COSC 2006: Data Structures I

The stack ADT
Stack interface
Fixed array implementation
Dynamic array implementation
Linked implementation
Applications of stacks
Stack data type (1)

- Think of a stack of plates.
- Unlike an array, there is no random access to the individual plates in the stack.
- To access the 4th plate it is necessary to take 3 plates off the top of the stack.
- The basic operations take place only at the top:
  - pop: remove the plate at the top of the stack
  - push: put a new plate on top of the stack.
Stacks are used to solve many problems in computer science.

One important application is to keep track of return addresses as functions (methods) are called.

For example in a Java program if a method calls another method how does the system know how to return to the point just after the first method when the called method completes execution?
Stacking Return Addresses (1)

Here sA is the statement after the call to methodA and sB is the statement after the call to methodB
Stacking Return Addresses (2)

1. When methodA is called the return address of sA is pushed onto the stack.

2. When methodB is called the return address of sB is pushed onto the stack.

When methodB returns the address on top of stack is popped to find out where to return to in methodA.

When methodA returns the address on top of stack is popped to find out where to return to in the first method.
Push and pop operations

push(x)      push x onto stack (x is now at top of stack)
x = pop()   pop top of stack and store it in variable x

push(a)
push(b)
x = pop()
push(c)
y = pop()
z = pop()

Start with empty stack

11/9/2006
The basic operations are push and pop but we also need a way to check if the stack is empty or not. This can be done with a function called isEmpty() which returns true if the stack is empty and false otherwise.

It is an error to try to pop something from an empty stack.

It is also convenient to have an operation that returns the value at the top of the stack without removing it.

\[ x = \text{peek()} \]

return the top element without popping.

Without this operation you would have to do a pop and a push which is not convenient and would be less efficient.
Other operations (2)

Sometimes pop is defined so that it removes the top element but does not return it.

In this case it is necessary to do the two statements

\[
\begin{align*}
x &= \text{peek}() \\
\text{pop}() & \quad \text{pop}()
\end{align*}
\]

in order to return the top element and remove it.

Sometimes the peek operation is called top.
Example: matching parentheses

A simple example is to use a stack to determine whether an expression has matched parentheses.

Example: $a(bc(dfw)vsd)$ has matched parentheses

Example: $a(bc)dfw)vsd)$ has unmatched parentheses

The basic algorithm is to read the expression. Each time a left parenthesis is encountered push it onto the stack. Each time a right parentheses is encountered pop a left parenthesis from the stack. The parentheses are balanced only if we never try to pop from an empty stack and at end the stack should be empty.
Different kinds of parentheses

A stack can also be used to evaluate expressions involving several kinds of parentheses such as (), {}, [], <>.

The same basic algorithm can be used: we push the different kinds of left parentheses when encountered. When we encounter a right parenthesis we pop a left parenthesis and check if it's the right kind.
Normal algebra uses infix (operator between operands)
Postfix algebra uses postfix (operator after operands)

Infix notation

\[ a + b \]
\[ (a + b) \times c \]
\[ a + b \times c \]
\[ (a + b) \times (c / d) \]

Postfix notation

\[ a \ b \ + \]
\[ a \ b \ + \ c \ * \]
\[ b \ c \ * \ a \ + \]
\[ a \ b \ c \ * \ + \]
\[ a \ b \ + \ c \ d \ / \ * \]
Advantage of postfix

No parentheses are necessary

No operator precedence is needed

A stack can be used to efficiently evaluate postfix expressions

Some machine architectures are stack based

The Java virtual machine is a stack based architecture
Evaluating postfix expressions

Evaluate \((3 + 4) \times (6 - 2)\) using a stack and the postfix expression \(3 \ 4 \ + \ 6 \ 2 \ - \ *\).

Draw stack sideways with top element at the right end

| push(3) | 3 |
| push(4) | 3  4 |
| y = pop() | 3 |
| x = pop() | empty |
| push(x + y) | 7 |
| push(6) | 7  6 |
| push(2) | 7  6  2 |
| y = pop() | 7  6 |
| x = pop() | 7 |
| push(x - y) | 7  4 |

| y = pop() | 7 |
| x = pop() | empty |
| push(x * y) | 28 |
| result = pop() | empty |

The result is 28
Stack specification (ADT)

A Stack contains elements of a specific type and has the following operations. We include a name for the stack so that we can have several stacks.

- `s.push(x)` push x onto the top of stack s
- `x = s.pop()` pop the top element from stack s
- `x = s.peek()` store top of stack s in x without popping
- `s.isEmpty()` return true if stack s is empty else return false
In Java Stacks are objects and we can use a Stack interface as the specification of the stack operations. Any class that implements the interface "is a" Stack.

