Northeastern University

Lecture 17: Stacks - 2

Prof. Chen-Hsiang (Jones) Yu, Ph.D.

College of Engineering

Materials are edited by Prof. Jones Yu from

Data Structures and Abstractions with Java, 5th edition. By Frank M. Carrano and Timothy M. Henry.
ISBN-13 978-0-13-483169-5

Transforming an Infix Expression to a Postfix Expression

Infix Expression

a + b

Postfix Expression

a b +

Converting Infix to Postfix

`a + b * c`

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

`a ^ b ^ c`

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^^
c	*a b c*	^^
	a b c ^	^
	a b c ^ ^	

`a - b + c`

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
−	*a*	−
b	*a b*	−
+	*a b* −	
	a b −	+
c	*a b* − *c*	+
	a b − *c* +	

Figure 5-7 & 5-8: Converting the infix expressions to postfix form

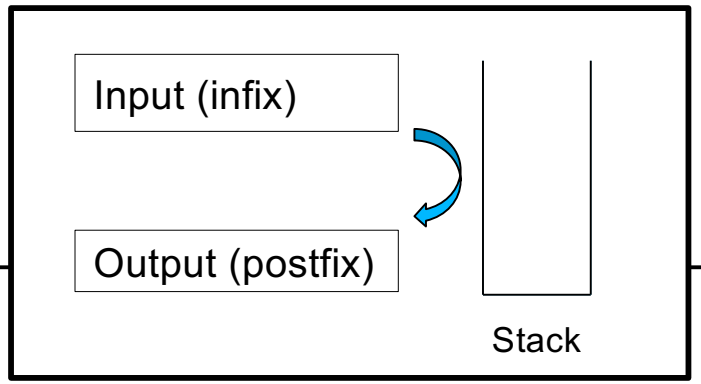
Converting Infix to Postfix

```
a / b * (c + (d - e))
```

Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b /*	
	a b /	*
(	*a b /*	* (
c	*a b / c*	* (
+	*a b / c*	* (+
(	*a b / c*	* (+ (
d	*a b / c d*	* (+ (
−	*a b / c d*	* (+ (−
e	*a b / c d e*	* (+ (−
)	*a b / c d e −*	* (+ (
	a b / c d e −	* (+
)	*a b / c d e − +*	* (
	a b / c d e − +	*
	*a b / c d e − + **	

Figure 5-9: Steps in converting an infix expression to postfix form

Infix-to-postfix Conversion



Operand	Append each operand to the end of the output expression.
Operator ^	Push ^ onto the stack.
Operator +, -, *, or /	Pop operators from the stack, appending them to the output expression, until either the stack is empty or its top entry has a lower precedence than the newly encountered operator. Then push the new operator onto the stack.
Open parenthesis	Push (onto the stack.
Close parenthesis	Pop operators from the stack and append them to the output expression until an open parenthesis is popped. Discard both parentheses.

Infix-to-postfix Algorithm - Part 1

```
Algorithm convertToPostfix(infix)
// Converts an infix expression to an equivalent postfix expression.

operatorStack = a new empty stack
postfix = a new empty string
while (infix has characters left to parse)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:   // operand
         Append nextCharacter to postfix
         break
      case '^' :
         operatorStack.push(nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
                precedence of nextCharacter <= precedence of operatorStack.peek())
         {
            Append operatorStack.peek() to postfix
            operatorStack.pop()
         }
         operatorStack.push(nextCharacter)
         break
```

Infix-to-postfix Algorithm - Part 2

```
        case '( ' :
            operatorStack.push(nextCharacter)
            break
        case ')' :
            // Stack is not empty if infix expression is valid
            topOperator = operatorStack.pop()
            while (topOperator != '(')
            {
               Append topOperator to postfix
               topOperator = operatorStack.pop()
            }
            break
        default:
            break // Ignore unexpected characters
    }
}

while (!operatorStack.isEmpty())
{
   topOperator = operatorStack.pop()
   Append topOperator to postfix
}

return postfix
```

Exercise

- Using above infix-to-postfix algorithm to represent following infix expression as a postfix expression:
  - » (a + b) / (c - d)
  - » a - (b / (c - d) * e + f) ^ g

Answer

- (a + b) / (c - d)
  - » Postfix: a b + c d - /
- a - (b / (c - d) * e + f) ^ g
  - » Postfix: a b c d – / e * f + g ^ -

