CSE 12
The Stack Abstract Data Type

- The Stack ADT
- Stack attributes and operations
- Designing a test plan for Stack
- Implementing the Stack ADT
- The Adapter design pattern
Introduction

• **Stack**: a Last-In-First-Out (LIFO) data structure – the *last* item put into the structure will be the *first* item to be taken out of it

• Values: a sequence of data items of type T

• Operations:
  - push(T)
  - T pop()
  - T peek()
  - boolean isEmpty()
  - int size()

*these are the essential ones!*
Examples of Stack applications

• Browser “back” button: takes you to the last page visited before the current page

• Editor “undo” button: takes you to the last state of the document before the current state

• Operating system runtime stack: stores activation records in order to support method calls
  – The activation record on top of the stack contains information about the currently executing method, the activation record below it represents the method executing when the current method was called, and so on
Stack Properties and Attributes

Properties
• A Stack is a LIFO structure.
• Considered as a linear structure, any operation is done at only one end of the structure (referred to as the ‘top’ of the Stack).

Attributes

\textit{size} : The number of elements in the stack.
\hspace*{1cm} Invariant: \textit{size} \geq 0
\textit{top} : Indicates the end of the stack where operations are performed.
A picture of a stack

Top of the stack
Stack Operations

Stack()

pre-condition: none

responsibilities: constructor – initialize the stack attributes

post-condition: size is 0

top refers to null (a special value not part of the stack)

returns: nothing
Stack Operations

push( Type )

pre-condition: none

responsibilities: push element onto the top of the stack

post-condition: element is placed on top of the stack
all other elements of the stack remain in their previous positions
size is incremented by 1

returns: nothing
Stack Operations

pop()

pre-condition: isEmpty() is false

responsibilities: remove and return the element at top

post-condition: the top element is no longer in the stack
all other elements of the stack remain in their previous positions
size is decremented by 1

returns: the element removed

throws: empty stack exception if pre-condition is not met
Stack Operations

peek()

pre-condition: isEmpty() is false

responsibilities: return the element at top

post-condition: the stack is unchanged

returns: the element at top

throws: empty stack exception if pre-condition is not met
## Behavior of Stack’s operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
<th>Object State</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Stack&lt;String&gt; s = new LinkedStack&lt;String&gt;()</code></td>
<td>Create an empty stack s</td>
<td>size = 0 top = null</td>
<td>a LinkedStack object for Strings</td>
</tr>
<tr>
<td><code>s.push (&quot;A&quot;)</code></td>
<td>Add “A” to stack s</td>
<td>size = 1 “A” top</td>
<td></td>
</tr>
<tr>
<td><code>s.push (&quot;B&quot;)</code></td>
<td>Add “B” to stack s</td>
<td>size = 2 “A” “B” top</td>
<td></td>
</tr>
<tr>
<td><code>String str = s.peek()</code></td>
<td>Peek at the top element of stack s</td>
<td></td>
<td>“B”</td>
</tr>
<tr>
<td><code>str = s.pop()</code></td>
<td>Pop top element from stack s</td>
<td>size = 1 “A” top</td>
<td>“B”</td>
</tr>
<tr>
<td><code>s.isEmpty()</code></td>
<td>See if the stack s is empty</td>
<td></td>
<td>false</td>
</tr>
<tr>
<td><code>str = s.pop()</code></td>
<td>Pop top element from stack s</td>
<td>size = 0 top = null</td>
<td>“A”</td>
</tr>
<tr>
<td><code>s.isEmpty()</code></td>
<td>See if the stack s is empty</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td><code>str = s.peek()</code></td>
<td>Peek at the top element of stack s</td>
<td></td>
<td>exception</td>
</tr>
</tbody>
</table>
Designing a Test Plan

Ask questions!

• What are the “corner cases” for this collection type?
  – empty? full? not empty and not full? one element? one element away from being full?

• How does each operation change the state of the collection?
  – Should come from operations description and pre-conditions
    – the specification is important!

