

# Stacks

A stack is a list with the restriction that insertion and deletion can be performed at only one position, the **top** of the stack.

A stack can be compared to a Pez dispenser



- Only the top item can be accessed
- Can only extract one item at a time

A stack allows access only to the top element.

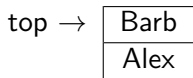
A stack is a **LIFO** (last in first out) list.

# Stack Operations

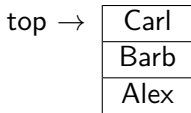
**Push:** inserts an item at the top position

**Pop:** removes the item at the top position

**Top** or **Peek:** returns the item at the top position



Pushing Carl



Popping an empty stack is an error.

Stack operations are fast (constant time) and powerful.

# Stack Interface

```
public interface StackInt < E > {  
    /** Pushes an item onto the top of the stack and returns  
        the item pushed.  
        @param obj The object to be inserted  
        @return The object inserted  
    */  
    E push(E obj);  
  
    /** Returns the object at the top of the stack  
        without removing it.  
        post: The stack remains unchanged.  
        @return The object at the top of the stack  
        @throws EmptyStackException if stack is empty  
    */  
    E peek();  
  
    /** Returns the object at the top of the stack  
        and removes it.  
        post: The stack is one item smaller.  
        @return The object at the top of the stack  
        @throws EmptyStackException if stack is empty  
    */  
    E pop();  
  
    /** Returns true if the stack is empty; otherwise,  
        returns false.  
        @return true if the stack is empty  
    */  
    boolean empty();  
}
```

We will study a few applications using stacks:

- Palindrome finder
- Balanced symbol checker
- Evaluating postfix expressions
- Converting infix to postfix

# Palindromes

A **palindrome** is a string that reads the same in either direction:

- Able was I ere I saw Elba
- A man, a plan, a canal: Panama
- Abba

How can we check if a string is a palindrome “without” a stack?

# Palindromes

A **palindrome** is a string that reads the same in either direction:

- Able was I ere I saw Elba
- A man, a plan, a canal: Panama
- Abba

How can we check if a string is a palindrome “without” a stack?

**One way is to use recursion.**

**Note that recursion uses an implicit stack.**

**Recursion can be removed by simulating it with a stack.**

**In this case we can use a stack directly.**

## Palindromes: coding details

See `PalindromeFinder.java` ...

# Testing for Balanced Parentheses

$(w * (x + y) / z - (p / (r - q)))$

$(w * [x + y] / z - [p / \{r - q\}])$

$(w * [x + y) / z - [p / \{r - q\}])$

How can we tell which expressions are syntactically correct?

# Think, then code

- Before writing even one line of code, we need a plan
- Algorithm for testing an expression = ?

# Testing for Balanced Parentheses: Coding Details

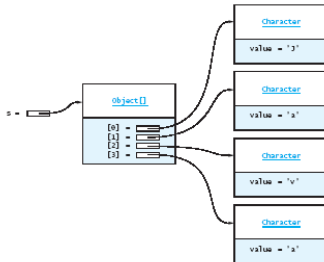
# Implementing a Stack

The Java API includes a Stack class as part of the package `java.util`

`Vector` is a growable array of objects

The elements of a `Vector` can be accessed using an integer

**FIGURE 5.3**  
Characters of "Java"  
stored in Stack s  
(a Vector)



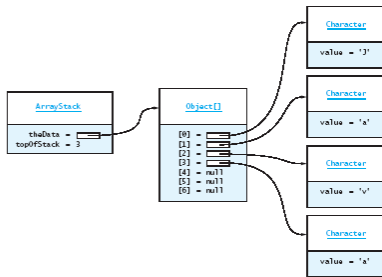
# Implementing a Stack: Array based

Need to allocate storage for an array with an initial default capacity when creating a new stack object

Need to keep track of the top of the stack

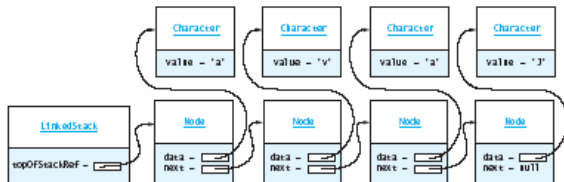
No size method is needed

**FIGURE 5.4**  
Stack of Character  
Objects in an Array



# Implementing a Stack: Linked List based

**FIGURE 5.5**  
Stack of Character  
Objects in a Linked List



# Comparison of Stack Implementations

Extending a `Vector` (as is done by Java) is a poor choice for stack implementation as all `Vector` methods are accessible

Easiest implementation would be to use an `ArrayList` component for storing data

All insertions and deletions are “constant time” regardless of the type of implementation

Linked List implementation is memory inefficient

# Evaluating postfix expressions

```
create an empty stack
while (more tokens exist)
  get next token
  if (token is an integer)
    push it
  else if (token is an operator)
    pop the right operand
    pop the left operand
    apply the operator
    push the result
  else
    unexpected input token!
pop stack and return value
```

Trace it: 12 20 + 2 15 \* 3 / -

# Converting infix to postfix

Given an infix expression with **operands** and **operators**, output an equivalent postfix expression.

Initially, assume no parentheses appear in the infix expression

Operands appear in the **same** order in both infix and postfix form  
Operators may need to be reordered in postfix form

a + b * c	becomes	a b c * +
a * b + c	becomes	a b * c +
a + b - c	becomes	a b + c -

How to determine the correct order and placement of the operators?

## Infix to postfix conversion: strategy

a + b \* c becomes a b c \* +  
a \* b + c becomes a b \* c +  
a + b - c becomes a b + c -

- Send operands straight to the output
- Hold pending operators on a stack
- Send higher precedence operators first
- For operators of equal precedence, send in left to right order

# Example

a + b - c \* d / e

token	action	stack	postfix
a	append	empty	a
+	push	+	a
b	append	+	a b
-	pop/append	empty	a b +
-	push	-	a b +
...	...	...	...

How do we handle the operators?

# Converting Infix to Postfix

Set postfix to be an empty string

Set operator\_stack to an empty stack

While more tokens in infix string

    Get the next token

    If the token is an operand

        Append the token to postfix

    Else if the token is an operator

        Call processOperator to handle it

    Else Indicate a syntax error process\_operator

Pop remaining operators and add them to postfix

## Converting Infix to Postfix: processOperator

```
if the operator stack is empty
  push operator
else
  let topOp be the operator on the top of the stack
  if the precedence of the operator is greater than topOp
    push operator
  else
    while stack is not empty and precedence is <= topOp
      pop topOp off the stack and append to postfix
      if the operator stack is not empty
        let topOp be the top operator
    push operator
```

## Generalize: allow parentheses within expressions

Rough idea:

- treat ( and ) as operators
- parentheses appear on stack, but are never added to the output
- both ( and ) get **lowest** precedence

See pages 294–295 and `InfixToPostfixParens.java`