Introduction to Object-Oriented Programming

`Stacks and Queues`

Christopher Simpkins

chris.simpkins@gatech.edu

Stacks and Queues

- Stacks
- Queues
- Design Exercise

What is a stack?

[1]

Fat stacks

[1] Source: http://blogs.amctv.com/breaking-bad/photo-galleries/breaking-bad-season-5-episode-photos/

What is a stack?

[2]

Tasty stacks

[2] Source:"Silver dollar pancakes" by Ehedaya at en.wikipedia - Own work (Original caption: "self-made"). Licensed under Public domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Silver_dollar_pancakes.

Gratuitous Super Troopers

 [3]

[3] Source: http://sumidiot.blogspot.com/2010/05/super-troopers.html

Stack ADT

Data:

- a list of elements

A *stack* is a LIFO (last in, first out) data structure with two defining operations:

- *push* adds an element to the stack
- *pop* returns and removes the most recently added element from the stack

A stack may also have

- an *isEmpty* operation, which is good style but not strictly necessary.
- a *peek* operation, which returns the next element to be removed from the stack with a pop operation but does not remove it.

ArrayList Stack Implementation

A stack can be implemented easily using `ArrayList`

- *push* adds elements to the end of the `ArrayList`.
- *pop* removes and returns the last element in the `ArrayList`.
- *isEmpty* delegates to `ArrayList`'s `isEmpty` method.

The entire implementation (as an inner class) is:

```
static class Stack<E> {
    private ArrayList<E> elems = new ArrayList<>();

    public void push(E item) {
        elems.add(item);
    }
    public E pop() {
        return elems.remove(elems.size() - 1);
    }
    public boolean isEmpty() {
        return elems.isEmpty();
    }
}
```

See ArrayListDataStructures.java.

Linked Stack Implementation

Here's a stack implemented with `Node`s.

```
public class LinkedStack<E> {
    private class Node<E> {
        E data;
        Node<E> next;

        Node(E data, Node<E> next) { this.data=data; this.next=next; }
    }
    private Node<E> head;

    public void push(E item) {
        head = new Node<E>(item, head);
    }
    public E pop() {
        E answer = head.data;
        head = head.next;
        return answer;
    }
    public boolean isEmpty() { return (head == null); }
}
```

Look familiar? See LinkedStack.java.

Queue ADT

Data:

- a list of elements

A *queue* is a FIFO (first in, first out) data structure with two defining operations:

- *enqueue* adds an element to the queue
- *dequeue* returns and removes the least recently added element from the queue

A queue may also have

- an *isEmpty* operation, which is good style but not strictly necessary.
- a *peek* operation, which returns the next element to be removed from the queue with a dequeue operation but does not remove it.

ArrayList Queue Implementation

A queue can be implemented easily using `ArrayList`

- *enqueue* adds elements to the end of the `ArrayList`.
- *dequeue* removes and returns the first element in the `ArrayList`.
- *isEmpty* delegates to `ArrayList`'s `isEmpty` method.

The entire implementation (as an inner class) is:

```
static class Queue<E> {
    private ArrayList<E> elems = new ArrayList<>();

    public void enqueue(E item) {
        elems.add(item);
    }
    public E dequeue() {
        return elems.remove(0);
    }
    public boolean isEmpty() {
        return elems.isEmpty();
    }
}
```

See ArrayListDataStructures.java.

Linked Queue Implementation

```
public class LinkedQueue<E> {
    private class Node<E> ...
    private Node<E> head;
    private Node<E> last;

    public void enqueue(E item) {
        Node<E> newNode = new Node<E>(item, null);
        if (null == head) head = newNode;
        if (null != last) last.next = newNode;
        last = newNode;
    }
    public E dequeue() {
        E answer = head.data;
        head = head.next;
        return answer;
    }
    public boolean isEmpty() { return (head == null); }
}
```

Essentially same as `LinkedStack`, except we maintain a `last` reference and add elements to the end intead of the head. See LinkedQueue.java.

Comparing ArrayList and Linked Implementations

Here, again, is the `dequeue` method in `ArrayListQueue`:

```
private ArrayList<E> elems = new ArrayList<>();
public E dequeue() {
    return elems.remove(0);
}
```

And here is the `dequeue` method in `LinkedQueue`:

```
public E dequeue() {
    E answer = head.data;
    head = head.next;
    return answer;
  }
```

- What is the Big-O of the `dequeue` method in `ArrayListQueue`?
  - $\mathcal{O}(n)$.
- What is the Big-O of the `dequeue` method in `LinkedQueue`?
  - $\mathcal{O}(1)$.

