STACKS

Presentation for use with the textbook
An abstract data type (ADT) is an abstraction of a data structure.

An ADT specifies:
- Data stored
- Operations on the data
- Error conditions associated with operations

Example: ADT modeling a simple stock trading system
- The data stored are buy/sell orders
- The operations supported are
  - order buy(stock, shares, price)
  - order sell(stock, shares, price)
  - void cancel(order)
- Error conditions:
  - Buy/sell a nonexistent stock
  - Cancel a nonexistent order
A stack is one of the most commonly used data structures in computer science.

A stack can be compared to a Pez dispenser:
- Only the top item can be accessed.
- You can extract only one item at a time.

The top element in the stack is the last added to the stack (most recently).

The stack’s storage policy is *Last-In, First-Out*, or *LIFO*.
Main stack operations:

- **push(object)**: inserts an element
- **object pop()**: removes and returns the last inserted element

Auxiliary stack operations:

- **object top()**: returns the last inserted element without removing it
- **integer size()**: returns the number of elements stored
- **boolean isEmpty()**: indicates whether no elements are stored
“Rich” is the oldest element on the stack and “Jonathan” is the youngest (Figure a)

String last = names.top(); stores a reference to “Jonathan” in last

String temp = names.pop(); removes “Jonathan” and stores a reference to it in temp (Figure b)

names.push(“Philip”); pushes “Philip” onto the stack (Figure c)
Java interface corresponding to our Stack ADT

- Assumes null is returned from top() and pop() when stack is empty
- Different from the built-in Java class java.util.Stack

<table>
<thead>
<tr>
<th>Our Stack ADT</th>
<th>Class java.util.Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>size()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>empty()</td>
</tr>
<tr>
<td>push(e)</td>
<td>push(e)</td>
</tr>
<tr>
<td>pop()</td>
<td>pop()</td>
</tr>
<tr>
<td>top()</td>
<td>peek()</td>
</tr>
</tbody>
</table>

```java
public interface Stack<E> {
    int size();
    boolean isEmpty();
    E top();
    void push(E element);
    E pop();
}
```
### ANOTHER EXAMPLE

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Value</th>
<th>Stack Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(5)</td>
<td>–</td>
<td>(5)</td>
</tr>
<tr>
<td>push(3)</td>
<td>–</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>size()</td>
<td>2</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>pop()</td>
<td>3</td>
<td>(5)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>false</td>
<td>(5)</td>
</tr>
<tr>
<td>pop()</td>
<td>5</td>
<td>()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>()</td>
</tr>
<tr>
<td>pop()</td>
<td>null</td>
<td>()</td>
</tr>
<tr>
<td>push(7)</td>
<td>–</td>
<td>(7)</td>
</tr>
<tr>
<td>push(9)</td>
<td>–</td>
<td>(7, 9)</td>
</tr>
<tr>
<td>top()</td>
<td>9</td>
<td>(7, 9)</td>
</tr>
<tr>
<td>push(4)</td>
<td>–</td>
<td>(7, 9, 4)</td>
</tr>
<tr>
<td>size()</td>
<td>3</td>
<td>(7, 9, 4)</td>
</tr>
<tr>
<td>pop()</td>
<td>4</td>
<td>(7, 9)</td>
</tr>
<tr>
<td>push(6)</td>
<td>–</td>
<td>(7, 9, 6)</td>
</tr>
<tr>
<td>push(8)</td>
<td>–</td>
<td>(7, 9, 6, 8)</td>
</tr>
<tr>
<td>pop()</td>
<td>8</td>
<td>(7, 9, 6)</td>
</tr>
</tbody>
</table>
Exceptions vs. Returning Null

- Attempting the execution of an operation of an ADT may sometimes cause an error condition.
- Java supports a general abstraction for errors, called exception.
- An exception is said to be “thrown” by an operation that cannot be properly executed.
- In our Stack ADT, we do not use exceptions.
- Instead, we allow operations `pop` and `top` to be performed even if the stack is empty.
- For an empty stack, `pop` and `top` simply return `null`.
APPLICATIONS OF STACKS

Direct applications
- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of method calls in the Java Virtual Machine

Indirect applications
- Auxiliary data structure for algorithms
- Component of other data structures
A simple way of implementing the Stack ADT uses an array.