Evaluating Postfix Expression

Evaluating Postfix Expressions

a b / when *a* is 2 and b is 4, i.e., 2 4 /

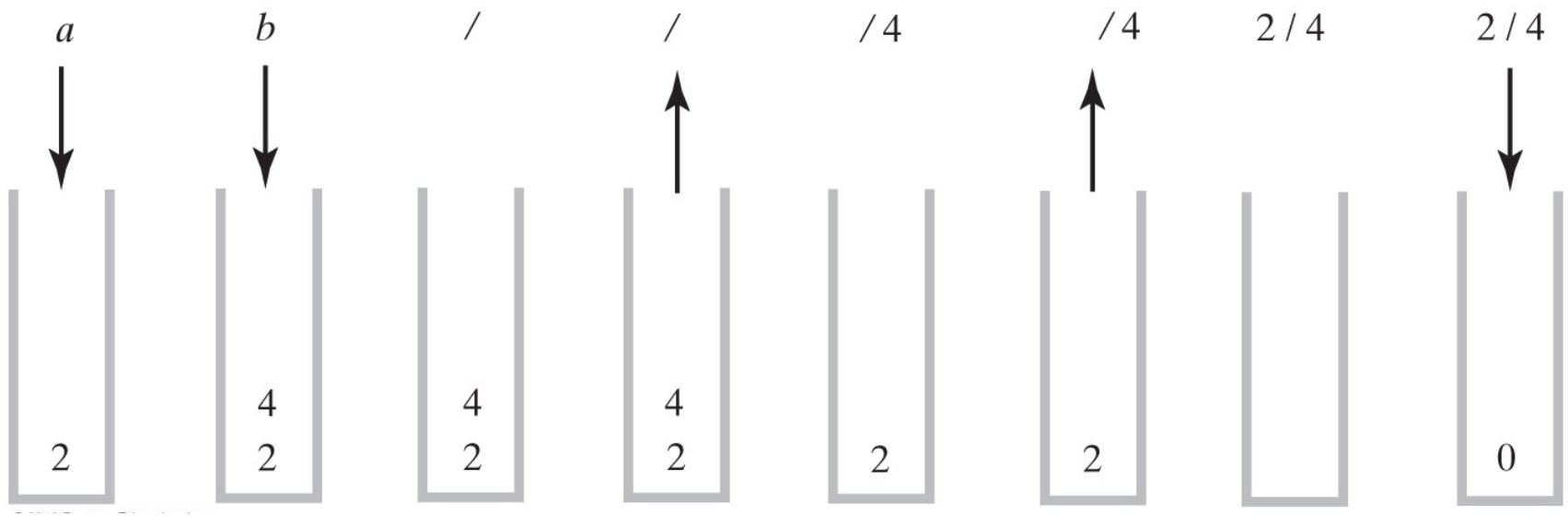


Figure 5-10: The stack during the evaluation of the postfix expression *a b /* when *a* is 2 and b is 4

Evaluating Postfix Expressions

a b + *c* / when *a* is 2, b is 4, and c is 3, i.e., 2 4 + 3 /

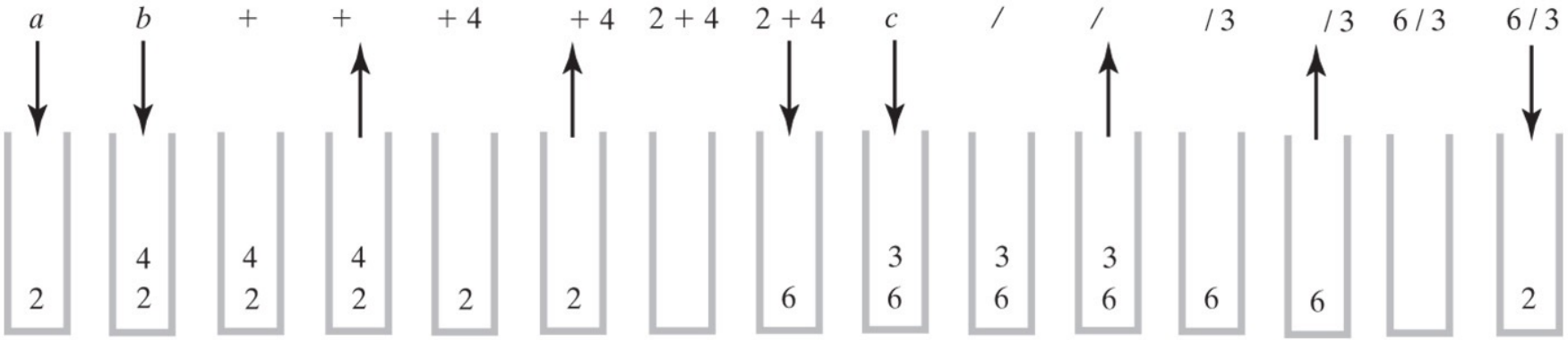


Figure 5-11: The stack during the evaluation of the postfix expression a b + c / when *a* is 2, *b* is 4, and *c* is 3

