Lab 3

Iterators + Collections

Notes:

- Don't forget describing question on Homework3
- Please actively participate on *piazza*; you will receive *extra credit* for helping your fellow classmates

Iterators

- Description: *Iterator* is an object that enables us to traverse a container
- Usage:

```
// Create Linked List
List<String> list = new List<String>();

// Add Elements
list.add("Foo");
list.add("Bar");
list.add("Baz");

// Iterate through the list using ForEach Loop or enhanced for loop
for (String str : list) {
  System.out.println(str);
}
```

Iterators (Cont'd)

Do not access iterators `next()` item when `hasNext()` is `false`

```
it = coll.iterator();

while (it.hasNext()) {
  item = it.next();

  // ... and etc
}
```

Iterators (Cont'd)

- Invalid Iterators:
  - After changes to collection, either 1) **addition** or 2) **removal** can cause all iterators to become invalid.
- `Iterable<E>`
  - Implementing this interface allows an object to be the target of the "for-each loop" statement.
- `Iterator<E>`
  - Interface that describes the blueprint of an iterator object

Iterators vs Enumeration in Java

- Iterator takes the place of *Enumeration* in the Java Collections Framework.
- Iterators differ from enumerations in two ways:
  - Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
  - Method names have been improved
    - `hasMoreElements()` and `nextElement()` in Enumeration
    - `hasNext()`, `next()` and `remove()`

Basic Design:

First implement `Iterable<E>`

```
class DynamicArray<E> implements Iterable<E> {

    public Iterator<E> iterator()
    {
        return new DynamicArrayIterator<E>(this);
    }
}
```

Basic Design (Cont'd)

Then implement `Iterator<E>`

```
class DynamicArrayIterator<E> implements Iterator<E> {

    public DynamicArrayIterator(DynamicArray<E> dynamicArray) { }

    public boolean hasNext() { }

    public E next() { }

    public void remove() { }
}
```

Basic Design (Cont'd)

Then implement `Iterator<E>`

```
class DynamicArray<E> implements Iterable<E> {

    public Iterator<E> iterator()
    {
        return new DynamicArrayIterator<E>(this);
    }
}

class DynamicArrayIterator<E> implements Iterator<E> {

    // ...

}
```

Homework `Iterable`, `Iterator` design

Then implement `Iterator<E>`

```
public class MyCollection implements Iterable<E> {

    // private fields here ...
    // int version = ...

    public Iterator<E> iterator()
    {
        return new DynamicArrayIterator<E>(this);
    }

    private class MyCollectionIterator implements Iterator<E> {

        // private field and methods of outer class can be accessed:
        // :> MyCollection.this.wellFormed()
        // :> MyCollection.this.version
    }
}
```

`AbstractCollection`

Provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.

Concerning `wellFormed` for `Collection`

```
private boolean wellFormed() {

    // 0. data is not null
    // TODO

    // 1. manyItems is a valid index of data
    // TODO

    // 2. there are no null values in the array
    // TODO

    // 3. test that there are no duplicate
    // TODO

    // All good!
    return true;
}
```

Concerning `wellFormed` for Iterator nested class

```
private boolean wellFormed() {

    // 0. The outer invariant holds
    // TODO

    // 0.5 Check if iterator state is consistent with collection
    // TODO

    // 1. currentIndex is between -1 (inclusive) and manyItems (exclusive)
    // TODO

    // 2. if currentIndex is -1, there is no current element in the iterator
    // TODO

    // All good!
    return true;
}
```

`remove()` method in iterator vs. collection

What is so special?

Hint: `remove()` method in `collection` is provided by `AbstractCollection` but we need to implement `remove()` method in iterator.

Extra: Java Stream APIs

Declarative vs. Imperative

Stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

```
boolean anyNulls = Stream.of(this.data)
    .limit(manyItems)
    .anyMatch(Objects::isNull)

int countOfDistinctItems = Stream.of(this.data)
    .limit(manyItems)
    .collect(Collectors.toSet())
    .size()
```

Lab assignment #3: