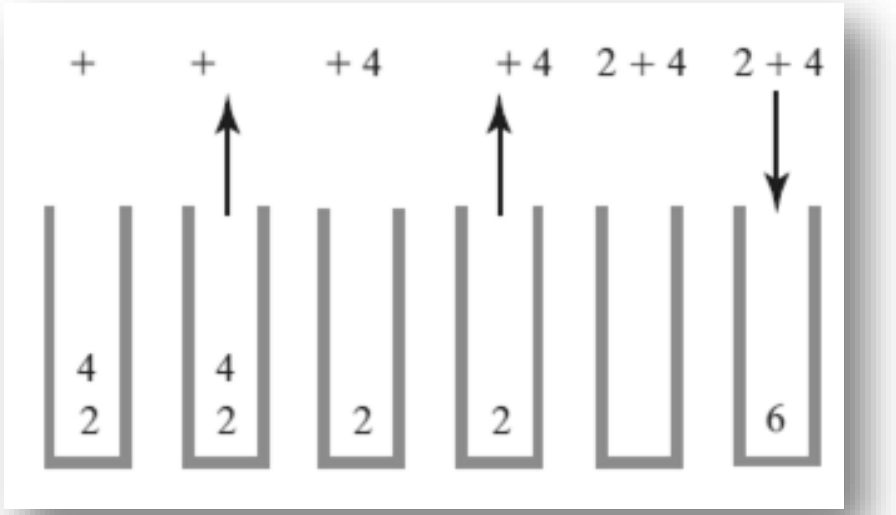
Evaluating Postfix Expressions: Example 2

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



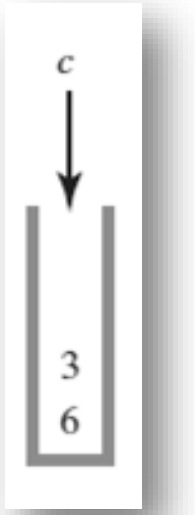
Evaluating Postfix Expressions: Example 2

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



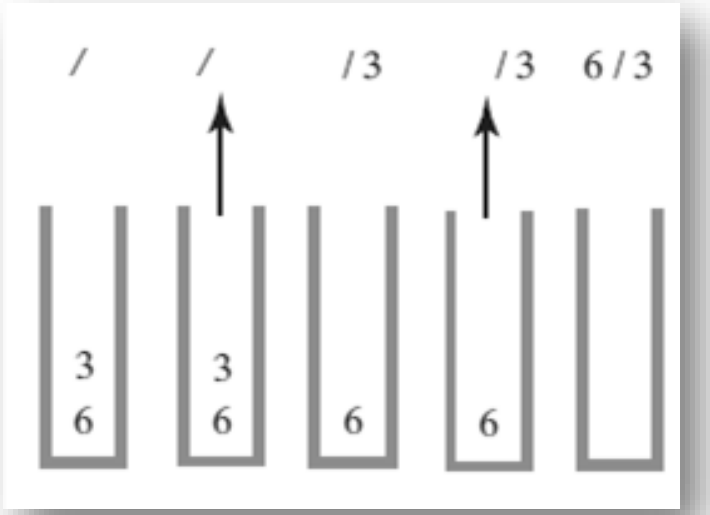
Evaluating Postfix Expressions: Example 2

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

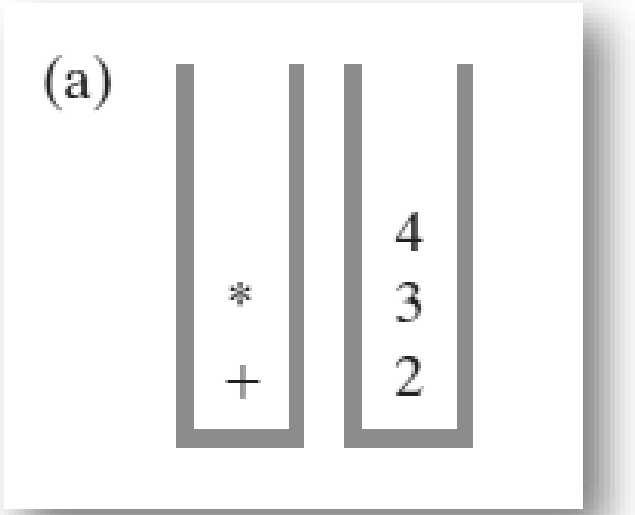
Evaluating Infix Expressions with 2 passes only

- We will use two stacks
  - Operator Stack
  - Operand stack
- Scan the expression once:
  - follow the steps of infix conversion to postfix,
  - **except**
    - 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