• What are the invalid conditions that must be checked?
  – Should come from operations pre-conditions and throws descriptions

Designing test cases helps clarify the specification. Don't start coding until you are clear about the behavior of the operations!
A Test Case: Pushing onto a Stack

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
<th>Object State</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack&lt;String&gt; s = new ListStack&lt;String&gt;()</td>
<td>Create an empty stack</td>
<td>size = 0</td>
<td>a ListStack for Strings</td>
</tr>
<tr>
<td></td>
<td></td>
<td>top = null position</td>
<td></td>
</tr>
<tr>
<td>s.push(&quot;www.nps.gov&quot;)</td>
<td>Push onto an empty stack</td>
<td>size = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>top = &quot;www.nps.gov&quot;</td>
<td></td>
</tr>
<tr>
<td>s.size()</td>
<td>Verify new stack state</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>s.isEmpty()</td>
<td>Verify new stack state</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>String str = s.peek()</td>
<td>Verify new stack state</td>
<td>&quot;www.nps.gov&quot;</td>
<td></td>
</tr>
<tr>
<td>s.push(&quot;www.nasa.gov&quot;)</td>
<td>Push onto a non-empty stack</td>
<td>size = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>top = &quot;www.nasa.gov&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://www.nps.gov">www.nps.gov</a></td>
<td></td>
</tr>
<tr>
<td>s.size()</td>
<td>Verify new stack state</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>str = s.peek()</td>
<td>Verify new stack state</td>
<td>&quot;www.nasa.gov&quot;</td>
<td></td>
</tr>
<tr>
<td>s.push(&quot;www.bls.gov&quot;)</td>
<td>Push onto a non-empty stack</td>
<td>size = 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>top = &quot;www.bls.gov&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://www.nasa.gov">www.nasa.gov</a></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://www.nps.gov">www.nps.gov</a></td>
<td></td>
</tr>
<tr>
<td>s.size()</td>
<td>Verify new stack state</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>str = s.peek()</td>
<td>Verify new stack state</td>
<td>&quot;www.bls.gov&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Software reuse and OO Design

• It’s a good idea to make use of existing software in new designs
  – If the existing software is of good quality (correct, efficient, and robust)
  – If you can use the existing software (check licensing issues)

• Two important OO software design patterns for software reuse are the Inheritance and the Adapter patterns
Inheritance Design Pattern

Problem:

• We are given an interface $I$, and we need to create a class $C$ that implements $I$
• We know of an existing class $B$ that offers some of the functionality required by $I$
• And this functionality in $B$ corresponds exactly to a subset of the API of $I$
• Solve this problem with class derivation and inheritance: define the new class $C$ to extend the existing class $B$

The Inheritance design pattern solves this problem by inheriting the API of the existing class, which exactly corresponds to a subset of the target interface.
Adapter Design Pattern

Problem:

• We are given an interface $I$, and we need to create a class $C$ that implements $I$

• We know of an existing class $A$ that offers some or all of the functionality required by $I$

• However, this functionality in $A$ is specified using a an API that does not correspond to a subset of $I$

• It would not be correct to define the new class $C$ to extend the existing class $A$… why?

• Instead, we use the Adapter pattern…

The Adapter design pattern solves this problem by *adapting* the API of the existing class to the target interface
Adapter Design Pattern

- Start with the requirements of a given target interface (named, say, \( I \))
- Identify an existing class (named, say, \( A \)) that has attributes and behavior similar to those required by \( I \)
- Define a class named, say, \( C \), declared to implement \( I \)
- Define \( C \) to include an instance variable of type \( A \)
- So each instance of \( C \) includes an instance of \( A \): every \( C \) has-a \( A \)
  - This is called containment or composition
- The methods in \( C \) are defined to call methods in \( A \) to do (some of) their work
  - This is called message forwarding or delegation
Adapter Design Pattern: UML Diagram

• This UML diagram shows class C adapting class A in order to implement interface I.
Adapter Design Pattern: Consequences

• The adapter pattern allows you to quickly implement a new class by reusing an existing class that provides at least something similar to the behavior needed.