We can use the generic Object type for elements stored in stack.

```java
public interface Stack {
    public void push(Object e);
    public Object pop() throws EmptyStackException;
    public Object peek() throws EmptyStackException;
    public void clear(); // gives an empty stack
    public boolean isEmpty();
    public int size(); // number of elements on stack
    public String toString();
}
```
In Java 1.5 (Java 5) we can use a generic type \( E \) instead of the Object type to define a generic stack. Then type casting is not necessary and auto boxing/unboxing can be used to easily work with stacks of primitive types.

```java
class Stack<E>
{
    public void push(E e);
    public E pop() throws EmptyStackException;
    public E peek() throws EmptyStackException;
    public void clear(); // gives an empty stack
    public boolean isEmpty();
    public int size(); // number of elements on stack
    public String toString();
}
```
We need an EmptyStackException exception class. Such classes are easily written by extending RuntimeException.

```java
package stacksAndQueues;
public class EmptyStackException extends RuntimeException {
    public EmptyStackException(String errorMessage) {
        super(errorMessage);
    }
}
```

Actually this class exists in java.util but we will use our own.
Here we use an array to hold the elements on the stack. An array index is used to mark the top of the stack. The simplest implementation is to use a fixed size array. In this case we can have a full stack so we will need the following exception class to use if push would overflow the array.

```java
package stacksAndQueues;
public class FullStackException extends RuntimeException {
    public FullStackException(String errorMessage) {
        super(errorMessage);
    }
}
```
The array holds references to the objects in the stack. Index top has the value -1 for an empty stack. Otherwise it locates the top element on stack.

- Array (empty stack)
  - top = -1

- Array with 3 objects in stack
  - top = 2
  - push(a), push(b), push(c)
The class is called FixedArrayStack<

package stacksAndQueues;
public class FixedArrayStack<E>
    implements Stack<E>
{
    private E[] data; // array of refs
    private int top; // index of top of stack

    // continued next slide
The constructor checks for a valid maxCapacity and creates an array of type Object that is typecast to an array of type E. Note: It is not possible to directly create an array of type E. This is the only time that a typecast is needed so in Java 5 you will get a warning which you can safely ignore.

```java
public FixedArrayStack(int maxCapacity) {
    if (maxCapacity < 1)
        throw new IllegalArgumentException(
            "maxCapacity less than 1");

    data = (E[]) new Object[maxCapacity];
    top = -1; // empty stack
}
```
Fixed array implementation (4b)

Can also have a default constructor for an array of size 10

```java
public FixedArrayStack()
{
    this(10);
}
```
In this fixed array implementation push needs to check if the stack is full. If not the top index can be incremented and the new data element can be stored in location indicated by top. Note that isFull is not part of the Stack interface. It is only needed in this implementation.

```java
public void push(E e) throws FullStackException {
    if (isFull())
        throw new FullStackException("push: stack overflow");
    top++;
    data[top] = e;
}
```
pop needs to check for an empty stack. If stack is not empty then save the data element before decrementing the top index. The data element can then be returned.

```java
public E pop() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException("pop: stack is empty");
    E topElement = data[top];
data[top] = null; // help garbage collector
top--; return topElement;
}
```
peek also needs to check for an empty stack. If stack is not empty then the data element at top of stack can be returned.

```java
public E peek() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException("peek: stack is empty");

    return data[top];
}
```
To clear the stack so that it is empty it is only really necessary to set the top index to -1 but since the array is not being deleted we can help the garbage collector by removing references to objects by recreating the array.

```java
public void clear()
{
    data = (E[]) new Object[data.length];
    top = -1;
}
```
**isEmpty, size and isFull are easy**

```java
public boolean isEmpty()
{
    return top == -1;
}

public int size()
{
    return top + 1;
}

public boolean isFull()
{
    return size() == data.length;
}
```
Fixed array implementation (10)

The contract for toString in the interface says that toString should use the format [a,b,c,...] where a is top of stack. Therefore we need to traverse the array backwards.

```java
public String toString()
{
    StringBuilder s = new StringBuilder();
    s.append("[");
    for (int k = size() - 1; k >= 0; k--)
    {
        if (k > 0)
            s.append(data[k] + ",");
        else
            s.append(data[k]);
    }
    s.append("]");
    return s.toString();
}
```
Dynamic array

Recall that a dynamic array is an array that can be expanded in size when it becomes full:

- (1) make a new array twice the size of original array
- (2) copy the original array data to the new array
- (3) make a reference to the new array so that the original one can be garbage collected.

In a similar way we could also provide a trim operation which decreases the size to just that of the elements being used.
The class is called DynamicArrayStack<E> and it implements the interface Stack<E> using an array of references of type E and an integer top for the top index.

```java
package stacksAndQueues;
public class DynamicArrayStack<E> implements Stack<E> {
    private E[] data; // array of refs
    private int top;   // index of top of stack

    // continued next slide
```
The constructor checks for a valid initialCapacity and creates an array of type Object that is typecast to an array of type E. Note: It is not possible to directly create an array of type E. This is the only time that a typecast is needed so in Java 5 you will get a warning which you can safely ignore.

```java
public DynamicArrayStack(int initialCapacity) {
    if (initialCapacity < 1)
        throw new IllegalArgumentException("initialCapacity less than 1");

    data = (E[]) new Object[initialCapacity];
    top = -1; // empty stack
}
```
Dynamic array impl (2b)

