What is a stack?

- Some familiar stacks
- Add item on top of stack
- Remove item that is topmost
  - Last In, First Out … LIFO
Specifications of the ADT Stack

<table>
<thead>
<tr>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A collection of objects in reverse chronological order and having the same data type</td>
</tr>
</tbody>
</table>

| OPERATIONS |
|-----------------|-----------------|-----------------|
| **PSEUDOCODE** | **UML**         | **DESCRIPTION** |
| pop()            | +pop(): T        | Task: Removes and returns the stack's top entry. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty before the operation. |

Specifications of the ADT Stack

| OPERATIONS |
|-----------------|-----------------|-----------------|
| peek()          | +peek(): T      | Task: Retrieves the stack's top entry without changing the stack in any way. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty. |
| isEmpty()       | +isEmpty(); boolean | Task: Detects whether the stack is empty. Input: None. Output: Returns true if the stack is empty. |
| clear()         | +clear(); void  | Task: Removes all entries from the stack. Input: None. Output: None. |
Design Decision

• When stack is empty
  – What to do with \texttt{pop} and \texttt{peek}?  
• Possible actions
  – Assume that the ADT is not empty;
  – Return null.
  – Throw an exception (which type?).

The \textbf{Stack} Interface

\texttt{// An interface for the ADT stack.}

\texttt{public interface StackInterface<T>}
\texttt{
{
  // push() – Adds a new entry to the top of
  // this stack.
  // The parameter newEntry An object
  // to be added to the stack.
  public void push(T newEntry);
}}
// pop() - Removes and returns this stack's
top entry.
// Returns the object at the top of
// the stack
// Throws EmptyStackException if the
// stack is empty before the
// operation.
public T pop();

// peek() - Retrieves this stack's top entry.
// Returns the object at the top of
// the stack.
// Throws EmptyStackException if the
// stack is empty. */
public T peek();

// empty() - Detects whether this stack is
// empty.
// Returns true if the stack is
// empty
public boolean isEmpty();

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

(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

Security Note

• Design guidelines
  – Use preconditions and postconditions to document assumptions.
  – Do not trust client to use public methods correctly.
  – Avoid ambiguous return values.
  – Prefer throwing exceptions instead of returning values to signal problem.
Processing Algebraic Expressions

- **Infix**: each binary operator appears between its operands \( a + b \)
- **Prefix**: each binary operator appears before its operands \( + a b \)
- **Postfix**: each binary operator appears after its operands \( a b + \)
- **Balanced expressions**: delimiters paired correctly

Scanning an expression that contains the balanced delimiters
\{ [ ( ) ] \}
Processing Algebraic Expressions

Scanning an expression that contains the balanced delimiters 
{ [ ( ) ] 

Processing Algebraic Expressions

Scanning an expression that contains the balanced delimiters 
[ ( ) ]

