Stacks

Chapter 5

Contents

- Specifications of the ADT Stack
- Interfaces and StackInterface
- How to use Stack methods
- Need for “Wrapper” classes
  - And how to use their utility methods
- Using a Stack to Process “postfix” Algebraic Expressions
  - Reading a string and extracting “tokens”
  - Processing tokens with a stack
- The Program Runtime Stack
- Java Class Library: The Class `Stack`

Specifications of a Stack

- Organizes entries according to order added
- All additions added to one end of stack
  - Added to “top”
  - Called a “push”
- Access to stack restricted
  - Access only top entry
  - Remove called a “pop”

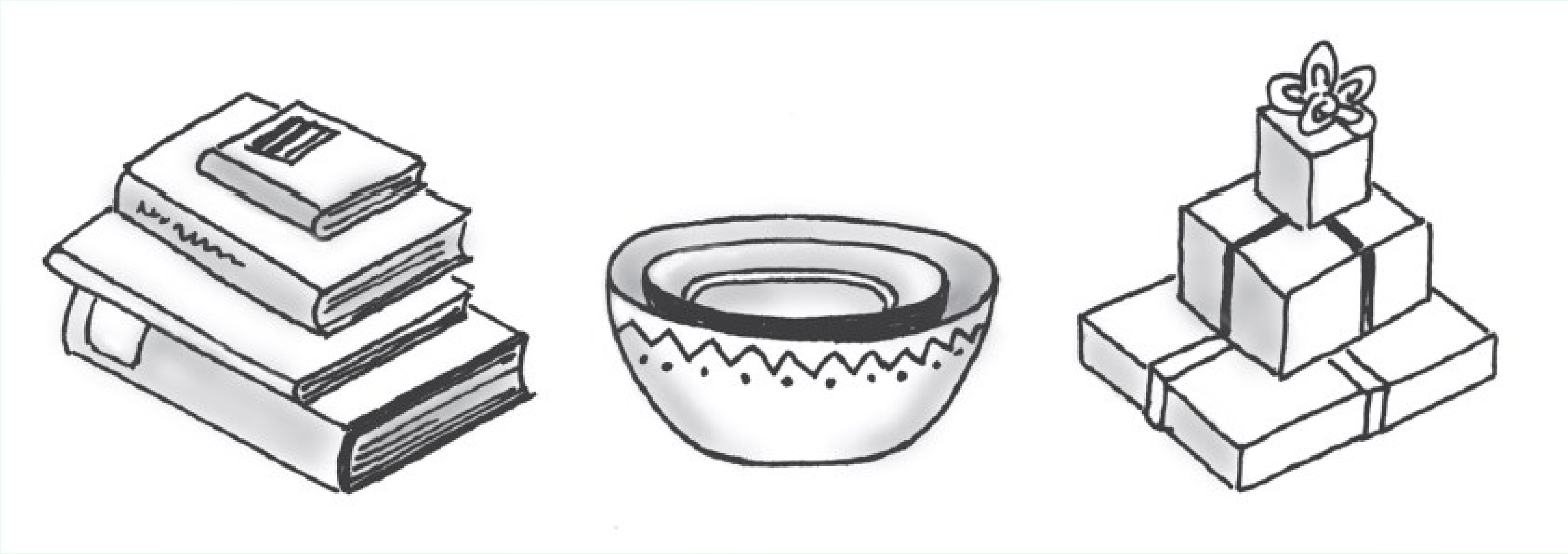


Figure 5-1 Some familiar stacks

All Stack-capable classes must implement these methods

interface StackInterface<T>

Constructor Summary
(Constructors provided by implementing classes)

Method Summary	
void	clear() Removes all items in this stack
boolean	isEmpty() Tests if this stack is empty.
T	peek() Returns item at the top of this stack without removing it if the stack is empty, returns null
T	pop() Removes and returns the item at the top of this stack if this stack is empty, returns null
void	push(T item) Pushes an item onto the top of this stack.

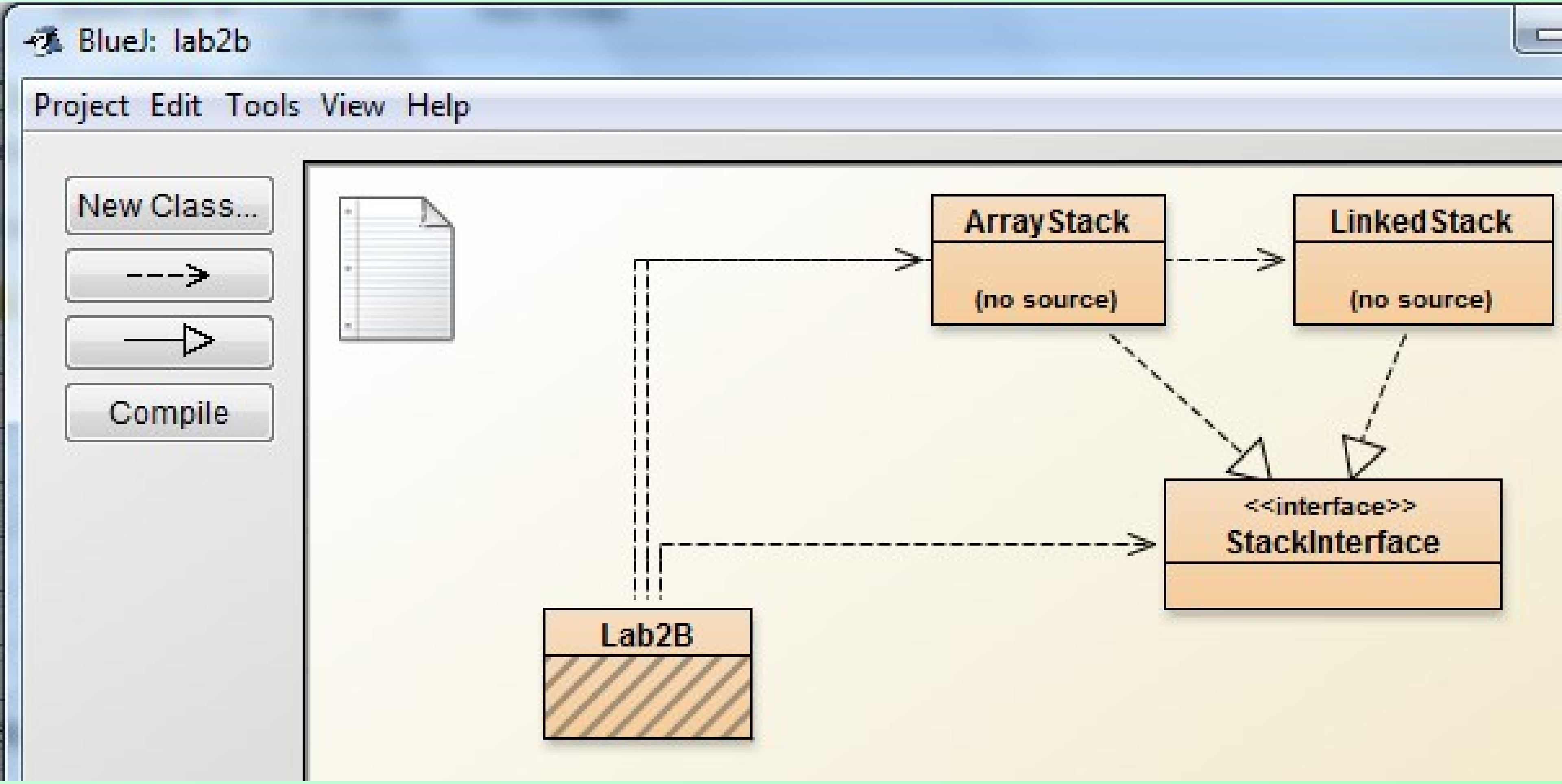
Stack Interface

```
public interface StackInterface < T >
{
   /** Adds a new entry to the top of this stack.
   @param newEntry an object to be added to the stack */
   public void push (T newEntry);

   /** Removes and returns this stacks top entry.
   @return either the object at the top of the stack or, if the
   stack is empty before the operation, null */
   public T pop ();

   /** Retrieves this stacks top entry.
   @return either the object at the top of the stack or null if
   the stack is empty */
   public T peek ();

   /** Detects whether this stack is empty.
   @return true if the stack is empty */
   public boolean isEmpty ();

   /** Removes all entries from this stack */
   public void clear ();

} // end StackInterface
```

What is an Interface?

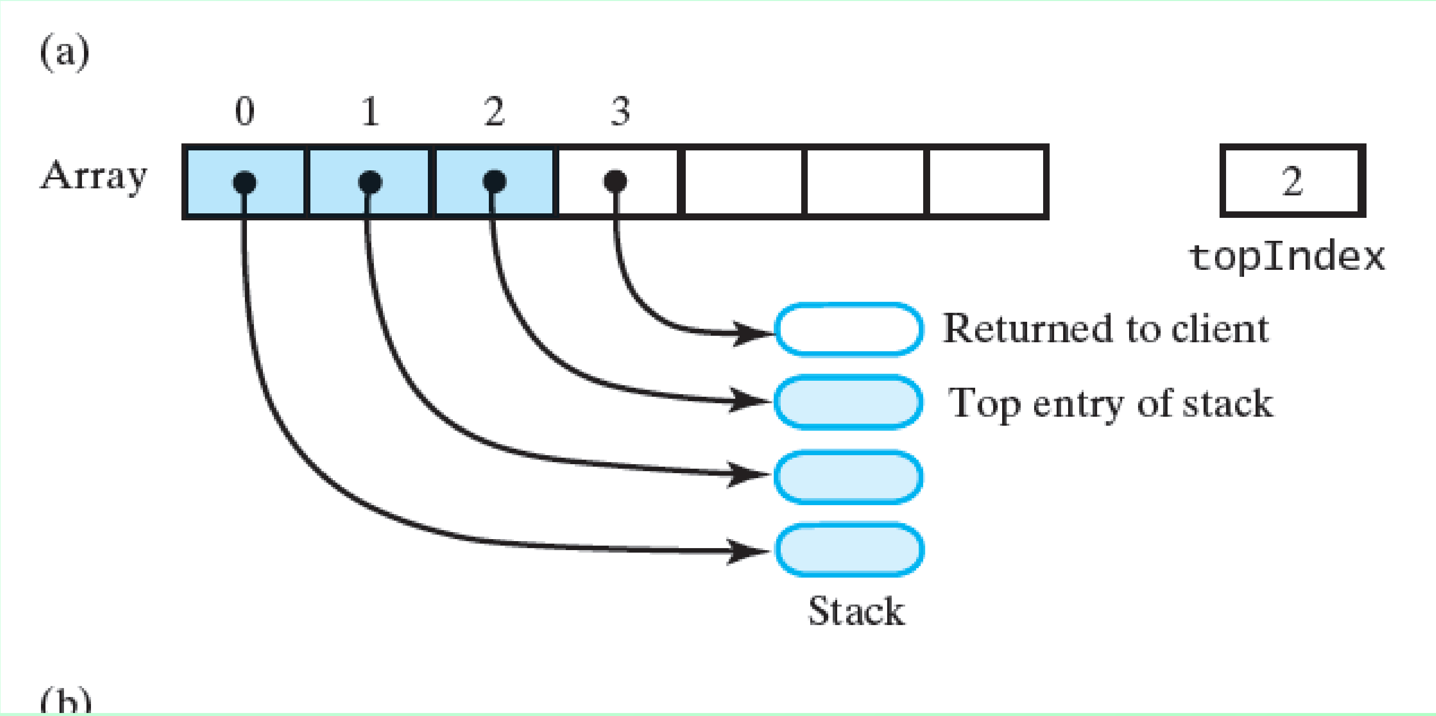
- Like a class, except none of the methods defined
  - Only “signatures” showing what they take/return
  - Uses keyword <u>interface</u> rather than <u>class</u>
  - <u>Like a Contract</u>
    - If a class satisfies methods in StackInterface
    - it can be used interchangeably with any other class that satisfies method in StackInterface
  - Allows multiple implementations for a given ADT,