Can also have a default constructor for an initial array of size 10

```java
public DynamicArrayStack()
{
    this(10);
}
```
In this dynamic array implementation push does not need to check if the stack is full. Instead the reallocate method is called to expand the array. The top index can be incremented and the new data element can be stored in location indicated by top. The isFull method is not needed in the dynamic implementation.

```java
public void push(E e)
{
    if (size() == data.length)
        reallocate(); // double array size

    top++;
    data[top] = e;
}
```
The pop method is identical in both the fixed and dynamic implementations.

```java
public E pop() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException(
            "pop: stack is empty");
    E topElement = data[top];
    data[top] = null; // help garbage collector
    top--;
    return topElement;
}
```
The peek method is identical in both the fixed and dynamic implementations.

```java
public E peek() throws EmptyStackException
{
    if (isEmpty())
        throw new EmptyStackException("peek: stack is empty");

    return data[top];
}
```
The clear method is identical in both the fixed and dynamic implementations.

```java
public void clear()
{
    data = (E[]) new Object[data.length];
    top = -1;
}
```
isEmpty and size are the same in both the fixed and dynamic implementations. In the dynamic implementation an isFull method is not needed.

```java
public boolean isEmpty()
{
    return top == -1;
}

public int size()
{
    return top + 1;
}
```
The toString method is the same in both fixed and dynamic implementations.

```java
public String toString()
{
    StringBuilder s = new StringBuilder();
    s.append('[');
    for (int k = size() - 1; k >= 0; k--)
    {
        if (k > 0)
        {
            s.append(data[k] + ',');
        } else
        {
            s.append(data[k]);
        }
    }
    s.append(']');
    return s.toString();
}
```
Dynamic array impl (9)

Reallocate performs 3 steps
(1) Make a new array twice as big as the current one
(2) Copy current array data to the new array
(3) Make new array the current one (garbage collect old one)

System.arraycopy is a more efficient version of the for loop
for (int k = 0; k < data.length; k++)
    newData[k] = data[k];

public void reallocate()
{
    int newCapacity = 2 * data.length;
    E[] newData = (E[]) new Object[newCapacity];
    System.arraycopy(data, 0, newData, 0, data.length);
    data = newData;
}
Constructing stacks

We now have two different implementations of the stack interface so we can construct a stack using either of them as follows:

**Fixed stack for as many as 100 characters.**

```java
Stack<Character> stack =
    new FixedArrayStack<Character>(100);
```

**Dynamic stack initially for 100 characters but expandable when this limit is reached.**

```java
Stack<Character> stack =
    new DynamicArrayStack<Character>(100);
```
package stacksAndQueues;
public class StackTester
{
    public void doTest()
    {
        Stack<Double> s = new DynamicArrayStack<Double>(5);
        s.push(1.0);
        s.push(2.0);
        s.push(3.0);
        System.out.println(s);
        s.pop(); s.pop();
        System.out.println(s);
    }
    public static void main(String[] args)
    {
        new StackTester().doTest();
    }
}
Linked stack (1)

- We can use a linked structure (list) to implement a stack called a LinkedStack.
- The head of the list is the top of the stack.
- Pushing an element means adding a new node at the head of the list.
- Popping an object means deleting the node at the head of the list.
- This implementation is dynamic.
Linked representation of the stack \([e_1, e_2, e_3, e_4]\) with \(e_1\) the top element. Here \(e_1, e_2, e_3, e_4\) are references to the data objects shown in the picture.
Linked stack (3)

Pushing an element e onto the stack:
(1) make a new node
(2) link it to e1
(3) link top to the new node

New list is
[ e, e1, e2, e3, e4 ]

Step (1)
Step (2)
Step (3)

top = new Node<E>(e, top);
size++;
**Linked stack (4)**

Pop an element from the stack:
*Just link top to the second node. The first node becomes an orphan*

New list is

\[
[ e2, e3, e4 ]
\]

```
E element = top.data;
top = top.next;
size--; return element;
```
The class is called LinkedStack<E> and it implements the interface Stack<E> using a linked structure whose head node is the top of the stack. The Node<E> class is an inner class.

```java
package stacksAndQueues;
public class LinkedStack<E> implements Stack<E> {
    private Node<E> top; // Node<E> is inner class
    private int size; // number of elements in stack

    // continued next slide
```
**Linked implementation (2)**

*Constructor just creates an empty list*

```java
public LinkedStack()
{
    top = null;
    size = 0;
}
```
push method adds a new node to the head of the list. If top is null this correctly makes a one-element list.

```java
public void push(E e) {
    top = new Node<E>(e, top);
    size++;
}
```
pop method checks for an empty stack.
If not empty the top reference is linked to the second node
If there is no second node (one-element list) then top correctly becomes null

```java
public E pop() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException(
            "pop: stack is empty");
    E element = top.data;
    top = top.next;
    size--;
    return element;
}
```
peek method checks for an empty stack. If not empty the data in the top node is returned.

```java
public E peek() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException("peek: stack is empty");
    return top.data;
}
```
Linked implementation (6)

clear, isEmpty, size are easy.

```java
public void clear()
{
    top = null;
    size = 0;
}

public boolean isEmpty()
{
    return size == 0;
}

public int size()
{
    return size;
}
```
toString method traverses the list

```java
public String toString()
{
    StringBuilder s = new StringBuilder();
    s.append('[');
    Node<E> cursor = top;
    while (cursor != null)
    {
        s.append(cursor.data.toString());
        if (cursor.next != null) // not last
        {
            s.append(',');
        }
        cursor = cursor.next;
    }
    s.append(']');
    return s.toString();
}
```
Linked implementation (8)

the inner node class

```java
private class Node<T>
{
    private T data;
    private Node<T> next;

    public Node()
    {
        this(null, null);
    }

    public Node(T data, Node<T> next)
    {
        this.data = data;
        this.next = next;
    }
}
```
Stack iterator (1)