Evaluating Postfix Expressions

```
Algorithm 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 '^' :
         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
    }
}

return valueStack.peek()
```

Evaluating Infix Expression

Evaluating Infix Expressions

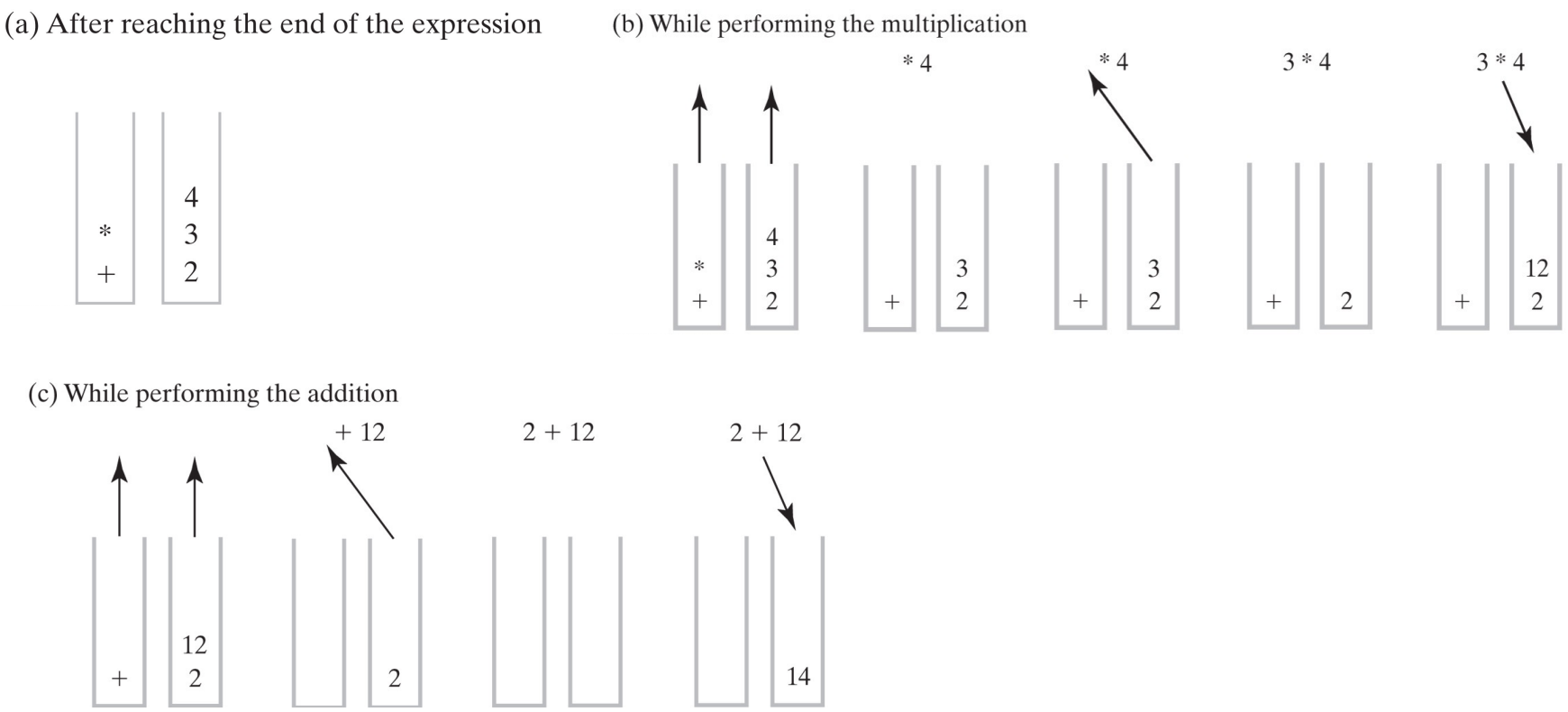
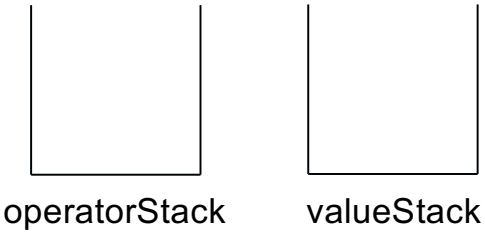