BlueJ showing StackInterface and 2 classes that implement it



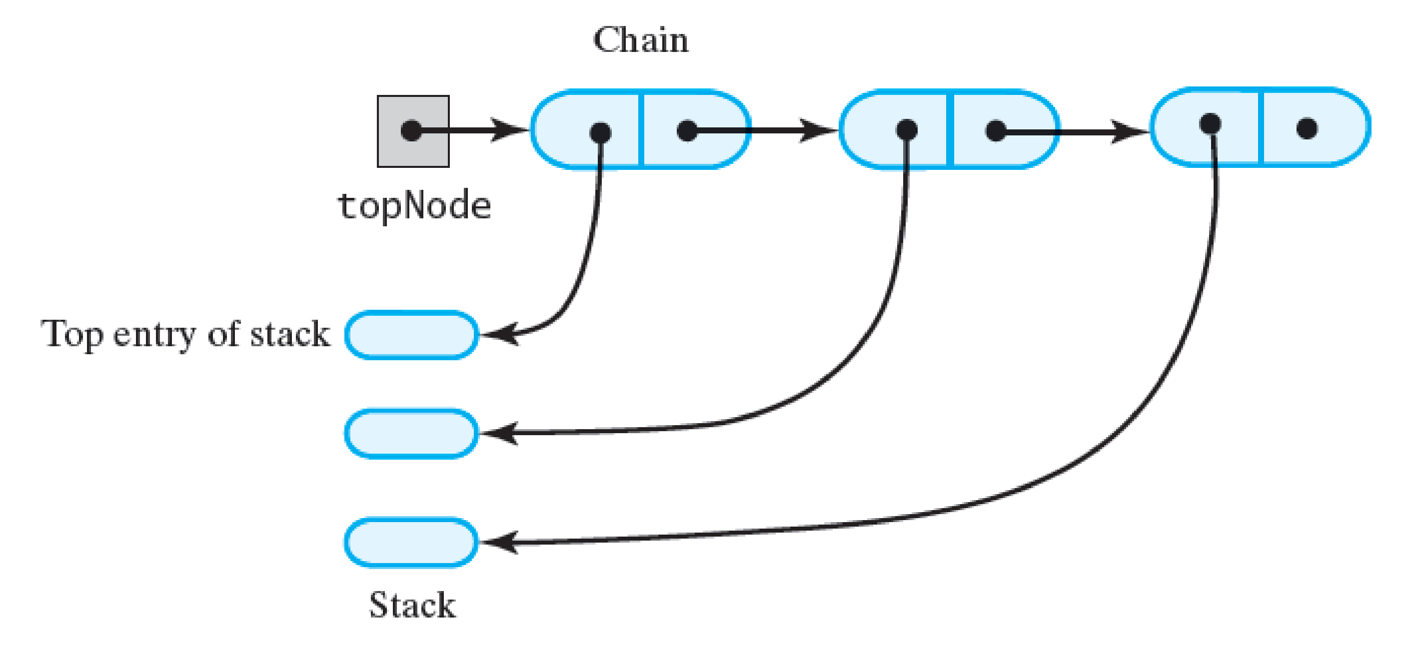
Stack Implementation 1)--ArrayStack

- For an ArrayStack, we choose to represent data using an internal array



Stack Implementation 2)-- LinkedStack

- For a LinkedStack we choose to represent data using a linked chain of nodes:



- Both approaches have pros and cons

Interfaces allow flexibility

- If both ArrayStack and LinkedStack implement StackInterface, we can use either one in a program that calls for stacks.
- Depending on our app, one or the other might perform better.
- Regardless of which we use, all stack code except instantiation is same

Using Class Stack

- Example usage

```
StackInterface<String> stringStack = new OurStack<String>();
stringStack.push("Jim");
stringStack.push("Jess");
stringStack.push("Jill");
stringStack.push("Jane");
stringStack.push("Joe");

String top = stringStack.peek(); // returns "Joe"
System.out.println(top + " is at the top of the stack.");

top = stringStack.pop();         // removes and returns "Joe"
System.out.println(top + " is removed from the stack.");

top = stringStack.peek();        // returns "Jane"
System.out.println(top + " is at the top of the stack.");
top = stringStack.pop();         // removes and returns "Jane"
System.out.println(top + " is removed from the stack.");
```

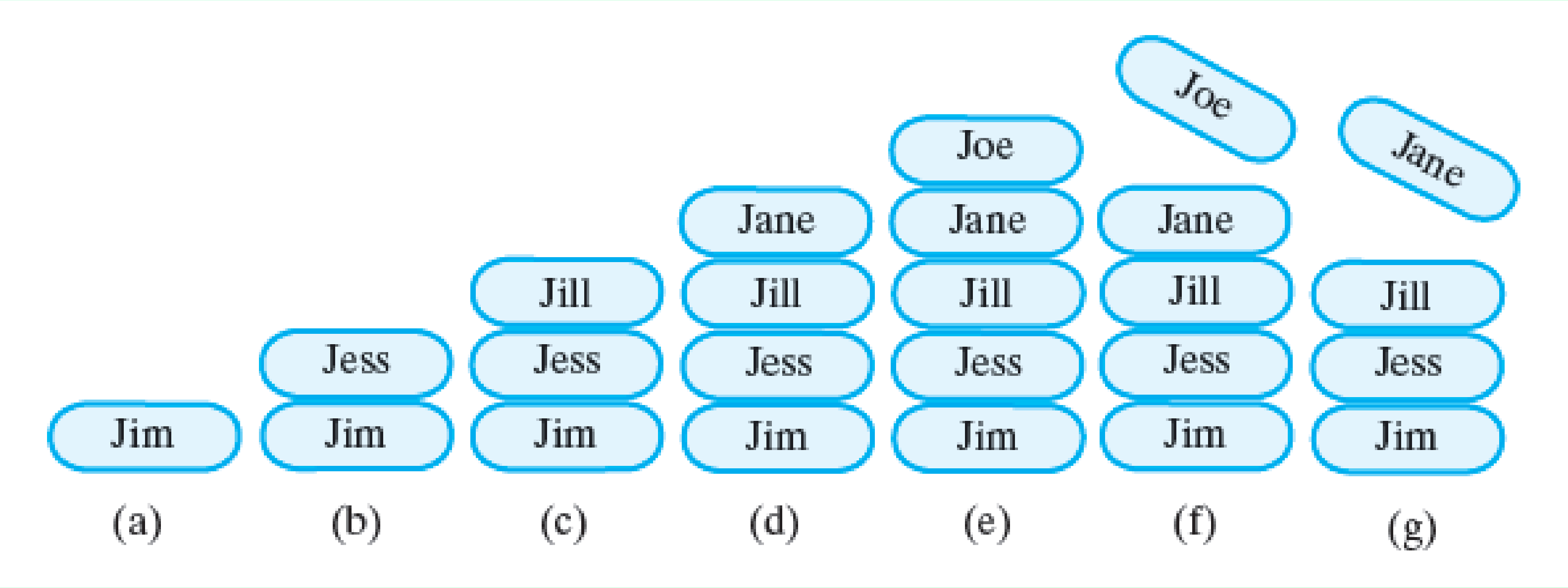


Figure 5-2 A stack of strings after (a) **`push`** adds Jim; (b) **`push`** adds Jess; (c) **`push`** adds Jill; (d) **`push`** adds Jane; (e) **`push`** adds Joe; (f) **`pop`** retrieves and removes Joe; (g) **`pop`** retrieves and removes Jane