• This is especially useful for rapid development of prototypes, when execution speed is not an issue.

• There may be a performance cost. The indirection introduced by the message forwarding from the \( C \) object to the \( A \) object requires additional time when an \( C \) method is called.

• [Keep in mind: the API of the adapted class \( A \) is visible only to the adapting class \( C \) and is completely hidden from the client]
ListStack: Applying the Adapter Pattern

Tasks to complete to implement a Stack by adapting an existing List class:

- Determine whether attributes or methods from List can take on the responsibility of attributes from Stack
- Determine whether methods from List can provide the behavior required by methods from Stack
- Implement the Stack methods using the information gathered in the first two tasks
Map Stack Attributes to List Attributes and/or Methods

• Remember: an attribute can either be stored (as a variable), or synthesized from other information available (by a method)

• Stack.size maps nicely to List.size()
  – cannot use attribute List.size directly; it is private!

• Stack.top – all stack operations occur at its top, which we can picture as one end of a linear structure.

• There are two ends to a list! Which end should correspond to Stack.top?
What attribute of List corresponds to Stack.top?

• Either the beginning or the end of a List could correspond to the top of a Stack. Let’s make the most efficient choice…

• What is the time cost of adding or removing an element at the head or at the tail of an N-element List…
  • If List is implemented using an array?
    Head: ________  Tail: __________
  • If List is implemented using a singly linked list?
    Head: ________  Tail: __________
  • If List is implemented using a doubly linked list with tail pointer?
    Head: ________  Tail: __________
Map Stack Attributes to ArrayList Attributes and/or Methods

Don’t underestimate the importance of doing this mapping first. Planning now saves time later.
Map Stack Methods to List Methods

- A consequence of that attribute mapping is that a push operation results in adding to the *tail* of the List

\[
\text{size()} - 1 \quad + \quad 1 \quad = \quad \text{size()}
\]

*tail* of list \quad next position beyond *tail* \quad location to “push” the new stack element

<table>
<thead>
<tr>
<th>Stack operation</th>
<th>List operation equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>push( element )</td>
<td>add( size(), element )</td>
</tr>
<tr>
<td>E pop()</td>
<td>E remove( size() - 1 )</td>
</tr>
<tr>
<td>E peek()</td>
<td>E get( size() - 1 )</td>
</tr>
<tr>
<td>int size()</td>
<td>int size()</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>boolean isEmpty()</td>
</tr>
</tbody>
</table>
What does the implementation look like?

```java
import java.util.ArrayList;
import java.util.List;
import java.util.EmptyStackException;

/**
 * An implementation of the Stack interface that adapts a java.util.List
 */
public class ListStack<E> implements Stack<E> {
    private java.util.List<E> stack;
    // the top element of stack is stored at position s.size() - 1 in the list.

    /**
     * Create an empty stack.
     */
    public ListStack() {
        stack = new ArrayList<E>();
    }
}
```

The contained instance of the adapted class
What does the implementation look like?

```java
/**
 * Determine if the stack is empty.
 * @return <tt>true</tt> if the stack is empty,
 * otherwise return <tt>false</tt>.
 */
public boolean isEmpty()
{
    return stack.isEmpty();
}

/**
 * Return the top element of the stack without removing it.
 * This operation does not modify the stack.
 * @return topmost element of the stack.
 * @throws EmptyStackException if the stack is empty.
 */
public E peek()
{
    if ( stack.isEmpty() )
        throw new EmptyStackException();
    return stack.get( stack.size() - 1 ) ;
}
```

message forwarding – let the List object do as much work as possible

Having done the attribute mapping, this is easy to figure out
What does the implementation look like?

```java
/**
 * Pop the top element from the stack and return it.
 * @return topmost element of the stack.
 * @throws EmptyStackException if the stack is empty.
 */
public E pop()
{
    if ( stack.isEmpty() )
        throw new EmptyStackException();
    return stack.remove( stack.size() - 1 );
}

/**
 * Push <tt>element</tt> on top of the stack.
 * @param element the element to be pushed on the stack.
 */
public void push( E element)
{
    stack.add( stack.size(), element );
}
```

Compare with `peek()`

List.size() - 1 is **top**, so “push” new element at List.size()
The Adapter design pattern is an example of software reuse: the adapted class has already been implemented and (hopefully!) tested and debugged.