- We can only access the elements of a stack at the top position.
- However, we may want to extract the data and process it in some manner without modifying the stack.
- The `toString` method is an example that displays the elements in a specific format.
- Producing this display in another format is not possible using the current interface.
Stack iterator (2)

- That's not quite true since we can pop all elements into another stack and process them as we pop them.
- Then we can pop them off the other stack and push them back on the original stack to recover the original stack.
- This would be quite inefficient.
- What we really need is an external immutable iterator to traverse the stack from top to bottom.
Stack Iterator example (1)

- make a new package
  - stacksAndQueues.iteratorExample
- Copy Stack.java to it (stack interface)
- Copy LinkedStack.java to it
- Copy EmptyStackException.java to it
- Remove the internal Node class from the LinkedStack class so that it is a separate class in Node.java
Stack Iterator example (2)

Now to get LinkedStack to work properly it is necessary to use the get methods instead of accessing the data fields directly since Node is no longer an inner class.

For example in the pop method it is now necessary to do the conversion.

```
E element = top.data;
top = top.next;
```

```
E element = top.getData();
top = top.getNext();
```

Similarly toString will need to be modified.
Now modify the Stack<E> interface so that it extends the Iterable<E> interface

```java
import java.util.Iterator;
public interface Stack<E> extends Iterable<E>
{
    // original methods go here and add the following
    // method

    /**
     * The iterator traverses the stack from top to
     * bottom and does not implement the remove method.
     */
    public Iterator<E> iterator();
}
```
The `Iterable<E>` interface in `java.lang` and the `Iterator<E>` interface in `java.util` are

```java
public interface Iterable<E>
{
    public Iterator<E> iterator();
}
```

```java
import java.util.*;
public interface Iterator<E>
{
    public boolean hasNext();
    public E next();
    public E remove(); // we won't support this
}
```
Since our `LinkedStack<E>` class must implement the `Iterable<E>` interface we need to add to it the following method

```java
public class LinkedStack<E> implements Stack<E>
{
    // original methods go here

    public Iterator<E> iterator()
    {
        return new LinkedIterator<E>(top);
    }
}
```
Write the `LinkedIterator<E>` class as follows.

```java
public class LinkedIterator<E> implements Iterator<E> {
    private Node<E> current;
    public LinkedIterator(Node<E> start) {
        current = start;
    }

    public boolean hasNext() {
        current != null;
    }

    // continued next slide
```
Stack Iterator example (6b)

Write the LinkedIterator\(<E>\) class next and remove methods:

```java
public E next()
{
    if (! hasNext())
        throw new NoSuchElementException("no element");
    E data = current.getData();
    current = current.getNext();
}

public void remove()
{
    throw new UnsupportedOperationException("remove is not supported");
}
```
Stack Iterator example (7)

Following statements show how to use the for each statement

```java
Stack<Integer> stack = new LinkedStack<Integer>();
stack.push(1);
stack.push(2);
stack.push(3);

for (Integer i : stack)
{
    System.out.println(i);
}
```
Stack Iterator example (8)

Following statements show how to use the iterator

```java
Stack<Integer> stack = new LinkedStack<Integer>();
stack.push(1);
stack.push(2);
stack.push(3);

Iterator<Integer> iter = stack.iterator();
while (iter.hasNext())
{
    System.out.println(iter.next());
}
```
Stack Iterator example (9)

A method that provides a string representation, one element per line instead of in the format [a,b,c,...] provided by toString

```java
public static <E> String stackToString(Stack<E> stack) {
    StringBuilder sb = new StringBuilder();
    for (E data : stack) {
        sb.append(data + "\n");
    }
    return sb.toString();
}
```
This class shows how to use a stack to match just one kind of parenthesis: ()..

```java
package stacksAndQueues.parenthesis;
import stacksAndQueues.EmptyStackException;
import stacksAndQueues.LinkedStack;
import stacksAndQueues.Stack;

public class ParenthesisMatcher
{
    private static final char LEFT = '(';
    private static final char RIGHT = ')';
}
Algorithm: push left parentheses on stack as they are encountered in the string. When a right parenthesis is encountered pop one of the left parentheses. Parentheses are not matched if stack is empty when a right parenthesis is encountered or if the stack is not empty when the string has been processed.

```java
public static int match(String str) {
    Stack<Character> s = new LinkedStack<Character>();
    // continued next slide
```
```java
for (int k = 0; k < str.length(); k++)
    {  char ch = str.charAt(k);
        if (ch == LEFT) 
            s.push(LEFT);
        else if (ch == RIGHT)
        {
            if (s.isEmpty()); // too many LEFT
                return 1;
            else
                s.pop();
        }
    }
    // -1 means too many RIGHT
    return s.isEmpty() ? 0 : -1;
} // end match
```
Test method. (Don't really need this method but its useful if you are using BlueJ or an Eclipse scrapbook page).