Figure 5-12: Two stacks during the evaluation of a + b * c when a is 2, b is 3, and c is 4

```
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
                     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
            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
         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 Program Stack

The Program Stack

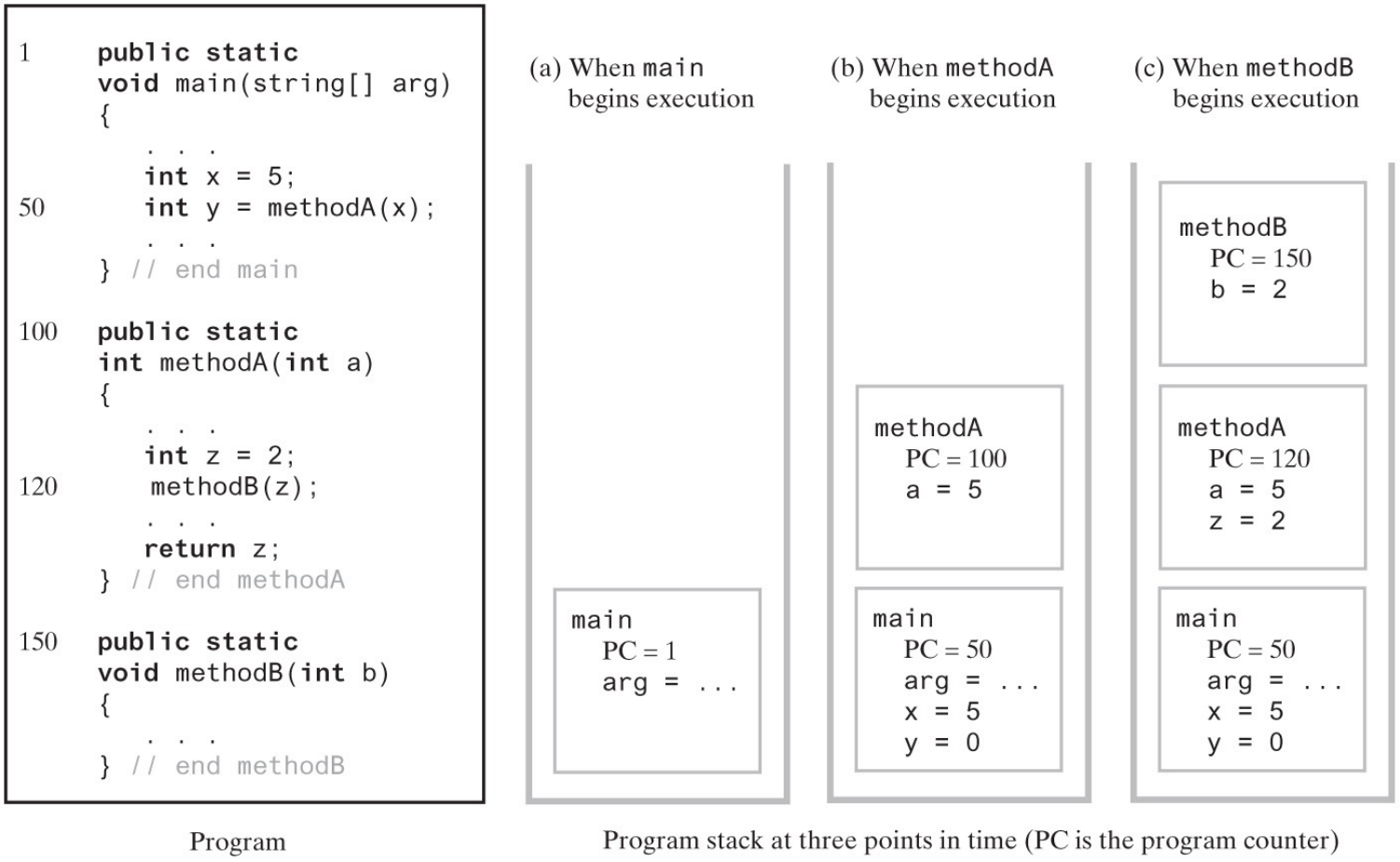
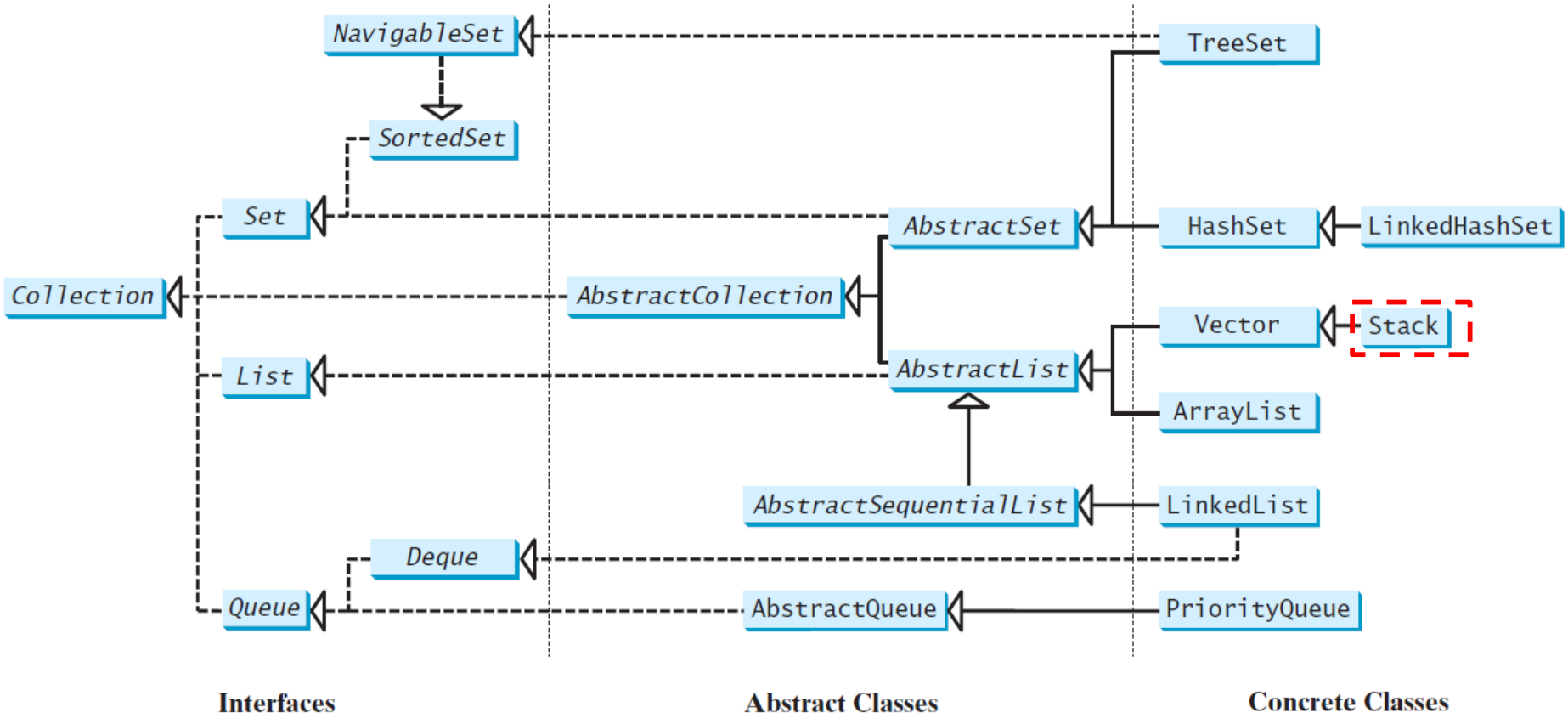


Figure 5-13: The program stack at three points in time: (a) when main begins execution; (b) when methodA begins execution; (c) when methodB begins execution

Java Class Library: The Class `Stack`

- Found in `java.util`
- Methods
  - A constructor – creates an empty stack
  - `public T push(T item);`
  - `public T pop();`
  - `public T peek();`
  - `public boolean empty();`

Java Collections Framework hierarchy



Source: Liang, Y. Daniel. Introduction to Java Programming, Comprehensive Version, 12th edition, Pearson, 2019.

Exercise

- A palindrome is a string of characters (a word, phrase, or sentence) that is the same regardless of whether you read it forward or backward -- assuming that you ignore spaces, punctuation, and case.
- For example, “Race car” and “Do geese see God?” are palindrome.
- Describe how you could use a stack to test whether a string is a palindrome.

Answer

- Let's define a character of interest as only those characters to be considered for the palindrome (e.g. not white space or punctuation).
- First compute the number of characters of interest in the string and call it N. Push the first $N/2$ characters of interest from the input onto the stack.
- If N is odd, discard the middle character (index = $N/2$) of interest from the input. Now compare the remaining $N/2$ characters of interest in the input with the characters popped from the stack.
- If we find a pair that is not the same (ignoring case), the string was not a palindrome; otherwise it was a palindrome.