We add elements from left to right.

A variable \( t \) keeps track of the index of the top element.

Algorithm \( \text{size}() \)

\[
\text{return } t + 1
\]

Algorithm \( \text{pop}() \)

\[
\text{if } \text{isEmpty}() \text{ then}\n\quad \text{return null}\n\text{else}\n\quad t \leftarrow t - 1\n\quad \text{return } S[t + 1]
\]
The array storing the stack elements may become full

A push operation will then throw a FullStackException

- Limitation of the array-based implementation
- Not intrinsic to the Stack ADT

Algorithm `push(o)`

```
if \( t = S.length - 1 \) then
    throw IllegalStateException
else
    \( t \leftarrow t + 1 \)
    \( S[t] \leftarrow o \)
```
IMPLEMENTING A STACK USING AN ARRAY

ArrayStack

theData = null
topOfStack = 3

Object[]

[0] = null
[1] = null
[2] = null
[3] = null
[4] = null
[5] = null
[6] = null
[7] = null
[8] = null
[9] = null

Character

value = 'J'

Character

value = 'a'

Character

value = 'v'

Character

value = 'a'
public class ArrayStack<E> implements Stack<E> {
    private int t = -1;  // index of the top element in stack
    private E answer = null;  // safe cast; compiler may give warning
    private int capacity = 1000;  // generic array used for storage
    private int size = 0;  // constructs stack with default capacity
    public ArrayStack() {  // Stack is full!
        this(CAPACITY);  // constructs stack with given capacity
    }
    public ArrayStack(int capacity) {  // index [0..capacity-1]
        data = (E[]) new Object[capacity];  // safe cast; compiler may give warning
        this(CAPACITY);  // constructs stack with given capacity
    }
    public void push(E e) {  // throws IllegalArgumentException if (size() == data.length) throw new IllegalStateException("Stack is full!");
        if (isFull()) {  // index [0..capacity-1]
            throw new StackOverflowError("Stack is full!");
        }
        data[++t] = e;
    }
    public int size() {  // returns number of items in stack
        return t + 1;
    }
    public boolean isEmpty() {  // returns true if stack is empty
        return t == -1;
    }
    public E pop() {  // returns top item in stack
        if (isEmpty()) {  // returns null if stack is empty
            return null;
        }
        t--;
        return data[t];
    }
    public E top() {  // returns top item in stack
        if (isEmpty()) {  // returns null if stack is empty
            return null;
        }
        return data[t];
    }
    public static final int CAPACITY = 1000;
    private static final int t = -1;
    private static final int data = (E[]) new Object[capacity];
}

// reference to help garbage collection
```java
// dereference to help garbage collection
```
public class ArrayStack<E> implements Stack<E> {

    // holds the stack elements
    private E[] S;

    // index to top element
    private int top = -1;

    // constructor
    public ArrayStack(int capacity) {
        S = (E[]) new Object[capacity];
    }

    public E pop() {
        if (isEmpty())
            return null;
        E temp = S[top];
        // facilitate garbage collection:
        S[top] = null;
        top = top - 1;
        return temp;
    }

    ... (other methods of Stack interface)
public class Tester {
    // … other methods
    public intReverse(Integer a[]) {
        Stack<Integer> s;
        s = new ArrayStack<Integer>();
        ... (code to reverse array a) ...
    }
}

public floatReverse(Float f[]) {
    Stack<Float> s;
    s = new ArrayStack<Float>();
    ... (code to reverse array f) ...
}
PERFORMANCE & LIMITATIONS OF ARRAY-BASED STACK

Performance
- Let $n$ be the number of elements in the stack
- The space used is $O(n)$
- Each operation runs in time $O(1)$

Limitations
- The maximum size of the stack must be defined a priori and cannot be changed (fixed size array)
- Trying to push a new element into a full stack causes an implementation-specific exception

<table>
<thead>
<tr>
<th>Method</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>top</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>push</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>pop</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
The linked-list approach has memory usage that is always proportional to the number of actual elements currently in the stack, and without an arbitrary capacity limit.