```java
public void doTest(String str) {
    int result = match(str);
    if (result == 0)
        System.out.println(str + " is matched");
    else
        System.out.println(str + " is not matched");
        if (result == -1)
            System.out.println("Too many left paren");
        else if (result == 1)
            System.out.println("Too many right paren");
}
```
GuiMatcher (1)

A nice way to test classes is to write a simple GUI. Here is one for testing ParenthesisMatcher that can easily be adapted to test other classes.
GuiMatcher (2)

Some typical output

```
(a)(b) (matched)
(a)(b)(c) (matched)
(a)(c)(d) (too many left parentheses)
(a)(c)(d) (too many right parentheses)
```

Enter expression then press enter or click execute

```
(a)(b)(c)(d)
```
A nice way to test classes is to write a simple GUI. Here is one for testing ParenthesisMatcher that can easily be adapted to test other classes.

```java
package stacksAndQueues.apps;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
public class GuiMatcher extends JFrame {
    private static final long serialVersionUID = 1L;
    private JTextField input;
    private JTextArea output;
    private JButton execute;
    ...
}
```
public GuiMatcher()
{
    setTitle("Balancing parentheses");
    Font courier = new Font("Courier New",
        Font.BOLD, 16);

    input = new JTextField(40);
    input.setEditable(true);
    input.setFont(courier);

    execute = new JButton("Execute");
    output = new JTextArea(10,50);
    output.setEditable(false);
    output.setFont(courier);
```java
JPanel p1 = new JPanel();
p1.setLayout(new FlowLayout());
p1.add(input);
p1.add(execute);

JLabel text =
    new JLabel("Enter expression" + " then press enter or click execute",
               JLabel.CENTER);

JPanel p2 = new JPanel();
p2.setLayout(new GridLayout(2,1));
p2.add(text);
p2.add(p1);
```
Container cp = getContentPane();
cp.setLayout(new BorderLayout());
cp.add(p2, BorderLayout.SOUTH);
cp.add(new JScrollPane(output), BorderLayout.CENTER);
pack();

execute.addActionListener(new ExecuteButtonHandler());

input.addActionListener(new EnterKeyHandler());
} // end of constructor
public class ExecuteButtonHandler
  implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      doMatch();
    }
  }

public class EnterKeyHandler
  implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      doMatch();
    }
  }
private void doMatch()
{
    String s = input.getText().trim();
    output.append(s);
    int result = ParenthesisMatcher.match(s);
    if (result == 0)
        output.append(" (matched) ");
    else if (result == -1)
        output.append(" (too many left paren) ");
    else
        output.append(" (too many right paren) ");
    output.append("\n");
}
public static void main(String[] args)
{
    JFrame f = new GuiMatcher();
    f.setVisible(true);
    f.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
}
}
Postfix calculator (1)

We need to tokenize the input stream so that the individual tokens (strings) such as numbers and operators (+, -, ...) are delivered to the program one at a time.

Example: 3.14  4  +  2  / has 5 tokens separated by whitespace (one or more spaces, newlines)

The input stream can be tokenized using the Scanner class and its hasNext() and next() methods.
Postfix calculator (2)

Algorithm for evaluating a postfix expression

- If a number is encountered push it onto the stack
- If a binary operator such as (+, -, *, /) is encountered pop two numbers from the stack perform the operation and push the result onto the stack
- If a unary operator such as (neg or sin) is encountered pop one number, do calculation and push the result.
- We can also incorporate variables
This class has a `processToken` method that takes a token and performs the associated operation using the stack.

```java
package stacksAndQueues.calculator;
import stacksAndQueues.Stack;
public class PostfixEvaluator {
    Stack<Double> stack;

