Quiz 1 next week!

Please remember to bring your T-Card

**LO101**: Monday February 6, at 12 PM  
Surnames beginning with A-H: HA401  
Surnames beginning with I-Z: HA403  
If you haven’t been to Haultain before, please arrive a few minutes early.  
It can be confusing to get in.

**L0201**: Monday February 6, at 1 PM, EX200  
**L5101**: Wednesday February 8, at 5 PM, MS 3154  
Each quiz is different. All versions test a subset of the material listed on the "Practice Quiz" (see the "Readings" column of the "Lectures" link on the course website). We recommend that you look at this when studying for the quiz.

Assignment 2 posted

(There was an emailed announcement about this. I hope you got it. If you didn’t, check your email settings on the Portal.  
**Announcements are required reading.**)  
Setup can be finicky. Get it done now. Search for help on the DB, and come to one of the Getting help hours if you need it.  
Working on A2 will probably help you study for quiz 1.  
Read the Javadoc slides on the Lectures page.

Monday mini-lecture

Colin Chartier is going to be teaching a regular mini-lecture series, covering material that is relevant to CSC207. These lectures will occur Mondays at 2pm.
Interfaces

Stacks have push, pop, and isEmpty methods. There are lots of implementations — array, ArrayList, LinkedList, among others. We can describe how all of them work using an interface.

```java
/** A LIFO data structure. */
public interface Stack {
/** Add o to the top of this Stack.
 * @param o The object to be pushed.
 */
void push(Object o);
/** Remove and return the top item of this Stack.
 * @return the former top item of the this Stack.
 */
Object pop();
/** Return the top item of this Stack.
 * @return the top item of the this Stack.
 */
Object top();
/** Return whether this Stack is empty. */
boolean isEmpty();
}
```

A Stack implementation

```java
/** A Stack with fixed capacity. */
public class ArrayStack implements Stack {
/** The index of the top element in this Stack. Also the number. */
private int top;
/** contents[0 .. top-1] contains the elements in this Stack. */
private Object[] contents;
/** An ArrayStack with capacity for n elements. */
public ArrayStack(int n) {
    contents = new Object[n];
}
/** Add o to the top. (Ignore that we might overflow.) */
public void push(Object o) {
    contents[top++] = o;
}
/** Remove and return the top element of this Stack. */
public Object pop() {
    return contents[--top]; // What if top is 0?
}
/** Return true iff this Stack is empty. */
public boolean isEmpty() {
    return top == 0;
}
```

Using a Stack

You can’t create instances of interfaces. This is broken:

```java
Stack s = new Stack(15);
```

But you can write methods that use an interface:

```java
/**
 * Fill a stack with the integers 0 to n - 1 (inclusive),
 * with n - 1 at the top.
 * @param the Stack to fill
 * @param n the number of integers to put into the stack
 */
public static void fill(Stack s, int n) {
    for (int i = 0; i < n; i++) {
        s.push(new Integer(i));
    }
}
```

That function will work with any class that implements Stack. You should think of it as a service: it does work for anyone who needs their Stack filled with integers.
Queues (as an intro to generics)

Queue ops: enqueue, head, dequeue, size. Let’s also decide that all items in a queue must be the same type.

```java
/** A queue where all items must be of type T. */
public interface Queue<T> {
    /** Append o to me. */
    void enqueue(T o);
    /** Return my front item.
    * Precondition: size() != 0.
    */
    T head();
    /** Remove and return my front item.
    * Precondition: size() != 0.
    */
    T dequeue();
    /** Return my number of items. */
    int size();
}
```

```java
public class LinkedListQueue<T> implements Queue<T> {
    /** The items in me.  Head is index 0, tail is index size() - 1. */
    private LinkedList<T> contents = new LinkedList<T>();
    @Override
    public void enqueue(T item) {
        contents.add(item);
    }
    @Override
    public T head() {
        return contents.get(0);
    }
    @Override
    public T dequeue() {
        return contents.removeFirst();
    }
    @Override
    public int size() {
        return contents.size();
    }
}
```

```java
class QueueDemo {
    public static void fill(Queue<Integer> queue, int num) {
        for (int i = 0; i != num; i++) {
            queue.enqueue(i);
        }
    }
    public static void main(String[] args) {
        // Here is where we decide which Queue implementation to use.
        Queue<Integer> queue = new LinkedListQueue<>();
        fill(queue, 10);
        System.out.println(queue);
    }
}
```

Generics

"class Foo<T>" introduces a class with a type parameter T.

"<T extends Bar>" introduces a type parameter that is required to be a descendant of the class Bar — with Bar itself a possibility. In a type parameter, "extends" is also used to mean "implements".

"<? extends Bar>" is a type parameter that can be any class that extends Bar. We’ll never refer to this type, so we don’t give it a name.

"<? super Bar>" is a parameter that can be any ancestor of Bar.

The details of anything but the first one will not be tested in the quizzes or the exam, but you can use them on A2 and the project if you want.