Question 1 After the following statements execute, what string is at the top of the stack and what string is at the bottom?

```
StackInterface<String> stringStack = new OurStack<String>();
stringStack.push("Jim");
stringStack.push("Jess");
stringStack.pop();
stringStack.push("Jill");
stringStack.push("Jane");
stringStack.pop();
```

Question 2 Consider the stack that was created in Question 1, and define a new empty stack `nameStack`.

a. Write a loop that pops the strings from `stringStack` and pushes them onto `nameStack`.

b. Describe the contents of the stacks `stringStack` and `nameStack` when the loop that you just wrote completes its execution.

1. *Jill* is at the top, and *Jim* is at the bottom.

2. **a.**

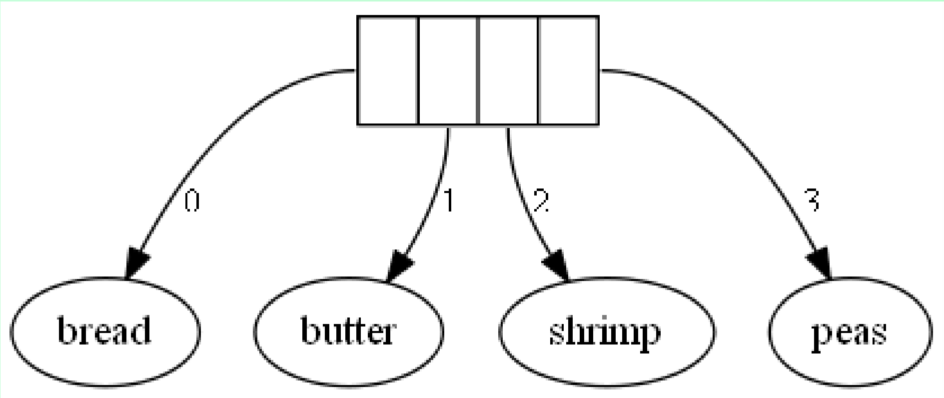
```
StackInterface<String> nameStack = new LinkedStack<String>();
while (!stringStack.isEmpty())
    nameStack.push(stringStack.pop());
```

b. `stringStack` is empty, and `nameStack` contains the strings that were in `stringStack` but in reverse order (*Jim* is at the top, and *Jill* is at the bottom).

Do Lab 2B
Practice Problems 1 & 2 Now

Need for Wrapper classes

- Bag or a Stack, List or Queue
  - all use an internal array of Object references.
- We can make Bags or Stacks of
  - String, Item, Name, Student, etc.



Need for Wrapper classes (2)

- But primitive collections are not allowed:
  - Bag<double>   Stack<char>
- To get around this, Java provides
  - Wrapper classes
    - Double, Integer, Boolean, Character, etc
    - These classes wrap primitive values in an object
    - Provide "autoboxing" and "auto-unboxing" to make them mostly the same as working with primitives

```
// A Double is an object that holds a double primitive valu
Double number = new Double(2.34);   // this is an object o
double value = 2.34;                // this is a primitive
```

```
value = number;   // autounboxing -- pulls primitive out of object
number = value;   // autoboxing -- puts primitive into object
```

Static Methods in Wrapper Classes

- The Wrapper classes provide a wide variety of "utility" methods for working with associated primitive counterparts.
  - Suppose we have
    - char symbol = 'x';
  - Character class has methods to tell if a char is a
    - alphabetical letter Character.isAlpha(symbol)
    - digit Character.isDigit(symbol)
  - Integer class has methods to convert String to int
    - Integer.parseInt("125"); Integer.parseInt("hello");

