Stacks and their Implementation

Stacks and Queues as Collections

- A stack
  - Last-in, first-out (LIFO) property
    - The last item placed on the stack will be the first item removed
  - Analogy
    - A stack of dishes in a cafeteria
- vs: A queue
  - First in, first out (FIFO) property
    - The first item added is the first item to be removed
  - Analogy
    - A queue of train commuters
The Contract for the Stack Collection

Stack operations
- Create an empty stack
- Add a new item to the stack
- Remove from the stack the item that was added most recently
- Retrieve (but not remove) from the stack the item that was added most recently
- Determine whether a stack is empty

<table>
<thead>
<tr>
<th>Stack ADT operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>Adds an element to the top of the stack</td>
</tr>
<tr>
<td>pop</td>
<td>Removes an element from the top of the stack</td>
</tr>
<tr>
<td>peek</td>
<td>Examines the element at the top of the stack</td>
</tr>
<tr>
<td>isEmpty</td>
<td>Determines if the stack is empty</td>
</tr>
<tr>
<td>size</td>
<td>Determines the number of elements on the stack</td>
</tr>
</tbody>
</table>
public interface Stack<E> {

    /** Adds an item to the top of a stack. */
    public void push(E newItem);

    /** Removes the top of a stack. */
    public E pop();

    /** Retrieves the top of a stack. */
    public E peek();

    /** Determines whether stack is empty. */
    public boolean isEmpty();

    /** Determines whether stack is empty. */
    public int size();

}
Using a Stack – push and pop

```java
import java.util.*; // For Java's Stack class
public class StackTest {
    public static void main (String[] args) {
        Stack<String> stk = new Stack<String>();
        stk.push("one");
        stk.push("two");
        stk.pop();
        stk.push("three");
        System.out.println("Contents of Stack: " + stk);
    }
}

// What does stk contain now?
```

How can we print all the elements of a stack without destroying it?

---

Print a Stack without destroying it

```java
/**
 * @returns String representation of the contents of stk from top to bottom,
 * assuming that the E on the stack have their own toString() method
 */
public String toString (Stack<E> stk) {
    // Create a temporary stack to hold contents of stk
    Stack<E> tempStack = new Stack<E>();
    String s = "[
    while(!stk.isEmpty()) {
        E element = stk.pop();
        s = s + element.toString() + " ";
        tempStack.push(element);
    }
    s = s + "]");
    // restore contents of stk
    while(!tempStack.isEmpty())
        stk.push(tempStack.pop());
    return s;
}
```
Example: Checking for Balanced Braces

An example of balanced braces
a{b{c}{d}{e}}f|g

Examples of unbalanced braces
a{b}]: Too many closing braces
{c{d}e: Too few closing braces
{f(g|h] : Mismatching braces

Checking Balanced Braces: Helper Methods

/* returns true if c is an open bracket */
public boolean open_bracket (char c) {
    return (c == '(') || (c == '{') || (c == '[') || (c == '<');
}

/* returns true if c is a close bracket */
public boolean close_bracket (char c) {
    return (c == ')') || (c == '}') || (c == ']') || (c == '>');
}

/* returns the closing bracket matching the input open bracket */
public char matching_bracket (char c) {
    if (c == '(') return ')';
    else if (c == '{') return '}'
    else if (c == '[') return ']
    else return '>';
Checking Balanced Braces Pseudocode

while (still more chars to read) {
    get next char in the string
    if it is open_bracket
        then push it on top of the stack
    if it is a close_bracket
        pop char off stack
        check to see if it matches bracket
}

Start by declaring input string balanced
while (still more chars to read && string still balanced) {
    get next char in the string
    if it is open_bracket
        then push it on top of the stack
    if it is a close_bracket
        if stack empty => not balanced
        pop char off stack
        check to see if it matches bracket
        if not matched => not balanced
}
if stack not empty => not balanced
/** @returns true if string S has balanced open and closed brackets */
public boolean isBalanced (String s) {
    Stack<Character> stk = new Stack<Character>();
    int i = 0; char nextChar, top; boolean balanced = true;
    while (balanced && (i < s.length())) {
        nextChar = s.charAt(i); // get the next character in the string
        if (open_bracket(nextChar)) // push open brackets onto the stack
            stk.push(new Character(nextChar));
        else if (close_bracket(nextChar)) {
            // check whether the matching open bracket is on top of stack
            if (stk.isEmpty())
                balanced = false;
            else {
                top = stk.pop().charValue();
                if (nextChar != matching_bracket(top)) balanced = false;
            }
        }
        i++;
    }
    return (balanced && stk.empty());
}

Implementing a Stack

Note: Up to now we have been using a stack. Let's see how it works inside.
Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array
  - A reference-based list
  - The ADT LinkedList
  - The ADT Vector

- Stack Interface
  - Provides a common specification for the three implementations

- StackException
  - Used by Stack Interface
  - Extends java.lang.RuntimeException

```java
package javafoundations;

public interface Stack<E> {
    /** Adds the specified element to the top of the stack. */
    public void push(E newItem) {

    /** Removes and returns the top element from the stack. */
    public E pop();

    /** Returns a reference to top element of this stack without removing it. */
    public E peek();

    /** Returns true if the stack contains no elements and false otherwise. */
    public boolean isEmpty();

    /** Returns the number of elements in the stack. */
    public int size();

    /** Returns a string representation of the stack. */
    public String toString();
}
```
Array-based Implementation

- ArrayStack class
  - Implements Stack
  - Private data fields
    - An array of Objects called items
    - The index count
    - Top of stack is at count-1

![Figure 6.4](image)
An array-based implementation

```java
package javafoundations;
package javafoundations.exceptions.*;
public class ArrayStack<E> implements Stack<E> {
    private E[] stack;  // Assumes top of stack is at stack[count-1]
    private int count;
    private final int DEFAULT_CAPACITY = 10;  // Will expand as needed
    public ArrayStack() {
        count = 0;
        stack = (E[]) (new Object[DEFAULT_CAPACITY]);
    }
    public boolean isEmpty() {
    }
    public void push(E newItem) {
        if (count == stack.length) expandCapacity();
    }
    public E pop() throws EmptyCollectionException {
    }
    public E peek() throws EmptyCollectionException {
    }
}
```
**LinkedNode-based Implementation**

- **LinkedStack class**
  - Implements Stack
  - Private data fields
    - LinearNode (top) that holds the top element in the Stack
    - A count variable

Let’s work on this together in class!