Questions for you

Answer these questions after class.

1. How to create(new) a Generic array?

1. How to check Delimiter Matching with stack?

1. What’s postfix expression and how to compute it?

The Stack Interface

```
public interface StackInterface<T> {
    void push(T element) throws
StackOverflowException;
    T pop()  throws StackUnderflowException;
    T peek() throws StackUnderflowException;
    boolean isEmpty();
    boolean isFull();
}
```

- We covered implementation of Stack data structure using a linked list. Now let’s implement it using an array.

Array-based Stack

```
public class ArrayStack<T> implements StackInterface<T> {
    // array-stack is bounded. Default capacity 100.
    protected final int DEFCAP = 100;
    protected T[] stack; // stores stack elements
    protected int topIndex=-1; // index of top element
}
```

- Other than this is a **generic** class, the data members are similar to `ArrayStringLog` that we've learned before.
- We use `topIndex` to point to the top element of the stack. It's initialized to `-1` indicating the stack is empty in the beginning. **Pushing and popping happen at the last element in the array.**
- When `(topIndex==stack.length-1)`, it has reached the capacity of the array, hence the stack is full.

Methods for ArrayStack

```
public boolean isEmpty() {
    return (topIndex == -1);
}
public boolean isFull() {
    return (topIndex == (stack.length-1));
}
public int size() {
    return topIndex + 1;
}
```

Constructors for ArrayStack

- **Conceptually**, we want to do this:

```
public class ArrayStack<T> implements StackInterface<T>
{
    … …
    public ArrayStack() {
       stack = new T[DEFCAP];
    }
}
```

Constructors for ArrayStack

- **Conceptually**, we want to do this:

```
public class ArrayStack<T> implements StackInterface<T>
{
    … …
    public ArrayStack() {
      stack = new T[DEFCAP];
    }
}
```

- **Unfortunately**, Java **does not** allow you to create a generic array (i.e. with a to-be-decided type).***

*** http://www.angelikalanger.com/Articles/Papers/JavaGenerics/ArraysInJavaGenerics.htm

Casting to a Generic Type

- The work-around is to create an array of type **`Object`** (i.e. the super-class of all Java objects), and **explicitly cast** the resulting array into type **`T[]`**.
  - The compiler will **warn** us about Type Safety (that it cannot be sure the elements stored in the array are of type T), but we can ignore this warning.
- This is ok because an array of objects is an array of references (i.e. each element is a pointer). Whether it's an array of Strings, Apples, Objects, the array takes the same amount of memory space.

Code for the Constructors

```
public ArrayStack(int maxSize) {
    stack = (T[]) new Object[maxSize];
}

public ArrayStack( ) {
    this(DEFCAP);
}
```

- As before we have two constructors, one with no parameter thus using the default capacity, and one with a custom capacity as parameter.
- Also note that the name of **the constructor does not contain the `<T>` part**.

Code for peek(), pop()

```
public T peek() {
  if (isEmpty())
    throw new StackUnderflowException(“underflow”);
  return stack[topIndex];
}
```

```
public T pop() {
  if (isEmpty())
    throw new StackUnderflowException(“underflow”);
  return stack[topIndex--];  <--What’s this thing?
}
```

Code for pop()

```
public T pop() {
  if (isEmpty())
    throw new StackUnderflowException(“underflow”);
  return stack[topIndex--];
}
```

This is equivalent to:

```
public T pop() {
  if (isEmpty())
    throw new StackUnderflowException(“underflow”);
  T element = stack[topIndex];
  topIndex--;
  return element;
}
```

Code for push()

```
public void push (T element) {
  if (isFull())
    throw new StackOverflowException(“overflow”);
  stack[++topIndex] = element;
}
```

Equivalent to:

```
public void push (T element) {
  if (isFull())
    throw new StackOverflowException(“overflow”);
  topIndex++; // or ++topIndex; is also fine
  stack[topIndex] = element;
}
```

Clicker Question #1

Which method's big-O costs is O(n) in ArrayStack (assuming the number of stack elements is **n**)?

(a) peek and pop

(b) push

(c) isEmpty and isFull

(d) size

(e) None of these

Stack Application 1
Reversing a Word

- Say you are to write a program that accepts a word, and outputs the word in reverse order.
  - `STRESSED -> DESSERTS`
- This can be done easily using a stack:
  - Push each character one by one to stack
  - Pop the stack one by one

Application 1 — Reversing a Word

```
public void reverse(String word) {
    Stack s = new Stack();
    for (int i=0; i<word.length; i++)
        s.push(word.charAt(i));
    while (!s.isEmpty())
        System.out.print(s.pop());
}
```

Application 2 — Delimiter Matching

- You want to write a program to make sure the parentheses in a math expression are balanced:
  - `(w * (x + y) / z - (p / (r - q) ) )`
- It may have several different types of delimiters: **`braces{}, brackets[], parentheses()`**

Each opening (left) delimiter must be matched by a corresponding closing (right) delimiter.

A delimiter that opens the last must be closed by a matching delimiter first. For example,
`[a * (b + c] + d )` is wrong!

Application 2 — Delimiter Matching

- Think for a moment about how you would implement this algorithm.

Application 2 — Delimiter Matching

- Read characters one-by-one from the expression.
- Whenever you see a **left** (opening) delimiter, **push** it to stack.
- Whenever you see a **right** (closing) delimiter, **pop** from stack and check **match** (i.e. same type?)
- If they don’t match, report mismatch error.
- What happens if the stack is **empty** when you try to match a closing delimiter?
- What happens if the the stack is **non-empty** after you reach to the end of the expression?