Q: What the best choice for the top of the stack: the front or back of the list?

With the top of the stack stored at the front of the list, all methods execute in constant time.
We can also implement a stack using a linked list of nodes. When the list is empty, pop returns null.
**THE ADAPTER DESIGN PATTERN**

- We want to effectively modify an existing class so that its methods match those of a related, but different, class or interface.

- Define a new class in such a way that it contains an instance of the existing class as a **hidden field**, and then to implement each method of the new class using methods of this hidden instance variable.
We will adapt SinglyLinkedList class of Section 3.2.1 to define a new LinkedStack class.

SinglyLinkedList is named `list` as a private field, and uses the following correspondences:

<table>
<thead>
<tr>
<th>Stack Method</th>
<th>Singly Linked List Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>list.size()</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>list.isEmpty()</td>
</tr>
<tr>
<td>push(e)</td>
<td>list.addFirst(e)</td>
</tr>
<tr>
<td>pop()</td>
<td>list.removeFirst()</td>
</tr>
<tr>
<td>top()</td>
<td>list.first()</td>
</tr>
</tbody>
</table>

```java
public class LinkedStack<E> implements Stack<E> {
    private SinglyLinkedList<E> list = new SinglyLinkedList<>();  // an empty list
    public LinkedStack() { }  // new stack relies on the initially empty list
    public int size() { return list.size(); }
    public boolean isEmpty() { return list.isEmpty(); }
    public void push(E element) { list.addFirst(element); }
    public E top() { return list.first(); }
    public E pop() { return list.removeFirst(); }
}
```
EXAMPLE: REVERSING AN ARRAY USING A STACK

A stack can be used as a general toll to reverse a data sequence.

```
1 /** A generic method for reversing an array. */
2 public static <E> void reverse(E[] a) {
3     Stack<E> buffer = new ArrayStack<>(a.length);
4     for (int i=0; i < a.length; i++)
5         buffer.push(a[i]);
6     for (int i=0; i < a.length; i++)
7         a[i] = buffer.pop();
8 }
```

The output from this method is the following:

$$\begin{align*}
\text{a} &= \{4, 8, 15, 16, 23, 42\} \\
\text{s} &= \text{[Jack, Kate, Hurley, Jin, Michael]} \\
\text{Reversing...} \\
\text{a} &= \{42, 23, 16, 15, 8, 4\} \\
\text{s} &= \text{[Michael, Jin, Hurley, Kate, Jack]} \\
\end{align*}$$
EXAMPLE: MATCHING PARENTHESES AND HTML TAGS

Consider arithmetic expressions that may contain various pairs of grouping symbols:

- Parentheses: “(” and “)”
- Braces: “{” and “}”
- Brackets: “[” and “]”

\[ (5+x)-(y+z) \]
Expression: \((w * [x + y] / z)\)

* balanced: true
* index: 0
Expression: \((w * [x + y] / z)\)

Balanced: true
Index: 1
Expression: \((w * [x + y] / z)\)

Balanced: true

Index: 2
Expression: \((w * [x + y] / z)\)

balanced : true
index : 3
Expression: \((w * [x + y] / z)\)

balanced : true
index : 4
Expression: \((w \times [x + y] / z)\)

balanced : true
index : 5
Expression: \((w \times [x + y] / z)\)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

balanced : true
index : 6
Expression: \((w * [x + y] / z)\)

```
0 1 2 3 4 5 6 7 8 9 10
( w * [ x + y ] / z )
```

Matches!
Balanced still true

balanced : true
index : 7
Expression: \((w * [x + y] / z)\)

Balanced: true
Index: 8
Expression: \((w * [x + y] / z)\)