Comparing ArrayList and Linked Implementations

Here, again, is the `dequeue` method in `ArrayListQueue`:

```
private ArrayList<E> elems = new ArrayList<>();
public E dequeue() {
    return elems.remove(0);
}
```

And here is the `dequeue` method in `LinkedQueue`:

```
public E dequeue() {
    E answer = head.data;
    head = head.next;
    return answer;
  }
```

- What is the Big-O of the `dequeue` method in `ArrayListQueue`?
  - $\mathcal{O}(n)$.
- What is the Big-O of the `dequeue` method in `LinkedQueue`?
  - $\mathcal{O}(1)$.

Comparing ArrayList and Linked Implementations

Here, again, is the `dequeue` method in `ArrayListQueue`:

```
private ArrayList<E> elems = new ArrayList<>();
public E dequeue() {
    return elems.remove(0);
}
```

And here is the `dequeue` method in `LinkedQueue`:

```
public E dequeue() {
    E answer = head.data;
    head = head.next;
    return answer;
  }
```

- What is the Big-O of the `dequeue` method in `ArrayListQueue`?
  - $\mathcal{O}(n)$.
- What is the Big-O of the `dequeue` method in `LinkedQueue`?
  - $\mathcal{O}(1)$.

Comparing ArrayList and Linked Implementations

Here, again, is the `dequeue` method in `ArrayListQueue`:

```
private ArrayList<E> elems = new ArrayList<>();
public E dequeue() {
    return elems.remove(0);
}
```

And here is the `dequeue` method in `LinkedQueue`:

```
public E dequeue() {
    E answer = head.data;
    head = head.next;
    return answer;
  }
```

- What is the Big-O of the `dequeue` method in `ArrayListQueue`?
  - $\mathcal{O}(n)$.
- What is the Big-O of the `dequeue` method in `LinkedQueue`?
  - $\mathcal{O}(1)$.

Design Exercise

Our data structures implement the core elements of their ADTs, but there are some problems from an OO design standpoint.

- What happens if you call `pop` on an empty `ArrayListStack`?
- What happens if you call `pop` on an empty `LinkedStack`?
- What if you start off using an `ArrayListStack` but then decide to switch to using a `LinkedStack`?

Designing Error Reports

Calling `pop` on an empty `ArrayListStack` results in:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
```

Calling `pop` on an empty `LinkedStack` results in:

```
Exception in thread "main" java.lang.NullPointerException
```

There are two problems with these error reports:

- They leak implementation details across an abstraction boundary - why should a user know that a stack is implemented using arrays?
- They don't report the actual user error that caused the exception - calling `pop` on an empty stack.

We can fix these design problems by throwing `java.util.EmptyStackException` in the `pop` methods if the stack is empty.

A Stack Interface

We could have both of our implementations implement a `Stack` interface:

```
public interface Stack<E> {

    public void push(E item);

    public E pop() throws java.util.EmptyStackException;

    public abstract boolean isEmpty();
}
```

Is there a problem with this approach?

- `java.util.EmptyStackException` is-a `RuntimExecption`, which is not checked, so implementing classes will not be required to declare it.

A Stack Interface

We could have both of our implementations implement a `Stack` interface:

```
public interface Stack<E> {

    public void push(E item);

    public E pop() throws java.util.EmptyStackException;

    public abstract boolean isEmpty();
}
```

Is there a problem with this approach?

- `java.util.EmptyStackException` is-a `RuntimExecption`, which is not checked, so implementing classes will not be required to declare it.

AbstractStack

Abstract classes to the rescue!

```
public abstract class AbstractStack<E> implements Stack<E> {
    public final E pop() {
        if (isEmpty()) { throw new java.util.EmptyStackException(); }
        return removeNext();
    }
    protected abstract E removeNext();
}
```

- This `pop` method will be the one and only pop method used by subclasses (because it's final), ensuring that `java.util.EmptyStackException` is thrown as we want.
- Subclasses must implement `removeNext()`, which does what their `pop` methods used to do and is not visible to clients because it's `protected`.

So all we have to do is extend `AbstractStack` and change the name of our `pop` methods to `removeNext`.

Closing Thoughts

Today we

- learned about two basic data structures: stacks and queues,
- learned about alternative data structure implementations,
- applied exception programming principles,
- designed an OO family of stack classes, and
- used Java langauge features (like abstract classes and methods, final methods, and protected methods) to implement our OO stack family design.

Programming Exercise

A string is said to have balanced parentheses if for every open paren there is a matching close paren that comes after it, and no closing paren occurs before a corresponding open paren. This is an example of a string with balanced parentheses:

```
(map (lambda (x) (* x x)) (list 1 2 3 4))
```

and this is an example of unbalanced parentheses:

```
(map (lambda (x) (* x x)) (list 1 2 3 4)))
```

- Write a method `public static boolean hasBalancedParens(String s)` that returns `true` if `s` contains balanced parentheses, `false` otherwise.
- Write a method `public static boolean isBalanced(String s)` that checks for balanced "parentheses" of many types, for example, `( [ ] ) { }` is balanced, but `[ { ] }` is not.