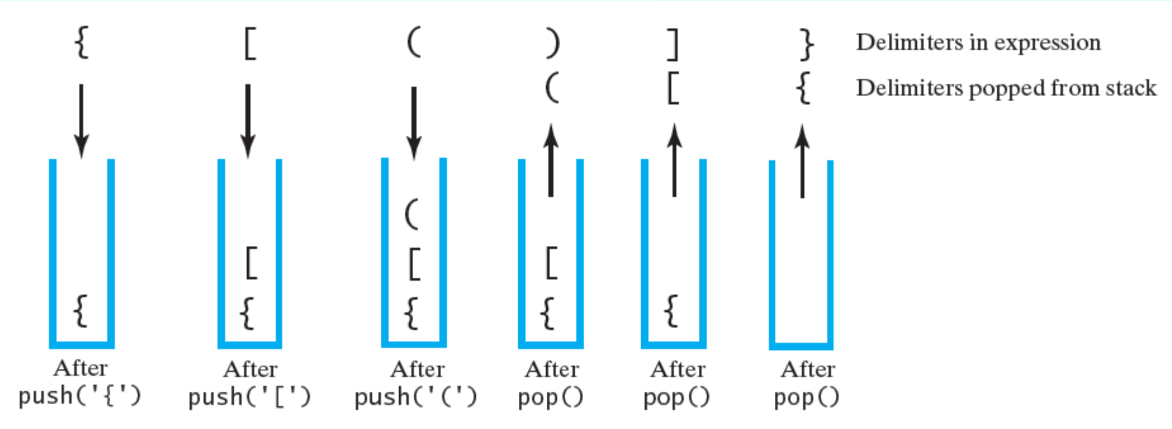
Demo this in lab now, Problems 3 and 4

Using a Stack to Process Algebraic Expressions

- Algebraic expressions composed of
  - Operands (variables, constants)
  - Operators (+, -, /, *, ^)
- Operators can be unary or binary
- Different precedence notations
  - Infix a + b
  - Prefix + a b
  - Postfix a b + ← we focus on this one

Using a Stack to Process Algebraic Expressions

- Precedence must be maintained
  - Order of operators
  - Use of parentheses (must be balanced)
- Use stacks to evaluate parentheses usage
  - Scan expression
  - Push symbols
  - Pop symbols



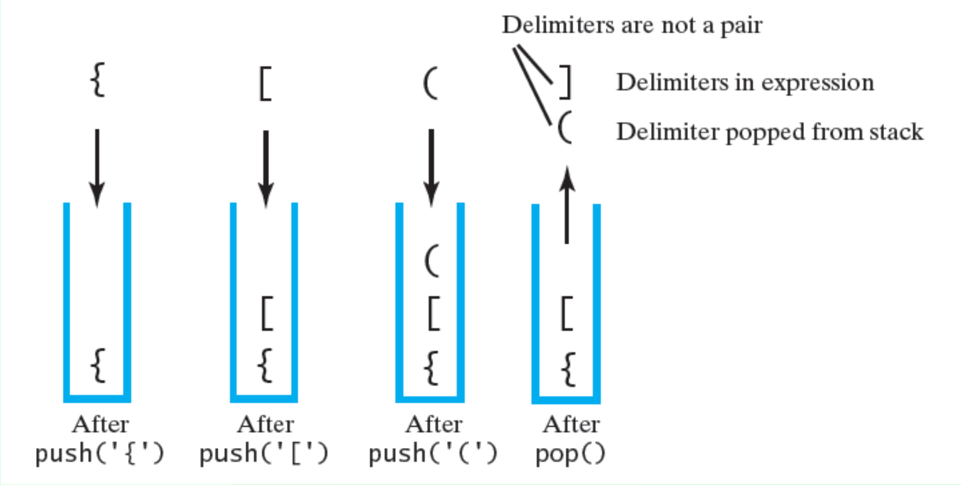


Figure 5-3 The contents of a stack during the scan of an expression that contains (a) balanced delimiters { [( )] } and (b) unbalanced delimiters { [( ]) }

Hardware-level Processing of Algebraic Expressions

Consider the arithmetic statement in the assignment statement:

x = a * b + c

Compiler must generate machine instructions

1. LOAD a
2. MULT b
3. ADD c
4. STORE x

Note: this is "infix" notation

The operators are between the operands

RPN or Postfix Notation

- Most compilers convert an expression in ***infix*** notation to ***postfix***
  - the operators are written after the operands
- So   a * b + c   becomes   a b * c +
- Advantage:
  - expressions can be written without parentheses

Postfix and Prefix Examples

INFIX	RPN (POSTFIX)	PREFIX
A + B	A B +	+ A B
A * B + C	A B * C +	+ * A B C
A * (B + C)	A B C + *	* A + B C
A - (B - (C - D))	A B C D---	-A-B-C D
A - B - C - D	A B-C-D-	---A B C D

Prefix : Operators come before the operands

Evaluating RPN Expressions

"By hand" (Underlining technique):

1. Scan the expression from left to right to find an operator.
2. Locate ("underline") the last two preceding operands and combine them using this operator.
3. Repeat until the end of the expression is reached.

Example:

```
     2 3 4 + 5 6 - - *
  →  2 3 4 + 5 6 - - *
  →  2 7 5 6 - - *
  →  2 7 5 6 - - *
  →  2 7 -1 - *
  →  2 7 -1 - *  →  2 8 *  →  2 8 *  →  16
```

Question 7 Using the previous algorithm, evaluate each of the following postfix expressions. Assume that a = 2, b = 3, c = 4, d = 5, and e = 6.