Stack is empty when 
) is encountered
Processing Algebraic Expressions

Scanning an expression that contains the balanced delimiters \{[]\}

Algorithm for Processing Algebraic Expressions

```java
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.

isBalanced = true
while ((isBalanced == true) and not at end of expression)
{
    nextCharacter = next character in expression
    switch (nextCharacter)
    {
        case '(' : case '[' : case '{' :
            Push nextCharacter onto stack
            break
        case ')' : case ']' : case '}' :
            if (stack is empty)
                isBalanced = false
            else
                isBalanced = true
    }
}
Algorithm for Processing Algebraic Expressions

BalanceChecker()

// A class that checks whether the parentheses, brackets, and braces in a string occur in left/right pairs.

public class BalanceChecker {
    // Decides whether the parentheses, brackets, and braces in a string occur in left/right pairs.
    // Returns true if the delimiters are paired correctly
    public static boolean checkBalance(String expression) {
        StacInterface<Character> openDelimiterStack = new ArrayStack<>();
        // Code to check balance...
    }
}
int characterCount = expression.length();
boolean isBalanced = true;
int index = 0;
char nextCharacter = ' ';

while (isBalanced &&
    (index < characterCount)) {
    nextCharacter = expression.charAt(index);
    switch (nextCharacter) {
        case '(' | case '[' | case '{':
            openDelimiterStack.push(nextCharacter);
            break;
        case ')': case ']': case '}':
            if (openDelimiterStack.isEmpty())
                isBalanced = false;
            else {
                char openDelimiter = openDelimiterStack.pop();
                isBalanced = isPaired(openDelimiter, nextCharacter);
            }
            break;
        // Ignore unexpected characters
        default: break;
    }
    index++;
}

// Ignore unexpected characters
default: break;
}

index++;
if (!openDelimiterStack.isEmpty())
    isBalanced = false;

return isBalanced;
}

// Returns true if the given characters, open and
// close, form a pair of parentheses, brackets,
// or braces.
private static boolean isPaired(char open,
        char close) {
    return (open == '(' && close == ')') ||
            (open == '[' && close == ']') ||
            (open == '{' && close == '}');
}
}
Converting $a + b * c$ Postfix

<table>
<thead>
<tr>
<th>Next Character in Infix Expression</th>
<th>Postfix Form</th>
<th>Operator Stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$a$</td>
<td>+</td>
</tr>
<tr>
<td>$+$</td>
<td>$a$</td>
<td>$+$</td>
</tr>
<tr>
<td>$b$</td>
<td>$a b$</td>
<td>$+$</td>
</tr>
<tr>
<td>$*$</td>
<td>$a b$</td>
<td>$+$</td>
</tr>
<tr>
<td>$c$</td>
<td>$a b c$</td>
<td>$+$</td>
</tr>
<tr>
<td></td>
<td>$a b c *$</td>
<td>$+$</td>
</tr>
<tr>
<td></td>
<td>$a b c * +$</td>
<td>$+$</td>
</tr>
</tbody>
</table>

Converting $a - b + c$ Postfix

<table>
<thead>
<tr>
<th>Next Character in Infix Expression</th>
<th>Postfix Form</th>
<th>Operator Stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$a$</td>
<td>$-$</td>
</tr>
<tr>
<td>$-$</td>
<td>$a$</td>
<td>$-$</td>
</tr>
<tr>
<td>$b$</td>
<td>$a b$</td>
<td>$-$</td>
</tr>
<tr>
<td>$+$</td>
<td>$a b -$</td>
<td>$+$</td>
</tr>
<tr>
<td>$c$</td>
<td>$a b - c$</td>
<td>$+$</td>
</tr>
<tr>
<td></td>
<td>$a b - c +$</td>
<td>$+$</td>
</tr>
</tbody>
</table>
Converting $a \ ^\ ^\ ^b \ ^\ ^\ ^c$ Postfix

<table>
<thead>
<tr>
<th>Next Character in Infix Expression</th>
<th>Postfix Form</th>
<th>Operator Stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$a$</td>
<td>$\land$</td>
</tr>
<tr>
<td>$\land$</td>
<td>$a \ b$</td>
<td>$\land$</td>
</tr>
<tr>
<td>$b$</td>
<td>$a \ b$</td>
<td>$\land\land$</td>
</tr>
<tr>
<td>$\land$</td>
<td>$a \ b \ c$</td>
<td>$\land\land\land\land$</td>
</tr>
<tr>
<td>$c$</td>
<td>$a \ b \ c \ ^\ ^\ ^$</td>
<td>$\land\land\land\land\land$</td>
</tr>
</tbody>
</table>

Infix to Postfix Conversion

- **Operand**: Append each operand to the end of the output expression.
- **Operator $\land$**: Push $\land$ onto the stack.
- **Operator $\times$, $\div$, or $/$**: Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new 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 Conversion Algorithm

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:
            Append nextCharacter to postfix
            break
        case '+':
            operatorStack.push(nextCharacter)
            break
        case '-':
            operatorStack.push(nextCharacter)
            break
        case '*':
            operatorStack.push(nextCharacter)
            break
        case '/':
            operatorStack.push(nextCharacter)
            break
        case '(':
            operatorStack.push(nextCharacter)
            break
        case ')': // Stack is not empty if infix expression is valid
            topOperator = operatorStack.pop()
            while (topOperator != '(')
                Append topOperator to postfix
                operatorStack.pop()
                topOperator = operatorStack.pop()
Infix to Postfix Conversion Algorithm

Converting \( \frac{a}{b} \times (c + (d - e)) \) to Postfix

<table>
<thead>
<tr>
<th>Next Character from Infix Expression</th>
<th>Postfix Form</th>
<th>Operator Stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>/</td>
</tr>
<tr>
<td>/</td>
<td>ab</td>
<td>/</td>
</tr>
<tr>
<td>b</td>
<td>ab</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>ab/c</td>
<td>*</td>
</tr>
<tr>
<td>(</td>
<td>ab/c</td>
<td>*(</td>
</tr>
<tr>
<td>)</td>
<td>ab/cd</td>
<td>*(+</td>
</tr>
<tr>
<td>d</td>
<td>ab/cd</td>
<td>*(+-</td>
</tr>
<tr>
<td>e</td>
<td>ab/cde-</td>
<td>*(+-</td>
</tr>
<tr>
<td>)</td>
<td>ab/cde-+</td>
<td>*(+</td>
</tr>
<tr>
<td>)</td>
<td>ab/cde-*</td>
<td>*(+</td>
</tr>
<tr>
<td></td>
<td>ab/cde-*+</td>
<td>*(</td>
</tr>
<tr>
<td></td>
<td>ab/cde-*+</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>ab/cde-*+</td>
<td></td>
</tr>
</tbody>
</table>
Evaluating Postfix Expressions

Evaluating \( a \ b \ / \) where \( a = 4 \) and \( b = 2 \)

Evaluating Postfix Expressions

Evaluating \( a \ b \ + \ c \ / \) when \( a = 2 \), \( b = 4 \), and \( c = 3 \)
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 '^';
        case 'A';;
            break
        case 'I';
            break
        default: break // Ignore unexpected characters
    }
}
```
Evaluating $a + b \times c$ when $a$ is 2, $b$ is 3, and $c$ is 4