**balanced**: true

**index**: 9
Expression: \((w \times [x + y] / z)\)

balanced : true
index : 10

Matches! Balanced still true
/** Tests if delimiters in the given expression are properly matched. */

public static boolean isMatched(String expression) {
    final String opening = "(["; // opening delimiters
    final String closing = "]\);"; // respective closing delimiters
    Stack<Character> buffer = new LinkedStack<>();
    for (char c : expression.toCharArray()) {
        if (opening.indexOf(c) != -1) // this is a left delimiter
            buffer.push(c);
        else if (closing.indexOf(c) != -1) {
            // this is a right delimiter
            if (buffer.isEmpty()) // nothing to match with
                return false;
            if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
                return false; // mismatched delimiter
        }
    }
    return buffer.isEmpty(); // were all opening delimiters matched?
The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?
/** Tests if every opening tag has a matching closing tag in HTML string. */

public static boolean isHTMLMatched(String html) {
    Stack<String> buffer = new LinkedList<>();
    int j = html.indexOf(' '<);
    while (j != -1) {
        int k = html.indexOf('>', j+1);
        if (k == -1) { // invalid tag
            return false;
        }
        String tag = html.substring(j+1, k);
        if (!tag.startsWith("/")) { // strip away <>
            buffer.push(tag);
            j = html.indexOf(' '<, k+1);
        } else { // this is a closing tag
            if (buffer.isEmpty()) { // no tag to match
                return false;
            }
            if (!tag.substring(1).equals(buffer.pop())) { // mismatched tag
                return false;
            }
        }
        return buffer.isEmpty(); // were all opening tags matched?
    }
}
Additional Stack Applications

- Postfix and infix notation
  - Expressions normally are written in **infix** form, but
  - it easier to evaluate an expression in **postfix** form since there is no need to group sub-expressions in parentheses or worry about operator precedence

<table>
<thead>
<tr>
<th>Postfix Expression</th>
<th>Infix Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 7 *</td>
<td>4 * 7</td>
<td>28</td>
</tr>
<tr>
<td>4 7 2 + *</td>
<td>4 * (7 + 2)</td>
<td>36</td>
</tr>
<tr>
<td>4 7 * 20 -</td>
<td>(4 * 7) - 20</td>
<td>8</td>
</tr>
<tr>
<td>3 4 7 * 2 / +</td>
<td>3 + ((4 * 7) / 2)</td>
<td>17</td>
</tr>
</tbody>
</table>
EVALUATING POSTFIX EXPRESSIONS

- Write a class that evaluates a postfix expression
- Use the space character as a delimiter between tokens

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack&lt;Integer&gt;</td>
<td>operandStack</td>
</tr>
<tr>
<td>public int eval</td>
<td>String expression)</td>
</tr>
<tr>
<td>private int evalOp</td>
<td>(char op)  Returns the value of expression.</td>
</tr>
<tr>
<td>private boolean isOperator</td>
<td>char ch) Returns true if ch is an operator symbol.</td>
</tr>
</tbody>
</table>

EVALUATING POSTFIX EXPRESSIONS (CONT.)

1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result
1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result
1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result
EVALUATING POSTFIX EXPRESSIONS (CONT.)

1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
   5. push the token on the stack
5. else if the token is an operator
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result

28

\[
\begin{array}{cccc}
4 & 7 & * & 20 & - \\
\end{array}
\]
1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result
1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result
1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result
1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result
CONVERTING FROM INFIX TO POSTFIX

- Convert infix expressions to postfix expressions
- Assume:
  + Expressions consists of only spaces, operands, and operators
  + Space is a delimiter character
  + All operands that are identifiers begin with a letter or underscore
  + All operands that are numbers begin with a digit

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>private Stack&lt;Character&gt; operatorStack</td>
<td>Stack of operators.</td>
</tr>
<tr>
<td>private StringBuilder postfix</td>
<td>The postfix string being formed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>public String convert(String infix)</td>
<td>Extracts and processes each token in infix and returns the equivalent postfix string.</td>
</tr>
<tr>
<td>private void processOperator(char op)</td>
<td>Processes operator op by updating operatorStack.</td>
</tr>
<tr>
<td>private int precedence(char op)</td>
<td>Returns the precedence of operator op.</td>
</tr>
<tr>
<td>private boolean isOperator(char ch)</td>
<td>Returns true if ch is an operator symbol.</td>
</tr>
</tbody>
</table>
Example: convert
\[ w - 5.1 / \text{sum} * 2 \]
to its postfix form
\[ w \ 5.1 \ \text{sum} \ / \ 2 \ * \ - \]
CONVERTING FROM INFIX TO POSTFIX (CONT.)