- **a.** $a\ e + b\ d - /$
- **b.** $a\ b\ c * d * -$
- **c.** $a\ b\ c - / d *$
- **d.** $e\ b\ c\ a\ \wedge * + d -$

7.

a. -4.

b. -58.

c. -10.

d. 49.

Evaluating RPN Expressions

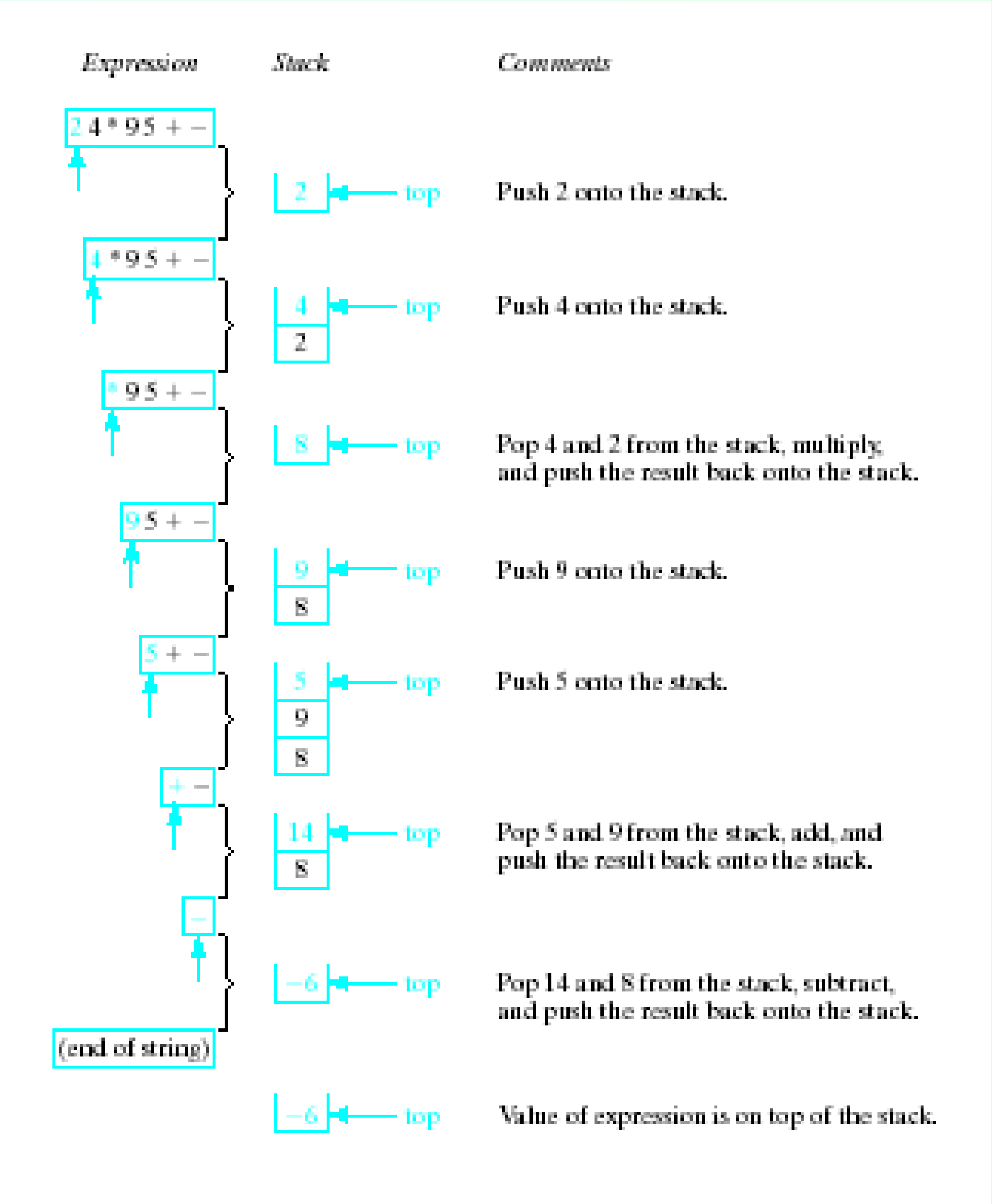
By using a stack algorithm

1. Initialize an empty stack
2. Repeat the following until the end of the expression is encountered
   a) Get the next token (const, var, operator) in the expression
   b) Operand – push onto stack
      Operator – do the following
      i. Pop 2 values from stack
      ii. Apply operator to the two values
      iii. Push resulting value back onto stack
3. When end of expression encountered, value of expression is the (only) number left in stack

Note: if only 1 value on stack, this is an invalid RPN expression

Evaluation of Postfix

- Note the changing status of the stack



Other uses of Stacks

- Converting infix expressions to postfix

```
A * B + C  →  ((A * B) + C)        A * (B + C)  →  (A * (B + C) )
           →  ((A B * C +                       →  (A (B C + *
           →  A B * C +                         →  A B C + *
```