Performing the Multiplication
Evaluating \(a + b \times c -\)

Performing the Addition

Algorithm for Evaluating infix Expressions

```plaintext
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 '*':
            operatorStack.push(nextCharacter)
            break
        case '+':
            operatorStack.push(nextCharacter)
            break
        case '-':
            operatorStack.push(nextCharacter)
            break
        case '/':
            operatorStack.push(nextCharacter)
            break
        default:
            valueStack.push(nextCharacter)
            break
    }
}```
Algorithm for Evaluating infix Expressions

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
Algorithm for Evaluating infix Expressions

```java
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

(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();`

**Linked Implementation**

- Each operation involves top of stack
  - `push`
  - `pop`
  - `peek`
- Head of linked list easiest, fastest to access
  - Let this be the top of the stack
Linked Implementation

A chain of linked nodes that implements a stack

// A class of stacks whose entries are stored in
// a chain of nodes.

public final class LinkedStack<T>
    implements StackInterface<T> {
    // References the first node in the chain
    private Node topNode;

    public LinkedStack() {
        topNode = null;
    }
    // Stack operations go here
private class Node {
    private T data; // Entry in stack
    private Node next; // Link to next node

    // Constructors, accessors and mutators go
    // here
}

Linked Implementation

A new node that references the node at the top of the stack
The new node is now at the top of the stack

```
push()

public void push(T newEntry) {  
    Node newNode = new Node(newEntry, topNode);  
    topNode = newNode;  
}
```
Linked Implementation

The stack before the first node in the chain is deleted

![Diagram showing the stack before the first node in the chain is deleted]

Linked Implementation

The stack after the first node in the chain is deleted

![Diagram showing the stack after the first node in the chain is deleted]
pop()

public T pop() {
    // Might throw EmptyStackException
    T top = peek();

    assert (topNode != null);
    topNode = topNode.getNextNode();
    return top;
}

isEmpty() and clear()

public boolean isEmpty() {
    return topNode == null;
}

public void clear() {
    // Causes deallocation of nodes in the chain
    topNode = null;
}
Array-Based Implementation

• Each operation involves top of stack
  – push
  – pop
  – peek
• End of the array easiest to access
  – Let this be top of stack
  – Let first entry be bottom of stack