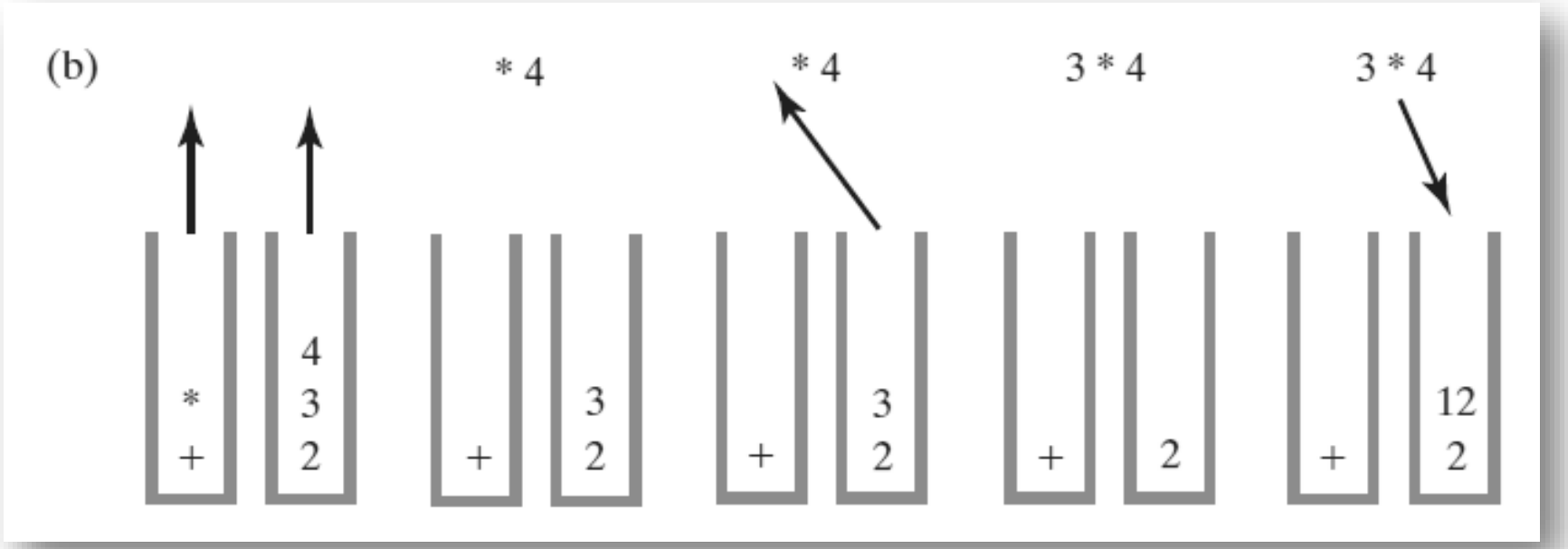
Two stacks during the evaluation of
a + b * c when a is 2, b is 3, and c is 4:

- after reaching the end of the expression;



