

Data Structures

Stacks

CS284

Structure of this week's classes

Stacks

Applications

Implementing a Stack

Stack

- ▶ 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 one added to the stack most recently
- ▶ The stack's storage policy is Last-In, First-Out, or LIFO

Operations on Stacks

- ▶ Only the top element of a stack is visible; therefore the number of operations performed by a stack are few
- ▶ We need the ability to
 - ▶ test for an empty stack (`boolean empty()`)
 - ▶ inspect the top element (`E peek()`)
 - ▶ retrieve the top element (`E pop()`)
 - ▶ put a new element on the stack (`E push(E obj)`)

StackInt<E> interface

```
public interface StackInt<E> {  
  
    E push(E obj);  
  
    E peek();  
  
    E pop();  
  
    boolean empty();  
}
```

An example: a stack of strings `java.util.List` interface

Jonathan
Dustin
Robin
Debbie
Rich

(a)

Dustin
Robin
Debbie
Rich

(b)

Philip
Dustin
Robin
Debbie
Rich

(c)

- ▶ “Rich” is oldest element on stack; “Jonathan” is youngest (Figure a)
- ▶ `String last = names.peek();`
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)

Stacks

Applications

Implementing a Stack

Finding Palindromes

- ▶ Palindrome: a string that reads identically in either direction, letter by letter
 - ▶ “kayak”
 - ▶ “I saw I was I”
 - ▶ “Able was I ere I saw Elba”
 - ▶ “Level, madam, level”
- ▶ We ignore case and whitespace
- ▶ Problem: Write a program that reads a string and determines whether it is a palindrome

PalindromeChecker class

PalindromeChecker

```
private String inputString  
private Stack<Character> charStack
```

```
public PalindromeChecker(String str) // Stores str and pushes onto stack  
private void fillStack()  
private String buildReverse()  
public boolean isPalindrome()
```

PalindromeChecker class

```
import java.util.*;

public class PalindromeChecker {
    private String inputString;
    private Stack<Character> charStack = new Stack<Character>();

    public PalindromeChecker(String str) {
        inputString = str;
        fillStack();
    }
    ...
}
```

String to Stack

```
private void fillStack() {  
    for(int i = 0; i < inputString.length(); i++) {  
        if (inputString.charAt(i) != ' '){  
            charStack.push(inputString.charAt(i));  
        }  
    }  
}
```

Stack to String

```
private String buildReverse(){
    StringBuilder result = new StringBuilder();
    while(!charStack.empty()) {
        result.append(charStack.pop());
    }
    return result.toString();
}
```

Compare original string with reversed one

```
public boolean isPalindrome() {  
    return inputString.equalsIgnoreCase(buildReverse());  
}
```

Testing

We can test this class using the following inputs:

- ▶ a single character (always a palindrome)
- ▶ multiple characters in a word
- ▶ multiple words
- ▶ different cases
- ▶ even-length strings
- ▶ odd-length strings
- ▶ the empty string (considered a palindrome)

Balanced Parenthesis

- ▶ When analyzing arithmetic expressions, it is important to determine whether an expression is balanced with respect to parentheses

$$(a + b * (c / (d - e))) + (d / e)$$

- ▶ The problem is further complicated if braces or brackets are used in conjunction with parentheses
- ▶ The solution is to use stacks!

Principal operations we shall implement

```
public static boolean isBalanced(String expression)
```

- ▶ Returns true if the expression is balanced w.r.t. parenthesis and false otherwise

```
private static boolean isOpen(char ch)
```

- ▶ Returns true if ch is an opening parenthesis

```
private static boolean isClose(char ch)
```

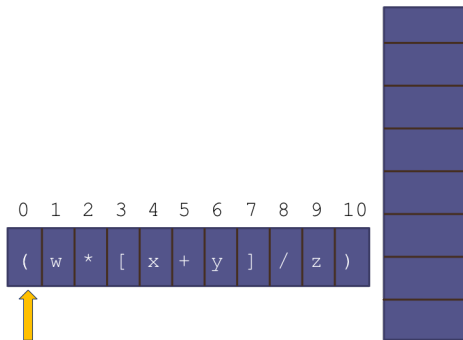
- ▶ Returns true if ch is a closing parenthesis

The algorithm for `isBalanced`

```
Create an empty stack of characters
balanced = true
Set index to 0
while (balanced and index < expression's length) {
  Get the next character in the data string
  if (the next character is an opening parenthesis)
    { Push it onto the stack }
  else if (the next character is a closing parenthesis)
    { Pop the top of the stack
      If (the stack was empty or its top does not match the c
        { balanced = false }
    }
  Increment index
}
Return true if balanced is true and the stack is empty
```