    public PostfixEvaluator(Stack<Double> stack) {
        this.stack = stack;
    }

    public String processToken(String token) {
        ...}
    }
```
## Tokens and their meaning (1)

<table>
<thead>
<tr>
<th>Token</th>
<th>operation</th>
<th>return value (String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>quit</td>
<td>terminate calculator by throwing a TerminateException</td>
<td>empty string</td>
</tr>
<tr>
<td>exit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>null</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show</td>
<td>show top of stack</td>
<td>top of stack value</td>
</tr>
<tr>
<td>pstack</td>
<td>show stack</td>
<td>string representation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[a,b,c,...] of the stack</td>
</tr>
<tr>
<td>clear</td>
<td>clear the stack</td>
<td>empty string</td>
</tr>
<tr>
<td>a number</td>
<td>push onto stack</td>
<td>empty string</td>
</tr>
</tbody>
</table>
## Tokens and their meaning (2)

<table>
<thead>
<tr>
<th>Token</th>
<th>operation</th>
<th>return value (String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>pop y, pop x</td>
<td>empty string</td>
</tr>
<tr>
<td></td>
<td>push x + y</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>pop y, pop x</td>
<td>empty string</td>
</tr>
<tr>
<td></td>
<td>push x - y</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>pop y, pop x</td>
<td>empty string</td>
</tr>
<tr>
<td></td>
<td>push x * y</td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>pop y, pop x</td>
<td>empty string</td>
</tr>
<tr>
<td></td>
<td>push x / y</td>
<td></td>
</tr>
<tr>
<td>div</td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>pop y, pop x</td>
<td>empty string</td>
</tr>
<tr>
<td>pow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Each token's operation is performed on the stack, with the order of operations determined by their precedence.
## Tokens and their meaning (3)

<table>
<thead>
<tr>
<th>Token</th>
<th>operation</th>
<th>return value (String)</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>dup</em></td>
<td>duplicate top of stack</td>
<td>empty string</td>
</tr>
<tr>
<td></td>
<td>peek and push</td>
<td></td>
</tr>
<tr>
<td><em>exch</em></td>
<td>exchange top two elements of stack</td>
<td>empty string</td>
</tr>
<tr>
<td><em>pop</em></td>
<td>pop the top of stack</td>
<td>empty string</td>
</tr>
<tr>
<td>Token</td>
<td>operation</td>
<td>return value (String)</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>neg</td>
<td>negate number at top of stack</td>
<td>empty string</td>
</tr>
<tr>
<td>rad</td>
<td>set radians mode for trig functions</td>
<td>empty string</td>
</tr>
<tr>
<td>deg</td>
<td>set degrees mode for trig functions</td>
<td>empty string</td>
</tr>
<tr>
<td>mode</td>
<td>do nothing</td>
<td>&quot;degrees mode&quot; or &quot;radians mode&quot;</td>
</tr>
<tr>
<td>sqrt</td>
<td>pop x and push sqrt(x)</td>
<td>empty string</td>
</tr>
</tbody>
</table>
## Tokens and their meaning (5)

<table>
<thead>
<tr>
<th>Token</th>
<th>operation</th>
<th>return value (String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>pop x and push sin(x)</td>
<td>empty string</td>
</tr>
<tr>
<td>cos</td>
<td>pop x and push cos(x)</td>
<td>empty string</td>
</tr>
<tr>
<td>tan</td>
<td>pop x and push tan(x)</td>
<td>empty string</td>
</tr>
<tr>
<td>Token</td>
<td>operation</td>
<td>return value (String)</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>asin</td>
<td>pop x and push asin(x)</td>
<td>empty string</td>
</tr>
<tr>
<td>acos</td>
<td>pop x and push acos(x)</td>
<td>empty string</td>
</tr>
<tr>
<td>atan</td>
<td>pop x and push atan(x)</td>
<td>empty string</td>
</tr>
<tr>
<td>Token</td>
<td>operation</td>
<td>return value (String)</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>exp</td>
<td>pop x and push exp(x)</td>
<td>empty string</td>
</tr>
<tr>
<td>log10</td>
<td>pop x and push log(x), base 10</td>
<td>empty string</td>
</tr>
<tr>
<td>log2</td>
<td>pop x and push log(x), base 2</td>
<td>empty string</td>
</tr>
<tr>
<td>ln</td>
<td>pop x and push ln(x), base e</td>
<td>empty string</td>
</tr>
<tr>
<td>logb</td>
<td>pop x, pop b and push log to base b of x</td>
<td>empty string</td>
</tr>
</tbody>
</table>
Tokens and their meaning (8)

<table>
<thead>
<tr>
<th>Token</th>
<th>operation</th>
<th>return value (String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>push $\pi$ onto stack</td>
<td>empty string</td>
</tr>
<tr>
<td>e</td>
<td>push $e$ onto stack</td>
<td>empty string</td>
</tr>
<tr>
<td></td>
<td>$e$ is $\exp(1)$</td>
<td></td>
</tr>
</tbody>
</table>

If token is invalid (not in table) then throw an InvalidTokenException
You can use the following method to determine if a token is a number.

```java
private boolean isNumber(String token) {
    try {
        Double.valueOf(token);
        return true;
    }
    catch (NumberFormatException e) {
        return false;
    }
}
```

If this method returns true you can push `Double.valueOf(token)` onto the stack.
We can use the following class as a console interface to the PostfixEvaluator class.

```java
package stacksAndQueues.calculator;
import java.util.Scanner;
import stacksAndQueues.LinkedStack;
import stacksAndQueues.Stack;
import stacksAndQueues.EmptyStackException;

public class ConsolePostfixCalculator
{
    public static void main(String[] args)
    {
        new ConsolePostfixCalculator().run();
    }
}
```
The run method uses the Tokenizer class to read tokens one at a time until the TerminateException is thrown.

```java
public void run()
{
    Stack<Double> stack =
        new LinkedStack<Double>();

    PostfixEvaluator evaluator =
        new PostfixEvaluator(stack);

    Scanner input = new Scanner(System.in);

    // continued next slide
```
The infinite while loop will exit when the break statement is executed.

```java
while (true) {
    String token = input.next();
    String message = "";

    try {
        message =
            evaluator.processToken(token);
    }
```
catch (EmptyStackException e) {
    System.out.println("Error: empty stack");
}
catch (InvalidTokenException e) {
    System.out.println(e.getMessage());
}
catch (TerminateException e) {
    System.out.println(e.getMessage());
}
if (! message.equals"")
    System.out.println(message);
} // end while
} // end run
} // end ConsolePostfixCalculator
Using Scanner class

The `java.util.Scanner` class can be used to tokenize the standard input stream (System.in) for use in the console version of the postfix calculator using whitespace characters as delimiters.