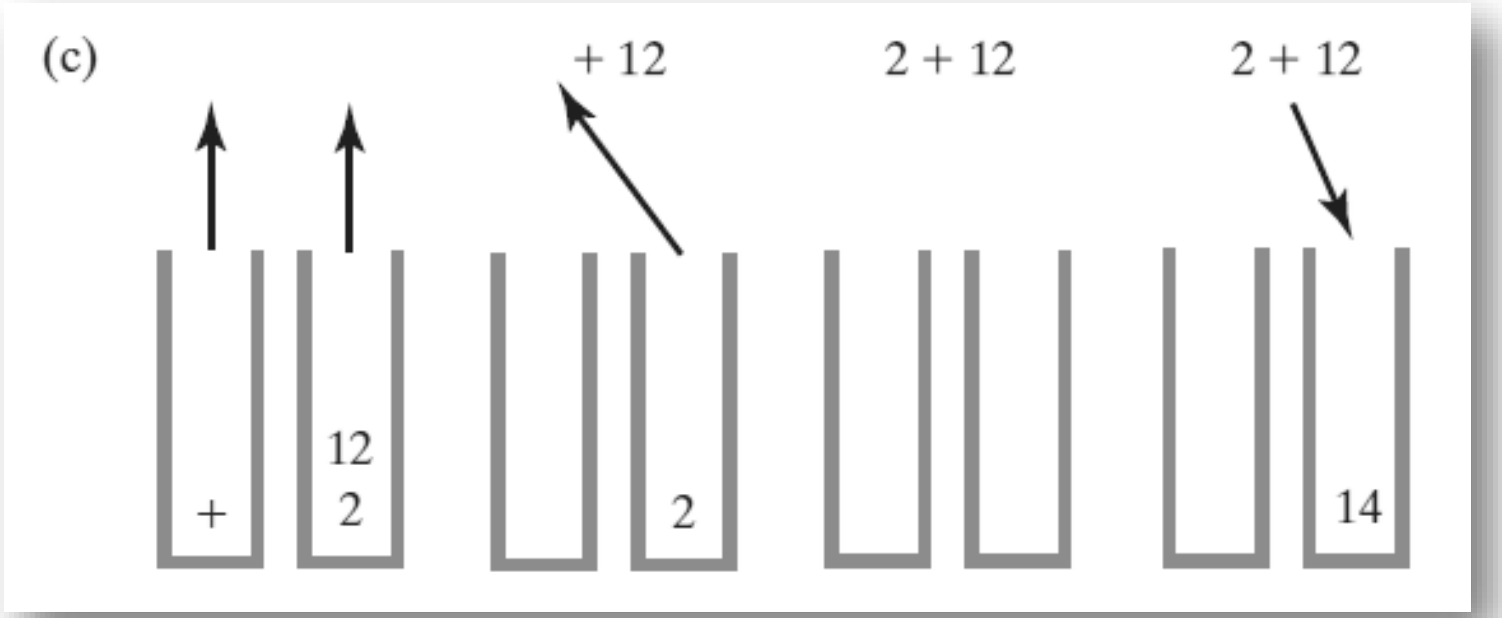
Evaluating Infix Expressions

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

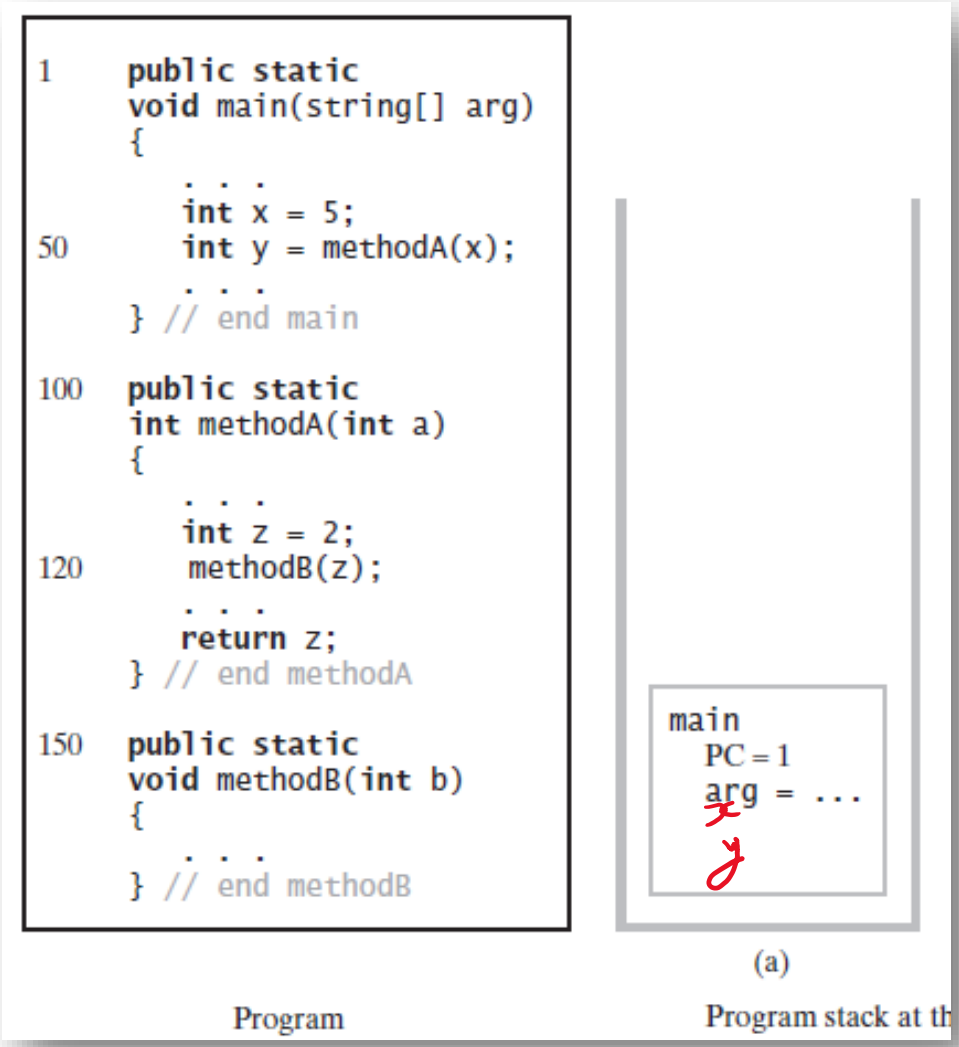
- The following code has three methods:
  - main
  - methodA
  - method
- main calls methodA
- methodA calls method
- Side note: methodA and method must be static
  - because they are called from main, which must static
- What are the local variables of each method?
- What are the parameters of each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