A Sample Execution

$$(w * [x + y]) / z$$



ParenChecker Class

```
public class ParenChecker {  
  
    private static final String OPEN = "[{";  
    private static final String CLOSE = ")]";  
  
    public static boolean isBalanced(String expression) {  
        ...  
    }  
  
    private static boolean isOpen(char ch) {  
        return OPEN.indexOf(ch) > -1;  
    }  
  
    private static boolean isClose(char ch) {  
        return CLOSE.indexOf(ch) > -1;  
    }  
}
```

ParenChecker Class (cont.)

```
public static boolean isBalanced(String expression) {
    Stack<Character> s = new Stack<Character>();
    boolean balanced = true;
    try {
        int index = 0;
        while (balanced && index < expression.length()) {
            char nextCh = expression.charAt(index);
            if (isOpen(nextCh)) {
                s.push(nextCh);
            } else if (isClose(nextCh)) {
                char topCh = s.pop();
                balanced = OPEN.indexOf(topCh) == CLOSE.indexOf(nextCh);
            }
            index++;
        }
    } catch (EmptyStackException ex) {
        balanced = false;
    }
    return (balanced && s.empty());
}
```

Stacks

Applications

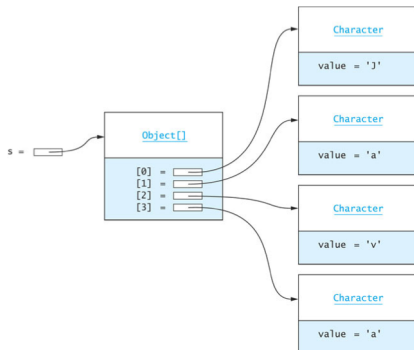
Implementing a Stack

A Stack as an Extension of Vector

- ▶ Java API includes Stack class as part of package `java.util`:

```
public class Stack<E> extends Vector<E>
```

- ▶ Elements of a Vector can be accessed using an integer index and the size can grow or shrink as needed to accommodate the insertion and removal of elements



Implementing a Stack as an Extension of Vector (cont.)

We can use Vector's add method to implement push:

```
public E push(obj E) {  
    add(obj);  
    return obj;  
}
```

pop can be coded as

```
public E pop throws EmptyStackException {  
    try {  
        return remove(size() - 1);  
    } catch (ArrayIndexOutOfBoundsException ex) {  
        throw new EmptyStackException();  
    }  
}
```

Disadvantage of Stack as Subclass of Vector

- ▶ Because a Stack is a Vector, all of Vector operations can be applied to a Stack (such as searches and access by index)
- ▶ But, since only the top element of a stack should be accessible, this violates the principle of **information hiding**

```
Stack<Integer> s = new Stack<Integer>();  
s.push(3);  
s.push(4);  
System.out.println(s.elementAt(0)); // prints 3
```


Implementing a Stack with a List Component

- ▶ Alternative: write a class, `ListStack`, that has a `List` component (in the example below, `theData`)

```
public class ListStack<E> implements StackInt<E>{
    private List<E> theData;
    public ListStack() {
        theData = new ArrayList<E>();
    }
    ...
}
```

Implementing a Stack with a List Component

- ▶ We can use `ArrayList`, `Vector`, or the `LinkedList` classes, as all implement the `List` interface.
- ▶ The `push` method, for example, can be coded as

```
public E push(E obj) {  
    theData.add(obj);  
    return obj;  
}
```

- ▶ A class which adapts methods of another class by giving different names to essentially the same methods (`push` instead of `add`) is called an **adapter class**
- ▶ Writing methods in this way is called **method delegation**