Algorithm for Method convert

1. Initialize postfix to an empty StringBuilder.
2. Initialize the operator stack to an empty stack.
3. while there are more tokens in the infix string
4. Get the next token.
5. if the next token is an operand
6. Append it to postfix.
7. else if the next token is an operator
8. Call processOperator to process the operator.
9. else
10. Indicate a syntax error.
11. Pop remaining operators off the operator stack and append them to postfix.
Algorithm for Method `processOperator`

1. if the operator stack is empty
   2. Push the current operator onto the stack.
   else
   3. Peek the operator stack and let `top0p` be the top operator.
   4. if the precedence of the current operator is greater than the precedence of `top0p`
      5. Push the current operator onto the stack.
   else
   6. while the stack is not empty and the precedence of the current operator is less than or equal to the precedence of `top0p`
      7. Pop `top0p` off the stack and append it to `postfix`.
   8. if the operator stack is not empty
      9. Peek the operator stack and let `top0p` be the top operator.
   10. Push the current operator onto the stack.
**CONVERTING FROM INFIX TO POSTFIX (CONT.)**

**INPUT:** $w - 5.1 / \text{sum} \times 2$

<table>
<thead>
<tr>
<th>Next Token</th>
<th>Action</th>
<th>Effect on operatorStack</th>
<th>Effect on postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w$</td>
<td>Append $w$ to postfix.</td>
<td></td>
<td>$w$</td>
</tr>
<tr>
<td>$-$</td>
<td>The stack is empty Push $-$ onto the stack</td>
<td>$-$</td>
<td>$w$</td>
</tr>
<tr>
<td>$5.1$</td>
<td>Append $5.1$ to postfix</td>
<td></td>
<td>$w$ $5.1$</td>
</tr>
<tr>
<td>$/$</td>
<td>precedence($/$) &gt; precedence($-$), Push $/$ onto the stack</td>
<td>$/$ $-$</td>
<td>$w$ $5.1$</td>
</tr>
<tr>
<td>$\text{sum}$</td>
<td>Append $\text{sum}$ to postfix</td>
<td></td>
<td>$w$ $5.1$ sum</td>
</tr>
<tr>
<td>$*$</td>
<td>precedence($*$) equals precedence($/$) Pop $/$ off of stack and append to postfix</td>
<td></td>
<td>$w$ $5.1$ sum /</td>
</tr>
</tbody>
</table>
## Converting from Infix to Postfix (Cont.)

<table>
<thead>
<tr>
<th>Next Token</th>
<th>Action</th>
<th>Effect on operatorStack</th>
<th>Effect on postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>precedence(*) &gt; precedence(-), Push * onto the stack</td>
<td>[*]</td>
<td>w 5.1 sum /</td>
</tr>
<tr>
<td>2</td>
<td>Append 2 to postfix</td>
<td>[-]</td>
<td>w 5.1 sum / / 2</td>
</tr>
<tr>
<td>End of input</td>
<td>Stack is not empty, Pop * off the stack and append to postfix</td>
<td>[-]</td>
<td>w 5.1 sum / 2 *</td>
</tr>
<tr>
<td>End of input</td>
<td>Stack is not empty, Pop - off the stack and append to postfix</td>
<td>[-]</td>
<td>w 5.1 sum / 2 * -</td>
</tr>
</tbody>
</table>
The ability to convert expressions with parentheses is an important (and necessary) addition.

Modify `processOperator` to push each opening parenthesis onto the stack as soon as it is scanned.

When a closing parenthesis is encountered, pop off operators until the opening parenthesis is encountered.