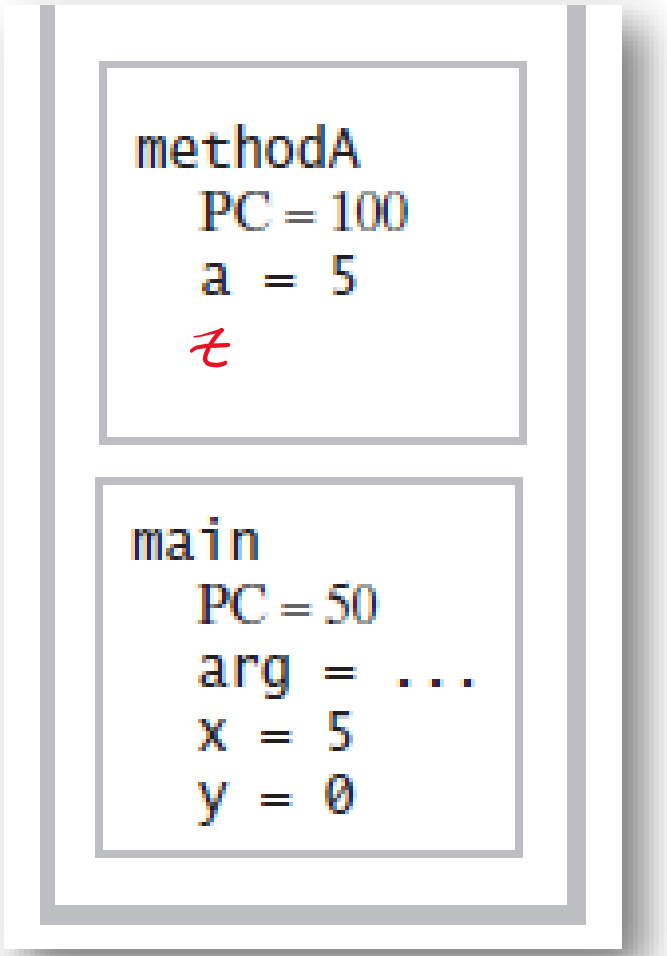


The Program Stack

- 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

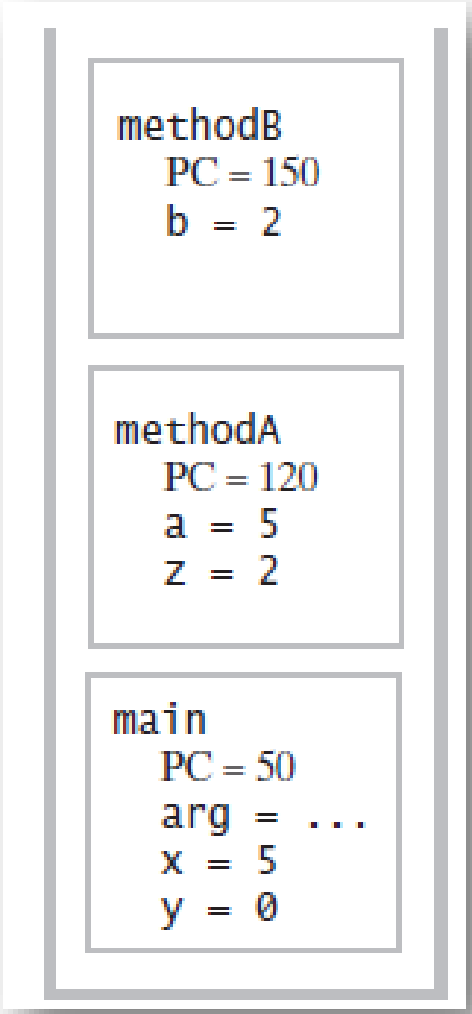


The Program Stack

- 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



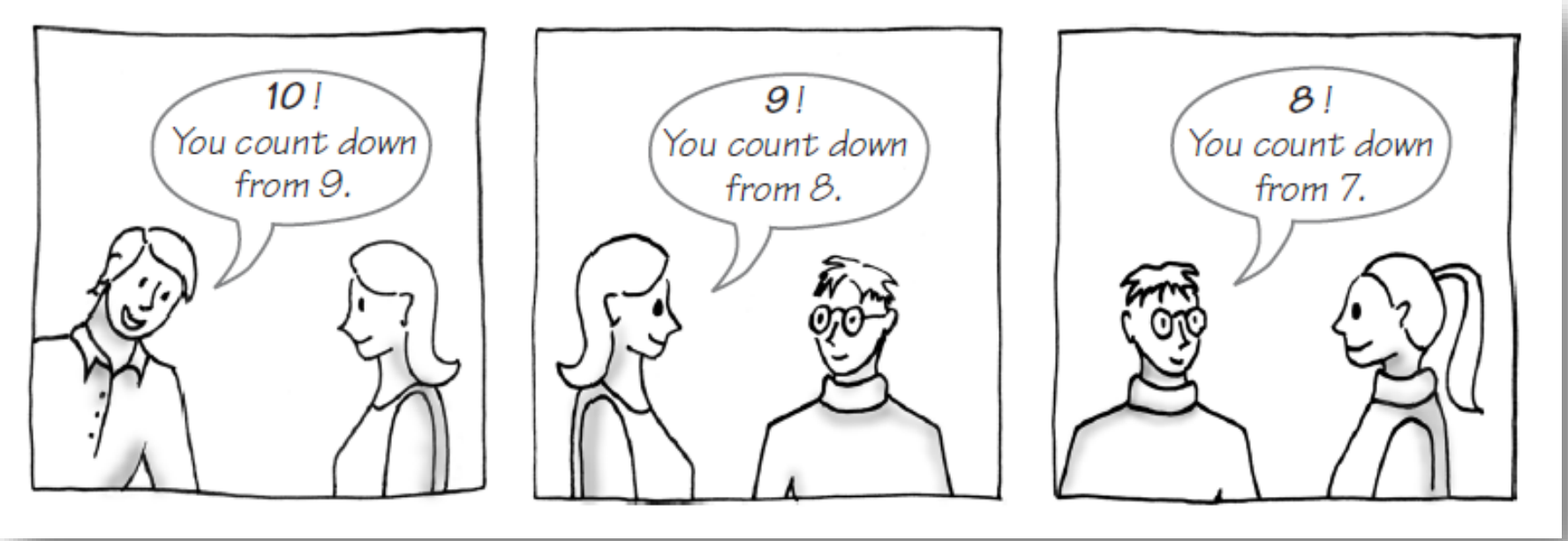