```java
Scanner input = new Scanner(System.in);
while (input.hasNext())
{
    String token = input.next();
    // process token here
}
```

Another way is

```java
Scanner input = new Scanner(System.in);
while (true)
{
    String token = input.next();
    // process token here using a break statement to exit the loop
}
```
Write class GuiPostfixCalculator similar to GuiMatcher that provides a Gui interface to the calculator.
Of course the console tokenizer uses System.in and it won't work in a gui so just use the version of the Scanner class that gets its input from a string (line typed in the JTextField box)

```java
public void processLine()
{
    String line = input.getText().trim();
    output.append(line + "\n");
    Scanner sc = new Scanner(line);
    while (sc.hasNext())
    {
        String token = sc.next();
        // ...
    }
}
```
Variables

Variables can be incorporated into the postfix interpreter by using the following syntax:

- `/x 3 def` defines `x` to have the value 3
- `/x x 1 add def` defines `x` to have value 1 more than its previous value.

When `/x` is encountered `x` is saved as a symbol

When `def` is encountered the value on top of the stack is popped and stored under the name `x` in a symbol table.
A simple symbol table

A symbol table is a special kind of map that uses strings as keys to represent reserved keyword such as add and sub, and to represent variables and their values.

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mul</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>pi</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>2.342</td>
</tr>
<tr>
<td>height</td>
<td>1</td>
<td>-4.56</td>
</tr>
<tr>
<td>width</td>
<td>1</td>
<td>1.2e-5</td>
</tr>
</tbody>
</table>

This will be discussed in class
Syntax for variables and defs

- A variable definition begins with /s where s is some symbol and ends with the keyword def.
- A symbol s begins with an alphabetic character and is followed by zero or more characters which can be alphanumerically (alphabetic or digits).

Regular expressions

- "[a-zA-Z][a-zA-Z0-9]*" matches a symbol s
- "/[a-zA-Z][a-zA-Z0-9]*/" matches a definition /s

This will be discussed in class
Infix to Postfix conversion
Infix expressions (1)

One way to evaluate the usual algebraic expressions is to first convert them to postfix and then use a postfix interpreter to evaluate them.

An infix expression has the operators between the operands and a postfix expression has the operators after the operands.

- Infix: \( a + b \)
- Postfix: \( a \ b \ + \)
- Infix: \( (a + b) \times (c - d) \)
- Postfix: \( a \ b \ + \ c \ d \ - \ * \)
- Infix: \( (a + b\times(c + d\times(e-f))) \)
- Postfix: \( a \ b \ c \ d \ e \ f \ - \ * \ + \ * \ + \)
Infix expressions (2)

- Observe in the examples
  - Infix                      Postfix
    - a + b                    a b +
    - (a + b)*(c - d)          a b + c d - *
    - (a + b*(c + d*(e-f)))    a b c d e f - * + * +

- The order of the values does not change in the conversion process.
- Matching parentheses enclose subexpressions that can be evaluated independently.
- Infix operators have a precedence.
Operator precedence (1)

- Postfix expressions do not have parentheses so no operator precedence is required.
- Infix expressions are evaluated as follows
  - ( and ) have the highest precedence so the subexpressions within are evaluated first
  - * an / are evaluated in the usual left to right order so they have same precedence
  - + and - are evaluated next in the usual left to right order so they have same precedence but a lower one than * and /
### Operator precedence (2)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>3</td>
</tr>
<tr>
<td>)</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>2</td>
</tr>
<tr>
<td>/</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

We can use a stack to hold operators according to their precedence. As operators are encountered they are pushed onto the stack but not before popping any higher precedence operators already on the stack and appending them to the output postfix expression.

When an operand is encountered it is appended to the output postfix expression.
Infix to postfix conversion (1)

First consider the algorithm for infix expressions without any parentheses:

- **Operand encountered in infix expression**
  - it is directly appended to the postfix expression

- **Operator encountered in infix expression**
  - it is pushed on a stack but not before any higher precedence operators are popped and appended to the postfix expression
  - when the infix expression is exhausted any remaining operators are popped and appended to the postfix expression
Algorithm without ()

Create an empty stack and postfix expression
WHILE more tokens in infix expression DO
    token ← nextToken
    IF token is an operand THEN
        append token to the postfix expression
    ELSE IF token is an operator THEN
        WHILE (stack not empty) and
            (precedence(token) < precedence(top of stack)) DO
            pop operator and append to postfix expression
        END WHILE
        push token on stack
    ELSE
        error: invalid token
    END IF
END IF
END WHILE
pop any remaining operators and append to postfix expression

Higher precedence operators must be done first
### Example: \( a + b * c - d / e \)

<table>
<thead>
<tr>
<th>step</th>
<th>token</th>
<th>operation</th>
<th>stack</th>
<th>postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>[ ]</td>