Clicker Question #2

What happens when using the delimiter matching algorithm to parse the following expression (assume the stack is unbounded):

```
a { b [ c { d } ( e ) ]] f
```

(a) stack underflow exception

(b) stack overflow exception

(c) delimiter mismatch error

(d) stack is non-empty after it’s done parsing

(e) there is no error: the expression is valid.

Stack Application: Postfix Expression

- Goal: evaluate arithmetic expressions.
- **Terminology**:
  - Operands: numbers
  - Operators: `+ - / *`
- For now, we focus on **binary** operators (i.e. each operator takes two numbers as input).
- **Infix notation**: operators are placed between two operands. This is what we are familiar with.
  - **`(2 + 14) * 23`**

Postfix Expression

- **Postfix notation**: operators are placed after operands.
  - `5 3 -` `// 5 - 3`
  - `A B /` `// A / B`
  - `2 14 + 23 *` `// (2+14)*23`
- **An operator acts on the two values to its left**, where a value may be either a number in the original expression or result of a previous operator.
- Note the order of the operands (**further left -> operator -> closer left**). This matters to – and /.

Postfix Expression

- **Postfix notation**: operators are placed after operands.
  - `5 3 -` `// 5 - 3`
  - `A B /` `// A / B`
  - `2 14 + 23 *` `// (2+14)*23`
- Also known as **RPN (Reverse Polish Notation)**
- There are RPN calculators You can buy, they are fun to play with

Additional Examples

Infix	Postfix
A+B–C	AB+C–
A*B/C	AB*C/
A+B*C	ABC*+
A*B+C	
A*(B+C)	
A*B+C*D	
(A+B)*(C–D)	
((A+B)*C)–D	
A+B*(C–D/(E+F))	

Additional Examples

Infix	Postfix
A+B–C	AB+C–
A*B/C	AB*C/
A+B*C	ABC*+
A*B+C	AB*C+
A*(B+C)	ABC+*
A*B+C*D	AB*CD*+
(A+B)*(C–D)	AB+CD–*
((A+B)*C)–D	AB+C*D–
A+B*(C–D/(E+F))	ABCDEF+/–*+

Evaluate Postfix Expression

- How do we write an algorithm to evaluate postfix expressions? Note that whenever you encounter an operator, you apply it to the last two operands. This suggests using a **stack to store the operands**.
- Scan the expression
- When we see an **operand**, push it onto the stack.
- When we see an **operator**, pop two values off of the stack, and apply the operator to them, then push the result back onto the stack.
- In the end, if we have exactly one value on the stack, that is our answer.

Evaluate Postfix Expression

- Let's simulate the algorithm for the expression:

`13 5 4 - 2 * 2 4 * + -`

- If we ever encounter an empty stack, or if we finish with more than one value on it, the postfix expression was not valid. Thus our algorithm can also test the validity of an input postfix expression.

Evaluate Postfix Expression

- Examples of invalid postfix expression:
  - `1 + *`
  - `1 2 3 +`
- What would happen in our algorithm for the above two expressions?

Clicker #3

- What's the value of this postfix expression (assuming integer division)

6 2 / 3 5 1 - * +

(a) 1

(b) 13

(c) 15

(d) -13

(e) -15

Pseudo-Code for Postfix

```
while more token exist {
    get the next token
    if token is an operand:
        stack.push(token);
    else { // token is operator
        operand2 = stack.pop();
        operand1 = stack.pop();
        result = (operand1) token (operand2);
        stack.push(result);
   }
}
return stack.pop();
```

The code above omits error detection and handling.

Comparing Running Time

- Will any of our methods take longer for a stack with many elements than for a stack with few elements?

- In other words, if N is the number of elements in the stack, what is the running time of each of our methods as a function of N (assuming N is large)?

- Does it make a difference which implementation we use: `ArrayStack` or `LinkedStack`?

Comparing Running Time

- In fact all of `push`, `pop`, `peek` methods take O(1) time in each case.

- In `ArrayStack`, pushing and popping each involve altering one element at the back of the array and changing topIndex, overall `O(1)`.

- In `LinkedStack`, we only need a `O(1)` pointer operations around the top node.

Clicker Question #4

Suppose we implement **`ArrayStack`** with an array a where **`a[0]`** is always the top element (in other words, elements are pushed and popped at the front of the array, NOT the end). What would be the big-O costs of **`push, pop`** and **`peek`** (N is the number of stack elements)?

(a) O(1) for all three

(b) O(1) for peek, O(N) for push and pop

(c) O(1) to push and peek, O(N) to pop

(d) O(1) to pop and peek, O(N) to push

Java's `ArrayList` Class

- It would be nice if the array capacity is not fixed!
- At the end of the Linked List lecture, we briefly mentioned arrays that can dynamically expand, thus overcoming the 'fixed capacity' limitation.
- Java provides several variable-length generic array structures, such as `ArrayList<T>`. It begins with some fixed capacity, but the capacity is reached, it copies itself into another array of twice the size, thus *appears* to be an array of unlimited size.

Java's `ArrayList` Class

- What's this doubling-the-capacity business?
  - Let's say we have an initial ArrayList of capacity 10, and we keep adding elements to it.
  - Adding the 11th element causes the ArrayList to allocate a new array of capacity 20, copy the existing 10 elements and the 11th element to the new array, then release the old array.
- By the time we have 81 elements we have done four capacity 'expansions': to 20, 40, 80, and 160.
- We will analyze this later, but it is O(n)….

Questions?