Recursion

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

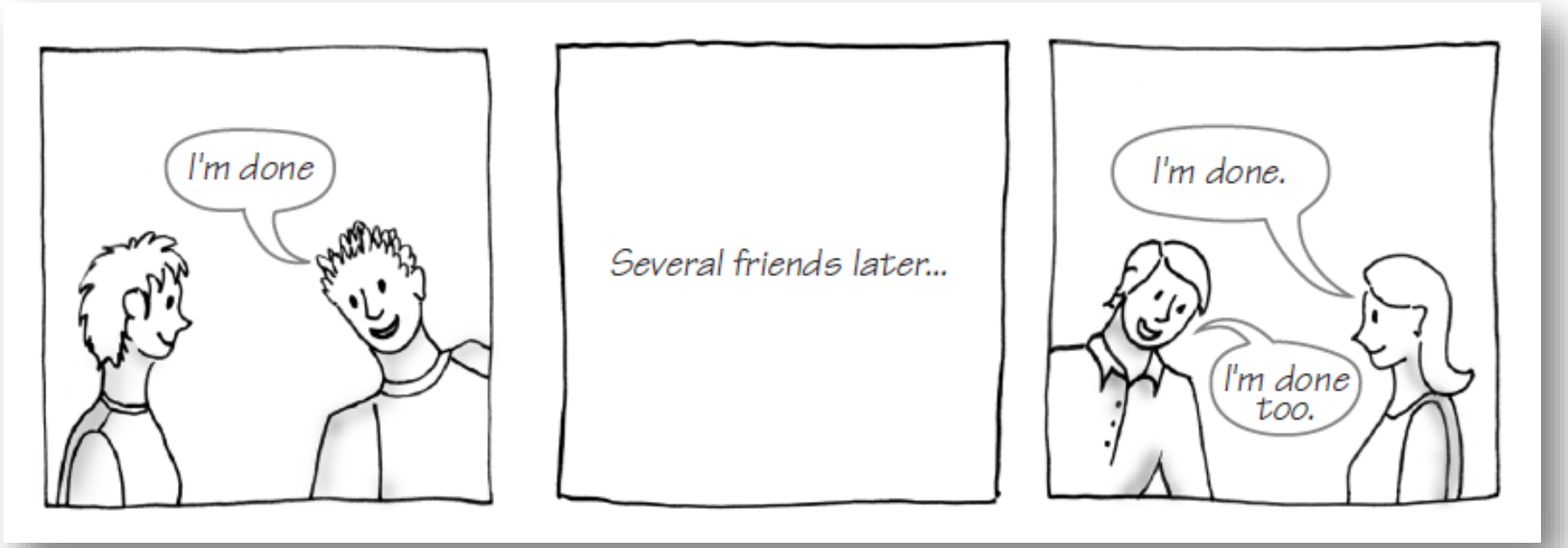


Example: The Countdown

Counting down from 10

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

```
/** 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

