

CS231 Fall 2013 Lecture 22

Stephanie R Taylor

October 25, 2013

1 Administrative Topics

- We take the quiz.
- Check out Lecture 11 notes for more on exceptions.

2 Stacks

Interface:

```
public interface Stack {  
    public int size();  
    public boolean isEmpty();  
    public Object top();  
    public void push(Object o);  
    public Object pop();  
}
```

Questions:

- Should we be allowed to put null onto a stack of objects?
- Can we add the same object twice to the stack?
- Can a stack become full?

- Which methods can be implemented as convenience methods? [peek and isEmpty]
- What should happen if we pop or peek an empty stack?

2.1 Implementation of Stacks

I want to go over the details of implementations of stacks

1. using arrays, and
2. using lists.

2.1.1 Implementing a stack with an array

The idea is that we store the items in an array, with the bottom of the stack at the 0^{th} element. We add elements at increasing indices and remove from the top (element with largest index). We begin with an initial capacity for the array, then expand as needed.

```
public class ArrayStack implements Stack {
    private int count;
    private Object[] data;

    public ArrayStack() {
        count = 0;
        data = new Object[10];
    }

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

    public int size() {
        return count;
    }

    public void push(Object obj) {
        if( count == data.length ) {
            // expand the array and copy over into it
            ....
        }
    }
}
```

```

        }
        data[count] = obj;
        count++;
    }

    public Object pop() throws EmptyStackException {
        if (count == 0) throw new EmptyStackException();
        count--;
        return data[count];
    }

    public Object peek() throws EmptyStackException {
        if (count == 0) throw new EmptyStackException();
        return data[count - 1];
    }
}

```

2.1.2 Implementing a stack with a linked list

We can do this with a singly-linked list. All we need to do is add and remove from the head.

```
public class NodeStack implements Stack {
```

For the list implementation, we need a Node that has an Object as cargo. We will nest it in the NodeStack class :

```

    private class Node {
        Object data;
        Node next;

        Node(Object d, Node n) {
            this.data = d;
            this.next = n;
        }
    }
}

```

The instance variable here is called topNode and it just point to the element on the top of the stack:

```

    private Node topNode;

    public NodeStack() {
        topNode = null;
    }
}

```

The constructor creates an empty stack by initializing the instance variable to null. So naturally we can test for an empty stack as so...

```
public boolean isEmpty() {  
    return (topNode == null);  
}
```

Now, for putting things on and taking things off the list, we could have used the list methods we developed in the last week:

push is the same as add:

```
public void push(Object obj) {  
    topNode = new Node (obj, topNode);  
}
```

pop is the same as removeFirstNode, except that we have to play some games with a temporary variable:

```
public Object pop() throws EmptyStackException {  
    if (topNode == null) throw new EmptyStackException();  
    Object ret = topNode.data;  
    topNode = topNode.next;  
    return ret;  
}
```

peek is a no-brainer.

```
public Object peek() throws EmptyStackException {  
    if (topNode == null) throw new EmptyStackException();  
    return topNode.data;  
}
```