<td>empty</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>append</td>
<td>[ ]</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>push</td>
<td>[ + ]</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>append</td>
<td>[ + ]</td>
<td>a b</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>push</td>
<td>[ * +]</td>
<td>a b</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>append</td>
<td>[ * +]</td>
<td>a b c</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>pop, append</td>
<td>[ + ]</td>
<td>a b c *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push</td>
<td>[ - + ]</td>
<td>a b c *</td>
</tr>
<tr>
<td>7</td>
<td>d</td>
<td>append</td>
<td>[ - + ]</td>
<td>a b c * d</td>
</tr>
<tr>
<td>8</td>
<td>/</td>
<td>push</td>
<td>[ / - + ]</td>
<td>a b c * d</td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td>append</td>
<td>[ / - + ]</td>
<td>a b c * d e</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>pop remaining</td>
<td></td>
<td>a b c * d e / - +</td>
</tr>
</tbody>
</table>

11/9/2006
Expressions with parentheses

- We add the following rules that ensure that parenthesized subexpressions are done first.
- If token is a left parenthesis push it on the stack to mark beginning of subexpression.
- If a right parenthesis is encountered pop subexpression and its left parenthesis.
- The while loop for popping operators of higher precedence needs to stop when a left parenthesis occurs.
Create an empty stack and postfix expression

WHILE more tokens in infix expression DO
  token ← nextToken
  IF token is an operand THEN
    append token to the postfix expression
  ELSE IF token is a left parenthesis THEN
    push it onto the stack as a marker
  ELSE IF token is a right parenthesis THEN
    WHILE top is not a left parenthesis DO
      pop operators and append to postfix expression
    END WHILE
    pop the left parenthesis

// see next slide
ELSE IF token is an operator THEN
    WHILE stack is not empty and
    top is not a left parenthesis and
    precedence(token) < precedence (top of stack) DO
        pop operator and append to postfix expression
    END WHILE
    push token on stack
ELSE
    error: invalid token
END IF
END WHILE
pop any remaining operators and append to postfix expression
### Example: \((a + b)*(c - d)\)

<table>
<thead>
<tr>
<th>step</th>
<th>token</th>
<th>operation</th>
<th>stack</th>
<th>postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(</td>
<td>push</td>
<td>[ ( ]</td>
<td>empty</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>append</td>
<td>[ ( ]</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td>push</td>
<td>[ + ( ]</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>append</td>
<td>[ + ( ]</td>
<td>a b</td>
</tr>
<tr>
<td>5</td>
<td>)</td>
<td>pop and append</td>
<td>[ ( ]</td>
<td>a b +</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pop left (</td>
<td>[ ]</td>
<td>a b +</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td>push</td>
<td>[ * ]</td>
<td>a b +</td>
</tr>
<tr>
<td>7</td>
<td>(</td>
<td>push</td>
<td>[ ( * ]</td>
<td>a b +</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td>append</td>
<td>[ ( * ]</td>
<td>a b + c</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>push</td>
<td>[ - ( * ]</td>
<td>a b + c</td>
</tr>
<tr>
<td>10</td>
<td>d</td>
<td>append</td>
<td>[ - ( * ]</td>
<td>a b + c d</td>
</tr>
<tr>
<td>11</td>
<td>)</td>
<td>pop and append</td>
<td>[ ( * ]</td>
<td>a b + c d-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pop left (</td>
<td>[ * ]</td>
<td>a b + c d-</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>pop remaining ops and append</td>
<td>a b + c d - *</td>
<td></td>
</tr>
</tbody>
</table>
**Example: a * ( b / c + d)**

<table>
<thead>
<tr>
<th>step</th>
<th>token</th>
<th>operation</th>
<th>stack</th>
<th>postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>[ ]</td>
<td>empty</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>append</td>
<td>[ ]</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>push</td>
<td>[ * ]</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>(</td>
<td>push</td>
<td>[ ( * ]</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>append</td>
<td>[ ( * ]</td>
<td>a b</td>
</tr>
<tr>
<td>5</td>
<td>/</td>
<td>push</td>
<td>[ / ( * ]</td>
<td>a b</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>append</td>
<td>[ / ( * ]</td>
<td>a b c</td>
</tr>
<tr>
<td>7</td>
<td>+</td>
<td>pop and append</td>
<td>[ ( * ]</td>
<td>a b c /</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push</td>
<td>[ + ( * ]</td>
<td>a b c /</td>
</tr>
<tr>
<td>8</td>
<td>d</td>
<td>append</td>
<td>[ + ( * ]</td>
<td>a b c / d</td>
</tr>
<tr>
<td>9</td>
<td>)</td>
<td>pop and append</td>
<td>[ ( * ]</td>
<td>a b c / d +</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pop left par</td>
<td>[ * ]</td>
<td>a b c / d +</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>pop remaining operators and append</td>
<td>[ ]</td>
<td>a b c / d + *</td>
</tr>
</tbody>
</table>
Trying it in Java

Create a string with no spaces containing single letters a - z for operands and single characters for the operators. Then use a for loop such as

```java
public String infixToPostfix(String infix) {
    Stack<Character> stack =
        new LinkedStack<Character>();
    StringBuilder postfix = new StringBuilder();
    try {
        for (int k = 0; k < infix.length; k++)
            { char ch = infix.charAt(k);
                ...
            }
    } catch (...) { ... }
}
```
Code slide

comment

Put code here