```
/** 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

```
/** 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

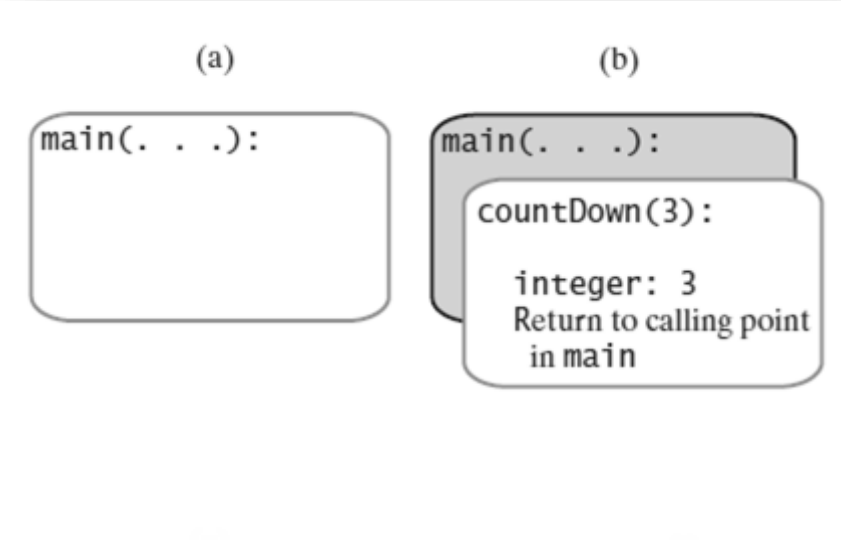
The stack of activation records during the execution of the call `countDown(3)`

pushing → activation records pile up

Tracing a Recursive Method

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

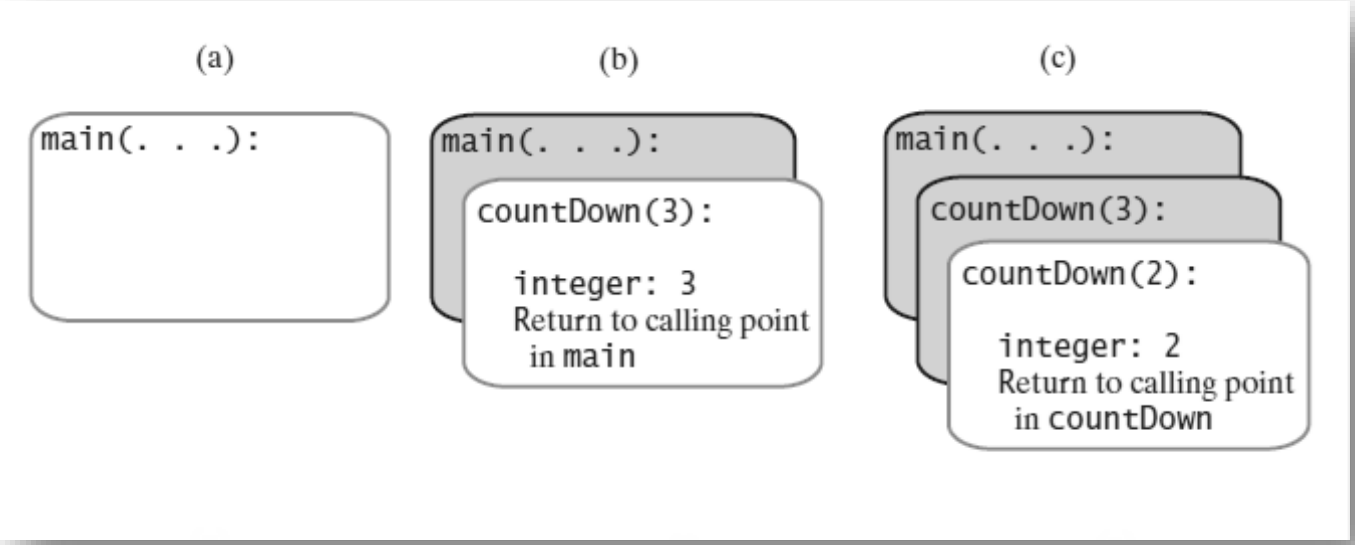
pushing → activation records pile up



Tracing a Recursive Method

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

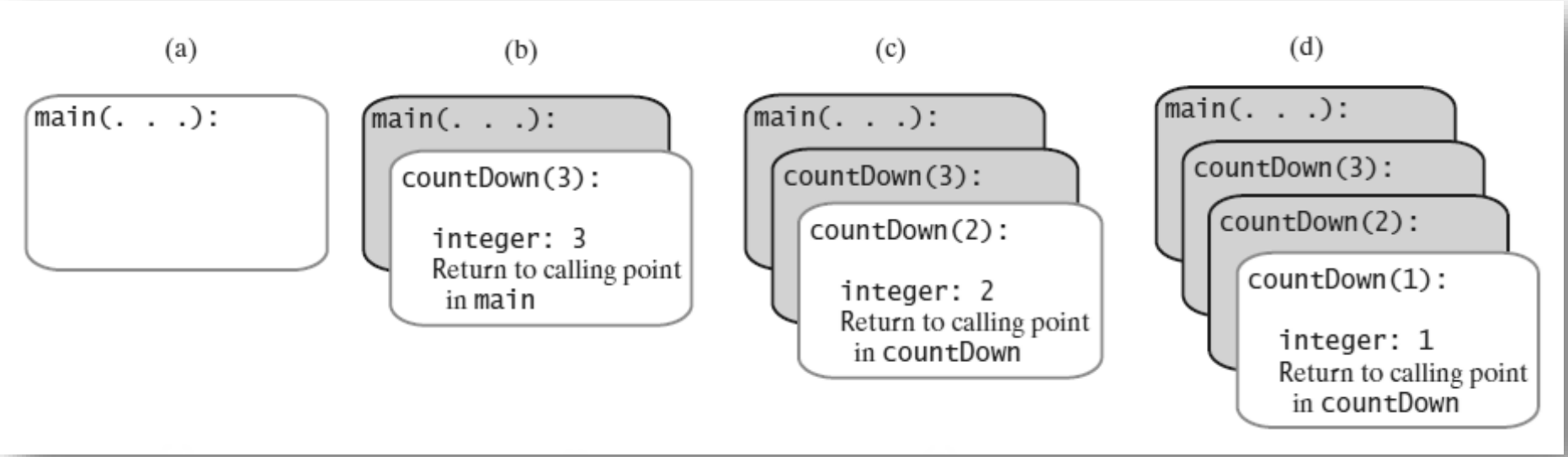
pushing → activation records pile up



Tracing a Recursive Method

