Announcements

- Program 4
    - Due Sunday 11/10/19 , 11:59PM
- Midterm 2
    - Wednesday 11/6
- Veterans Day
    - Monday 11/11 - No Class
    - This means **no quiz** on Wednesday 11/13

Midterm 2 Review

- Review Question 1
- Review Question 2
- Questions?

Ch 8 - Inheritance / Polymorphism

Sections: 3, 5, 7

Ch 9 - Exceptions

Sections: 1, 2, 3, 4

Ch 10 - Streams (File I/O)

Sections: 1, 2, 3, 4, 5

Ch 11 - Abstract Classes / Interfaces

Sections: 1, 2, 4, 5

Ch 12 - Generics

Sections: 1, 2, 3

Stacks and Queues

Stacks and queues are both data structures that are often taught together due to their relationship as opposites.

To understand how they are opposites, you must first understand LIFO vs FIFO

- Last In First Out (LIFO) - the last element into the data structure must be the first one to be removed (also known as FILO - First In Last Out)
- First In First Out (FIFO) - the first element into the data structure must be the first one to be removed

Stack and Queue (cont.)

Stack analogy - Stacking concrete blocks

- Cannot pull the blocks from the middle or bottom of the stack
- If want to remove a block, must pull off the block at the top, one at a time...
- In other words, the block at the top was the last in, and is therefore the first out

Queue analogy - Waiting in a line

- The person at the front of the line is the person who has been in the line the longest (i.e. the first to enter the line, First In), and accordingly is the first to leave the line (i.e. First Out).

Stack and Queue (cont.)

A **stack** is a data structure in which items are only inserted on or removed from the top of a stack.

- The **push** operation inserts an item on the top of the stack.
- The **pop** operation removes and returns the item at the top of the stack.
- The **peek** operation returns the item at the top of the stack without removing actually it.
- A stack can be implemented using a linked list or an array (or an arraylist)

Let’s look at an example of a stack implementation on the whiteboard….

Stack and Queue (cont.)

A **queue** is a data structure in which items are inserted at the end of the queue and removed from the front of the queue.

- The **push** operation inserts an item at the end of the queue (Sometimes this is referred to as “enqueue”)
- The **pop** operation removes and returns the item at the front of the queue. (Sometimes this is referred to “dequeue”)
- The **peep** operation returns but does not remove the item at the front of the queue.
- A queue can be implemented using a linked list, or an array (or an arraylist)

Let’s look at an example of a stack implementation on the whiteboard….

Stacks and Queues (cont.)

Let’s look at java’s documentation for Stacks and Queues, you will notice some differences in the method names and how they might be handled.

Collections

How do we loop through the elements in a Collection?

More specifically, looking at the data structures we know about (Linked List, Array List, Stack, and Queue) how do we traverse through all of the data if we are from the perspective of **USING** the data structure?

A naive approach might be to use the “toArray” method and traverse through the elements as you would any normal array

Let’s refer back to the collections interface...

Enhanced For Loop (For Each Loop)

The answer is by using an “**Enhanced For Loop**”, also known as a “**For Each Loop**”

Given some ArrayList strings called bookTitles, here is an example of an enhanced for loop:

```
for (String bookTitle : bookTitles) {
	System.out.println(bookTitle);
}
```

Enhanced For Loops (cont.)

One way to think of the enhanced for loop is: “*for each element in a data structure*”

For example, using the previous example, you could say: “*for each book title in a collection of book titles*”

```
// For each bookTitle in bookTitles
for (String bookTitle : bookTitles) {
	System.out.println(bookTitle);
}
```

Enhanced For Loops (cont.)

Notice the unique for loop syntax:

```
for (String bookTitle : bookTitles) {
        System.out.println(bookTitle);
}
```

The enhanced for loop declares a new variable that will be assigned with each successive element of a container object, such as an array or ArrayList.

Let’s look at an example...

Enhanced For Loop (cont.)

An enhanced for loop increases **readability of code** as well as provides a nice level of **convenience** when working with data structures.

If you make your own data structure, does this mean anyone using your data structure can use an enhanced for loop on it?

The answer is NO.

The enhanced for loop is *syntactic sugar* for using something called an: **Iterator**

The enhanced for loop can only be used with those data structures that implement the “ **Iterable**” interface...

Iterator

Iterator is an interface which belongs to Java’s collection framework.

It allows us to traverse the collection and access the data element.

The iterator interface contains three methods, we will look at two of them: “**hasNext**”, and “**Next**”. The third method is called “**remove**” and is rarely used.

Iterator (cont.)

- **hasNext**: It returns true if Iterator has more elements to iterate.
- **next**: It returns the next element in the collection until the hasNext method returns false. This method throws ‘**NoSuchElementException**’ if there is no next element.

Let’s look at an example...

Iterator (cont.)

Java’s Collection implements the **iterable interface** not the **iterator interface** , so what is the difference?

The **Iterable Interface** specifies a method called: “**iterator**” that returns an Iterator object. It is this interface that allows an Enhanced For Loop to be used on a Collection.

The **Iterator Interface** specifies the: “**hasNext**” and “**next**” methods, these methods are responsible for the traversal over the data in a Collection (or data structure).

Let’s look at Java’s Documentation...