CSC 20 Chapter 14: Stacks and Queues

There are two very common ways of organizing data as you get it. Keep it in the order it was received or keep it in the reverse order it was received.

Stack

Last-in-first-out (LIFO). If you want to access your data in the opposite order it was received, you want to use a stack. It's called a stack because stacks behave this way: the item on the top of your stack of papers is the last piece of paper you put there. The bottom of the stack is the oldest piece of paper.

LIFO is common in real life:

- When layoffs come, it's often the most recently hired person who is released first.
- The top plates at the cafeteria are the most recently washed.
- People exit elevators in LIFO order.
- Online content (blogs, news, Facebook, etc) is organized most recent on top.
- Vomit.

An API for a stack is easy:

- push(x): put x on the top of the stack.
- pop(): remove and return what is top of the stack.
- peek(): return what is top of the stack, but don't remove it.
- isEmpty(): return `true` if and only if the stack is empty.

In fact, stacks are so easy to implement (given a `List` implementation) that a complete one can be written right here.

```
public class Stack<E> {
    // fields
    private List<E> list;
    // methods
    public         Stack()      { list = new ArrayList<E>();         }
    public void    push(E item) { list.add(item);                    }
    public E       pop()        { return list.remove(list.size()-1); }
    public E       peek()       { return list.get(list.size()-1);    }
    public boolean isEmpty()    { return list.isEmpty();             }
}
```

Example usage: Solving a maze.

Example: Exponentiation.

```
import java.util.Stack;

class Power {

    public static int pow(int base, int exp) {
        Stack<Boolean> stack = new Stack<Boolean>();
        while (exp!=0) {
            stack.push(exp%2==1);
```

```
            exp = exp/2;
        }
        int res=1;
        while ( ! stack.isEmpty() ) {
            res = res * res;
            if (stack.pop()) {
                res = res * base;
            }
        }
        return res;
    }


    public static void main(String[] args) {
        System.out.println(pow(2,10));        // Prints 1024
    }
}
```

Example: Postfix expression evaluator.

```
import java.util.*;

public class Expr {

    public static int eval(Scanner in) {
        Stack<Integer> s = new Stack<Integer>();
        while (in.hasNext()) {
            if (in.hasNextInt()) {
                s.push(in.nextInt());
            } else {
                String op = in.next();
                if (op.equals("+")) {
                    s.push(s.pop()+s.pop());
                } else if (op.equals("*")) {
                    s.push(s.pop()*s.pop());
                } else {
                    throw new RuntimeException();
                }
            }
        }
        if (s.size()==1) {
            return s.pop();
        } else {
            throw new RuntimeException();
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        while (true) {
            try {
                System.out.print("> ");
                int res = eval(new Scanner(in.nextLine()));
                System.out.println(res);
```

```
            }
            catch (Exception e){
                System.out.println("Parse error");
            }
        }
    }
}
```

Queue

First-in-first-out (FIFO). If you want to access your data in the order it was received, you want to use a queue. It's called a queue because it behaves like standing in line (queue is British for a line of people): The first person in line is the first person served and the last person in line is the last person served.

FIFO is common in real life:

- Waiting in line for anything.
- Voicemail.
- Stock at stores (they try to sell older stuff before newer stuff).

An API for a queue is easy:

- add(x): put x at the back of the queue.
- remove(): remove and return what is at the front of the queue.
- peek(): return what is at the front of the queue, but don't remove it.
- isEmpty(): return `true` if and only if the queue is empty.

In fact, queues are so easy to implement (given a `List` implementation) that a complete one can be written right here.

```
public class Queue<E> {
    // fields
    private List<E> list;
    // methods
    public         Queue()      { list = new LinkedList<E>(); }
    public void    add(E item)  { list.add(item);             }
    public E       remove()     { return list.remove(0);      }
    public E       peek()       { return list.get(0);         }
    public boolean isEmpty()    { return list.isEmpty();      }
}
```

Example use: Find the shortest number of flight segments from one city to another.

Java Collections

`Stack<E>` in Java is a class.

```
Stack<String> s = new Stack<String>();
```

`Queue<E>` in Java is an interface, and `LinkedList` implements it.

```
Queue<String> q = new LinkedList<String>();
```

For Practice-It, you will need to use these types because they are the same ones the book explains. For your own work, however, you should prefer to use `Deque`, which Java recommends for new applications.

For a stack, use `addFirst` instead of `push` and `removeFirst` instead of `pop`. For a queue, use `addLast` instead of `add` and `removeFirst` instead of `remove`. In either case, `getFirst` and `isEmpty` work as expected.

Practice-It Exercise 14.20: interleave.

Write a method interleave that accepts a queue of integers as a parameter and rearranges the elements by alternating the elements from the first half of the queue with those from the second half of the queue

```
public static void interleave(Queue<Integer> q) {
    int n = q.size();
    if (n%2==1)
        throw new IllegalArgumentException();
    Queue<Integer> aux = new LinkedList<Integer>();
    for (int i=0; i<n/2; i++) {
        aux.add(q.remove());
    }
    for (int i=0; i<n/2; i++) {
        q.add(aux.remove());
        q.add(q.remove());
    }
}
```