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

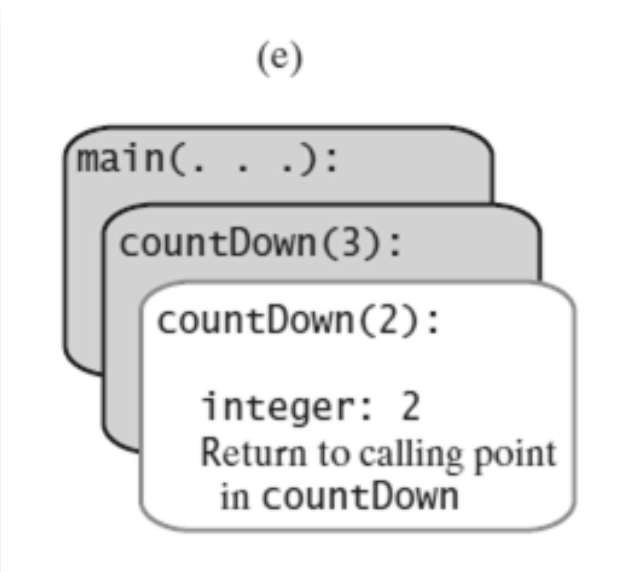
pushing → activation records pile up



Tracing a Recursive Method

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

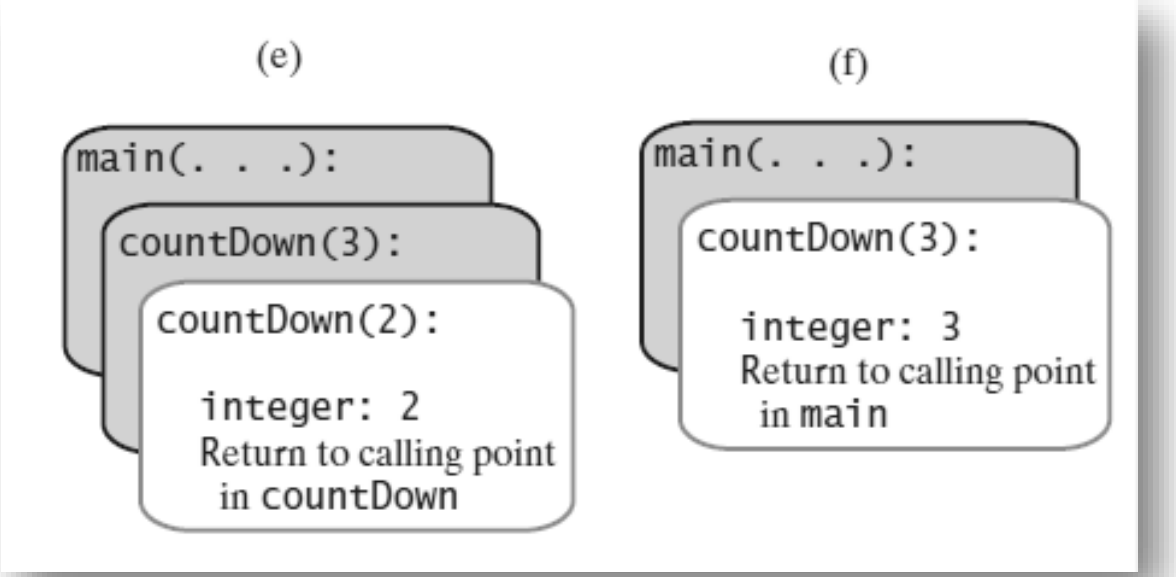
popping → activation records tear down



Tracing a Recursive Method

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

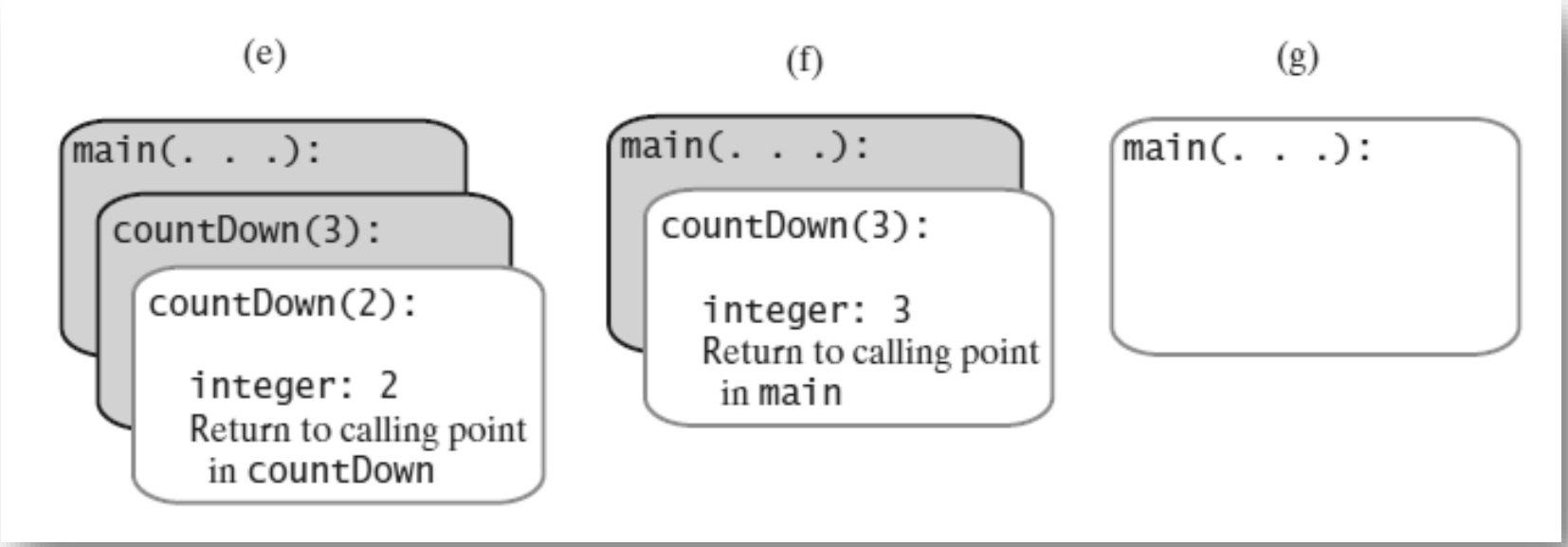
popping → activation records tear down



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

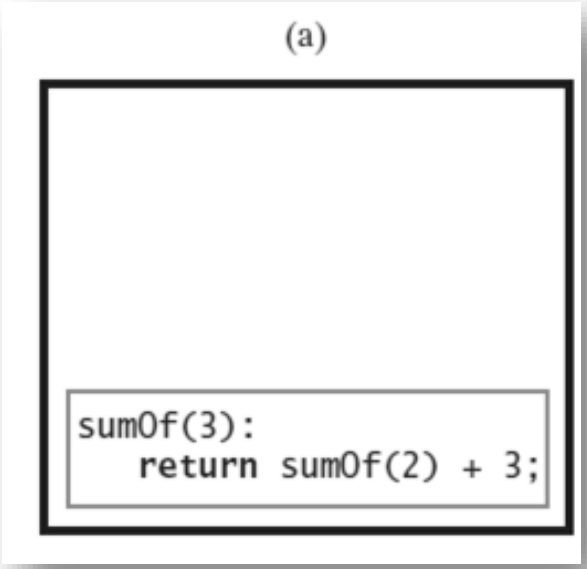
- Recursive method to calculate

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