Implementing a Stack Using an Array

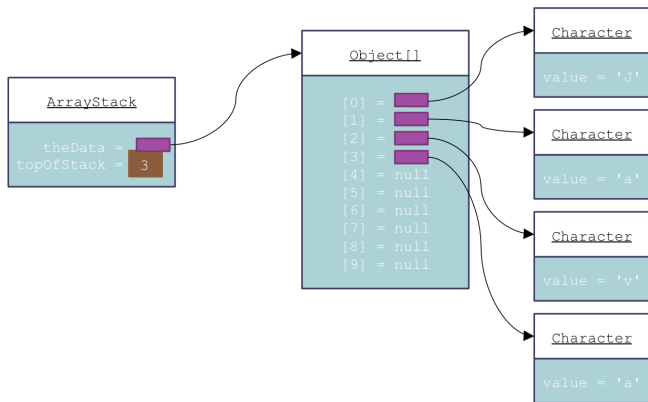
If we implement a stack as an array, we would need...

```
public class ArrayStack<E> implements StackInt<E> {
    private E[] theData;
    int topOfStack = -1;
    private static final int INITIAL_CAPACITY = 10;

    @SuppressWarnings("unchecked")
    public ArrayStack() {
        theData = (E[])new Object[INITIAL_CAPACITY];
    }
}
```

- ▶ **Note:** The `SuppressWarnings` annotation is just to avoid the compiler from warning: "Type safety: Unchecked cast from `Object[]` to `E[]`"

Implementing a Stack Using an Array

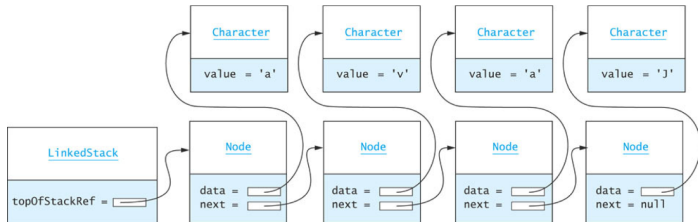


```
public E push(E obj) {  
    if (topOfStack == theData.length-1) { reallocate(); }  
    topOfStack++;  
    theData[topOfStack] = obj;  
    return obj;  
}
```

Implementing a Stack Using an Array

```
public E pop() {  
    if (empty()) {  
        throw new EmptyStackException();  
    }  
    return theData[topOfStack--];  
}
```

Implementing a Stack as a Linked List



when the list is empty, pop returns null

Implementing a Stack as a Linked List

```
import java.util.EmptyStackException;

public class LinkedStack<E> implements StackInt<E> {

    private static class Node<E> {
        private E data;
        private Node next;
    }
}
```

Implementing a Stack as a Linked List

```
// Constructors
private Node(E dataItem) {
    data = dataItem;
    next = null;
}

private Node(E dataItem, Node<E> nodeRef) {
    data = dataItem;
    next = nodeRef;
}
} //end class Node
```


Implementing a Stack as a Linked List

```
// Data Fields
/** The reference to the first stack node. */
private Node<E> topOfStackRef = null;

/** Insert a new item on top of the stack.
    post: The new item is the top item on the stack.
         All other items are one position lower.
    @param obj The item to be inserted
    @return The item that was inserted
 */
public E push(E obj) {
    topOfStackRef = new Node<E>(obj, topOfStackRef);
    return obj;
}
```

Implementing a Stack as a Linked List

```
/** Remove and return the top item on the stack.  
    pre: The stack is not empty.  
    post: The top item on the stack has been  
         removed and the stack is one item smaller.  
    @return The top item on the stack  
    @throws EmptyStackException if stack is empty  
 */  
public E pop() {  
    if (empty()) {  
        throw new EmptyStackException();  
    }  
    else {  
        E result = topOfStackRef.data;  
        topOfStackRef = topOfStackRef.next;  
        return result;  
    }  
}
```

Implementing a Stack as a Linked List

```
/** Return the top item on the stack.  
    pre: The stack is not empty.  
    post: The stack remains unchanged.  
    @return The top item on the stack  
    @throws EmptyStackException if stack is empty  
 */  
public E peek() {  
    if (empty()) {  
        throw new EmptyStackException();  
    }  
    else {  
        return topOfStackRef.data;  
    }  
}
```

Implementing a Stack as a Linked List

```
/** See whether the stack is empty.  
    @return true if the stack is empty  
    */  
public boolean empty() {  
    return topOfStackRef == null;  
}  
}
```

Comparison of Implementations

- ▶ Extending a `Vector` (as is done by Java) is a poor choice for stack implementation, since all `Vector` methods are accessible
- ▶ The easiest implementation uses a `List` component (`ArrayList` is the simplest) for storing data
 - ▶ An underlying array requires reallocation of space when the array becomes full, and
 - ▶ An underlying linked data structure requires allocating storage for links
 - ▶ As all insertions and deletions occur at one end, they are constant time, $\mathcal{O}(1)$, regardless of the type of implementation used

More Applications

- ▶ Evaluating postfix expressions
- ▶ Converting postfix to infix expressions