Array Implementation

An array that implements a stack; its first location references the top entry in the stack;
Array Implementation

An array that implements a stack; its first location references the bottom entry in the stack

Array Implementation

```java
import java.util.Arrays;
import java.util.EmptyStackException;
// A class of stacks whose entries are stored in an array.

public final class ArrayStack<T> implements StackInterface<T> {
    // Array of stack entries
    private T[] stack;

    // Index of top entry
    private int topIndex;
    private boolean initialized = false;
```
private static final int DEFAULT_CAPACITY = 50;
private static final int MAX_CAPACITY = 10000;

public ArrayStack() {
    this(DEFAULT_CAPACITY);
}

public ArrayStack(int initialCapacity) {
    checkCapacity(initialCapacity);

    // The cast is safe because the new array
    // contains null entries
    @SuppressWarnings("unchecked")
    T[] tempStack
        = (T[]) new Object[initialCapacity];

    stack = tempStack;
    topIndex = -1;
    initialized = true;
}

// Implementations for stack operations and
// checkCapacity and checkInitialization go
// here
Adding to the top - **push()**

```java
public void push(T newEntry) {
    checkInitialization();
    ensureCapacity();
    stack[topIndex + 1] = newEntry;
    topIndex++;
}
```

Adding to the top - **checkCapacity()**

```java
private void checkCapacity(int capacity) {
    if (capacity > MAX_CAPACITY)
        throw new IllegalStateException
            ("Attempt to create a stack "+ "whose capacity exceeds "+ "allowed maximum.");
}
```
Array-Based Implementation

Retrieving the top, the operation is $O(1)$

```java
public T peek()
{
    checkInitialization();
    if (isEmpty())
        throw new EmptyStackException();
    else
        return stack[topIndex];
} // end peek
```

Array-Based Implementation

An array-based stack after its top entry is removed by decrementing `topIndex`:
Array-Based Implementation

An array-based stack after its top entry is removed by setting \texttt{stack[topIndex]} to \texttt{null} and then decrementing \texttt{topIndex}.

Array-Based Implementation

Removing the top

```java
public T pop()
{
    checkInitialization();
    if (isEmpty())
        throw new EmptyStackException();
    else
    {
        T top = stack[topIndex];
        stack[topIndex] = null;
        topIndex--;
        return top;
    } // end if
} // end pop
```
Vector Based Implementation

• Vector: an object that behaves like a high-level array
  – Index begins with 0
  – Methods to access or set entries
  – Size will grow as needed
• Use vector’s methods to manipulate stack

A client using the methods given in StackInterface; these methods interact with a vector’s methods to perform stack operations

Implementation of a stack
The Class `Vector`

- Constructors
- Has methods to add, remove, clear
- Also methods to determine
  - Last element
  - Is the vector empty
  - Number of entries

Vector Implementation

```java
import java.util.Vector;
import java.util.EmptyStackException;

// A class of stacks whose entries are stored
// in a vector.
public final class VectorStack<T>
    implements StackInterface<T> {
    // Last element is the top entry in stack
    private Vector<T> stack;
    private boolean initialized = false;
    private static final int DEFAULT_CAPACITY = 50;
    private static final int MAX_CAPACITY = 10000;
```
public VectorStack() {
    this(DEFAULT_CAPACITY);
}

public VectorStack(int initialCapacity) {
    checkCapacity(initialCapacity);

    // Size doubles as needed
    stack = new Vector<>(initialCapacity);
    initialized = true;
}

// Implementations of stack operations and
// checkInitialization and checkCapacity go
// here

public void push(T newEntry) {
    checkInitialization();
    stack.add(newEntry);
}
**peek()**

*Retrieving the top*

```java
public T peek() {
    checkInitialization();
    if (isEmpty())
        throw new EmptyStackException();
    else
        return stack.lastElement();
}
```

**pop()**

*Removing the top*

```java
public T pop() {
    checkInitialization();
    if (isEmpty())
        throw new EmptyStackException();
    else
        return stack.remove(stack.size() - 1);
}
```
Rest of the VectorStack Class

```java
public boolean isEmpty() {
    return stack.isEmpty();
}

public void clear() {
    stack.clear();
}
```