```
/** @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
```

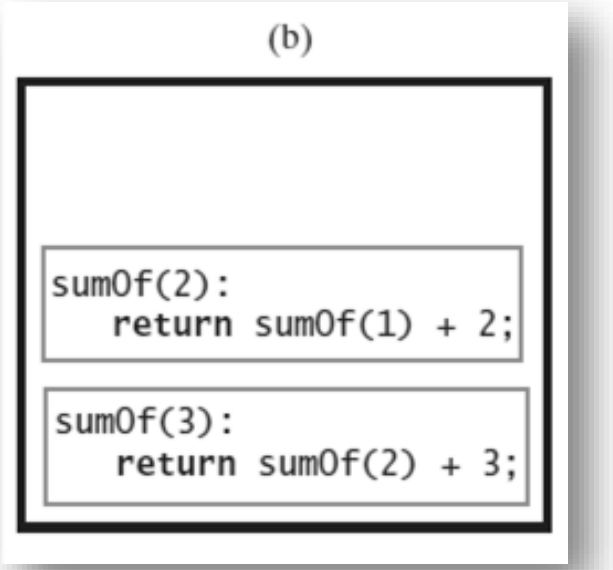
Tracing a Recursive Method

Tracing the execution of `sumOf(3)`



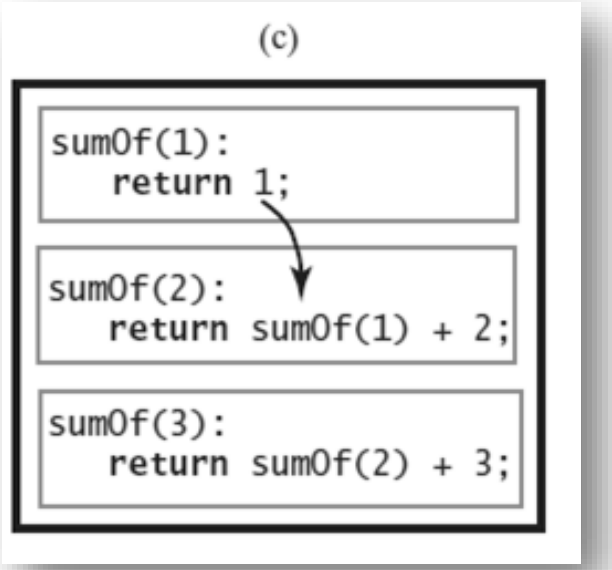
Tracing a Recursive Method

Tracing the execution of `sumOf(3)`



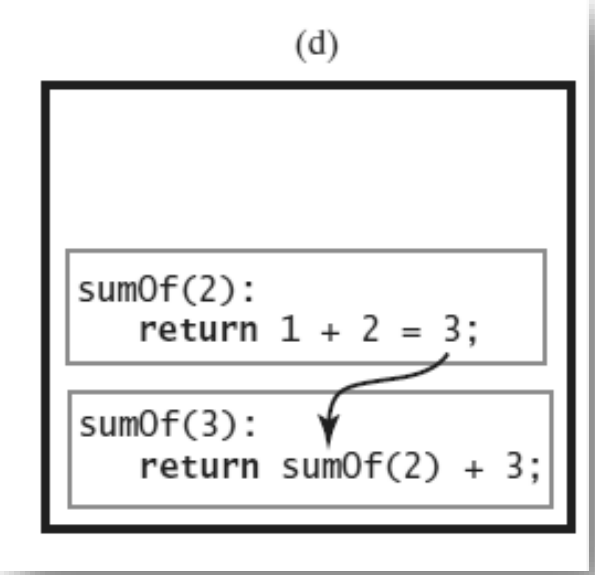
Tracing a Recursive Method

Tracing the execution of `sumOf(3)`



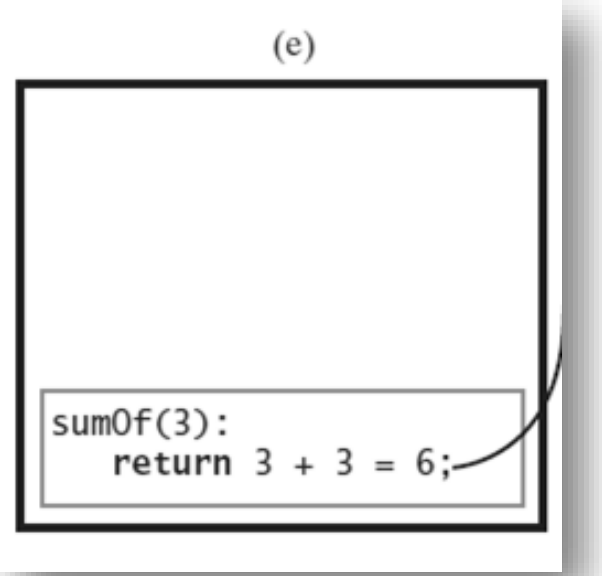
Tracing a Recursive Method

Tracing the execution of `sumOf(3)`



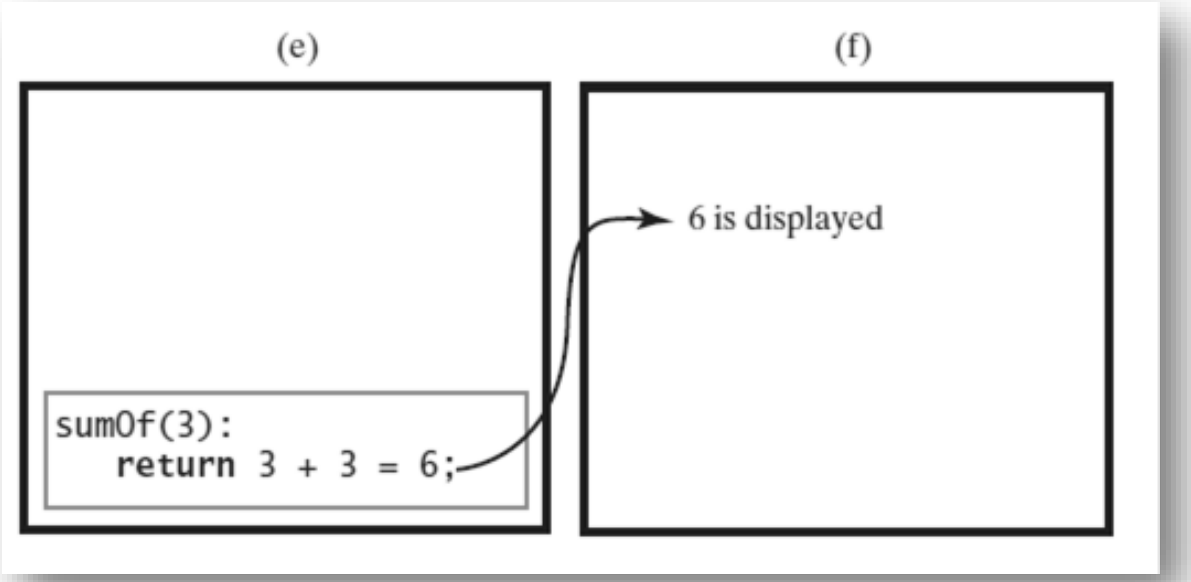
Tracing a Recursive Method

Tracing the execution of `sumOf(3)`



Tracing a Recursive Method

Tracing the execution of `sumOf(3)`



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

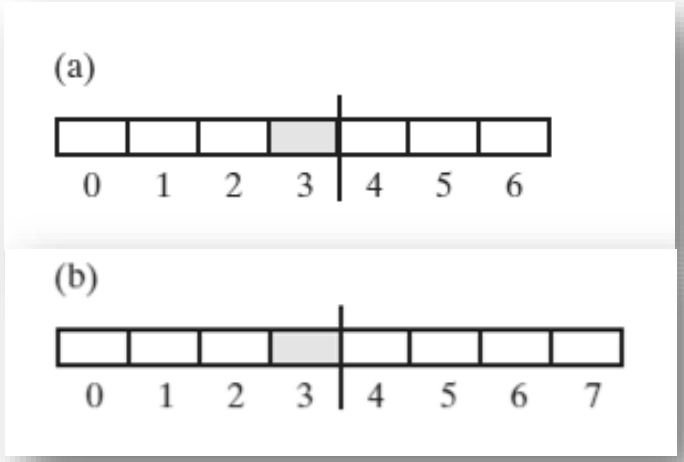
Alternatively, …

```
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
```