As a result, the adapting class's implementation can often be very simple: many methods just consist of a call to an adapted class method.

In that case, it may almost be enough to just visually check that the adapting class's methods are correct, assuming the adapted class is correct.

Still, it is better if a complete test plan is developed and implemented, to ensure that the required interface is implemented correctly.
Another implementation of Stack

• Now let's consider another strategy for implementing the Stack<E> interface

• Here we will not use the Adapter pattern, adapting an existing class...
  – instead, we will implement the stack operations directly, without delegation

• And we will use a linked list structure, not an array, to hold the elements of the stack
A Linked Implementation of Stack

Tasks to complete to implement a Stack directly by using a linked list structure:

- Decide which linked structure representation to use: singly linked list or doubly linked list

- Map the Stack ADT attributes to a linked list implementation. In doing this we will identify the data fields (instance variables) our class will need

- Implement the Stack ADT operations as methods in our class
Singly or Doubly Linked List?

• How do we decide? Think:

• What are the essential characteristics of a Stack?
  – Additions and deletions happen at one end, the top

• So, don’t need the additional complexity of a doubly linked list; a singly linked list will do
Map Stack ADT attributes to a linked list implementation

- A linked list is a linear structure. Which end should correspond to the *top* of the Stack?
  - In a singly linked list, where is it easiest and most efficient to do inserts and deletes?
    ⇒ Select that end of the linked list to correspond to *top*

- Can use a dummy node at the list head, which may simplify inserts/deletes at the head

- In fact, can have *top* always point to the dummy node, even though it does not correspond to any element on the stack...
Map Stack ADT attributes to a linked list implementation

```java
public class LinkedStack<E> implements Stack<E> {
    private int size;
    private SLNode<E> top;
}
```

An empty `LinkedStack` using a dummy node for `top`
The push() operation

1. create a new SLNode  
2. Set the new node’s successor field to be the same as top’s next field  
3. Set top’s successor field to reference the new node.  
4. Increment size by 1  

1. public void push( E element ) { 
2.     SLNode newNode = 
3.         new SLNode<E>(element, 
4.                     top.getSuccessor());  
5.     top.setSuccessor( newNode );  
6.     size++;  
7. }

Steps (a) and (b)  
Step (c)  
Pushing an element onto an empty stack
The push() operation

1. create a new SLNode
(a)

2. Set the new node’s successor field to be the same as top’s next field
(b)

3. Set top’s successor field to reference the new node.  
(c)

4. Increment size by 1
(d)

1. public void push( E element ) {
2
3    SLNode newNode =
4        new SLNode<E>(element,
5        top.getSuccessor()); (a, b)
6
7    top.setSuccessor( newNode ); (c)
8
9    size++; (d)
10  }

Pushing an element onto a non-empty stack
## Stack Operation Costs

<table>
<thead>
<tr>
<th>Stack Operation</th>
<th>ArrayList Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>push( element )</td>
<td>add( size(), element )</td>
<td>Ω(1)</td>
</tr>
<tr>
<td>pop()</td>
<td>remove( size() – 1 )</td>
<td>Ω(1)</td>
</tr>
<tr>
<td>top()</td>
<td>get( size() – 1 )</td>
<td>Ω(1)</td>
</tr>
</tbody>
</table>

The cost of Stack operations for the Adapter implementation using an ArrayList.

What is the cost of Stack operations for the direct implementation using a **singly linked list**?
Next time

- The Queue ADT
- Designing a test plan for Queue
- The Adapter pattern again
- Alternative implementations of Queue
- Circular arrays
- Time costs of Queue implementations

Reading: Gray, Ch 7