The Program (Runtime) Stack

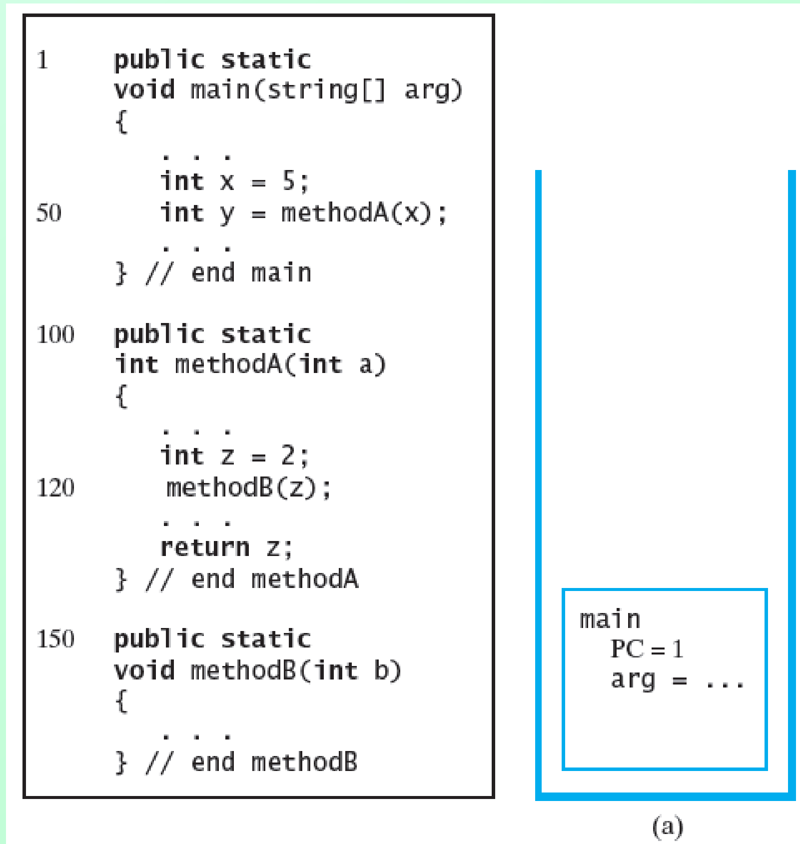
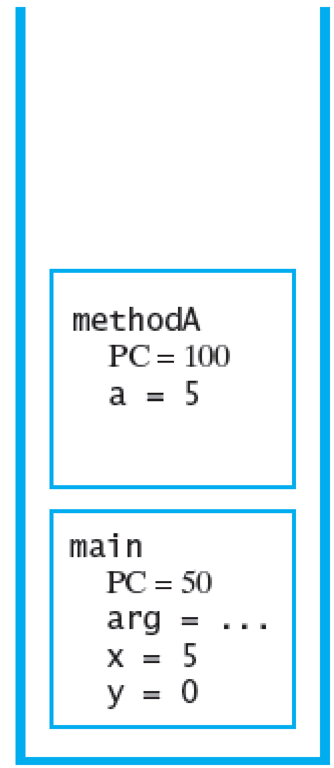


FIGURE 5-13 The program stack at three points in time:
(a) when **`main`** begins execution; (PC is the program counter)

The Program Stack

```
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
```

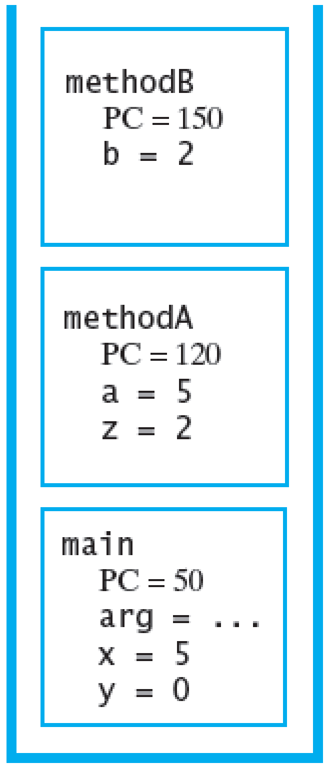


(b)

FIGURE 5-13 The program stack at three points in time:
(b) when **`methodA`** begins execution; (PC is the program counter)

The Program Stack

```
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
```



(c)

FIGURE 5-13 The program stack at three points in time:
(c) when **methodB** begins execution; (PC is the program counter)

Java Class Library: The Interface `Stack`

- Has a single constructor
  - Creates an empty stack
- Remaining methods – differences from our **`StackInterface`** are highlighted
  - `public T push(T item);`
  - `public T pop();`
  - `public T peek();`
  - `public boolean empty();`

End

Chapter 5