Lecture 3: More ArrayIntList

- Assignment 1
    - Due next Thursday
    - Like ArrayIntList, uses arrays to store data
    - Stores a collection of letters
    - Seems kind of boring, but good review, and we will use it later in the quarter
    - Resources
        - Style/commenting guides
        - IPL
        - Message board
- A summary of our ArrayIntList so far
    - Added: a second add() method
        - First add calls the second add - less redundancy
    - Added: a set() method
        - Why isn't it enough to have add() and remove() (which could do it)?
    - More extensive exception checks
    - More commenting
        - This stuff is HARD! We are picky
    - An extra private exception check method
        - Must follow the public interface of the spec EXACTLY
        - What if you work for a company
        - Write a "quick and dirty version", then "nice version"
        - What if the first version had extra methods? People might use them
        - INCLUDES CONSTRUCTORS
- A common operation - adding all element of another structure
    - addAll(ArrayIntList other)

```
public void addAll(ArrayIntList other) {
     for (int i = 0; i < other.size(); i++)
         add(other.get(i));
 }
```

    - Start by calling add() using accessor methods
    - But it can be more efficient to access fields directly

```
public void addAll(ArrayIntList other) {
     for (int i = 0; i < other.size; i++)
         add(other.elementData[i]);
 }
```

    - Sometimes you can't solve the problem at all without field access
    - Very useful for your homework - you'll also have to do some kind of "bulk" method with another object as a parameter
- Final version: removeAll(ArrayIntList), clear()
- Now I want to switch gears, consider a new concept
    - Client code, adds values
    - Now let's write code to find the cumulative sum of the list
    - How?

```
int sum = 0;
for (int i = 0; i < list.size(); i++) {
   sum += list.get(i);
}
System.out.println("sum = " + sum);
```

    - This works, but I want to consider a different approach: ITERATOR
    - This code relies on get(), which relies on fast access into the array
        - Arrays have this fast random access

        - But some other structures we will look at this quarter don’t
        - If you knew you’d only use array-based structures, you’d be fine
    - Like DVD vs. VHS
        - Some structures can jump, some must start at the beginning
    - May seem silly now, but it will be the ONLY way to iterate through some structures
        - Some structures don’t have indices
- What is an iterator?
    - “has next”
    - “get next”
    - “move to next”
    - Java combines the last two operations in one
    - Kind of like a Scanner! But on a structure not a file
- Let’s write the skeleton of our ArrayIntListIterator
    - Will have 2 methods, next() and hasNext()
- In Java, we usually have the data structure provide the iterator through a method
    - iterator()
    - Then we can rewrite our cumulative sum:

      ```
      ArrayIntListIterator i =  list.iterator();
      int sum = 1;
      while (i.hasNext()) {
         int next = i.next();
         sum = sum + next;
      }
      System.out.println("sum = " + sum);
      ```

    - Good idea to store the i.next() in a variable
        - Avoids duplicate calls
    - Can add a println inside the loop for extra clarity

      ```
      System.out.println("sum = " + sum + ", next = " + next);
      ```

- Iterators also support a remove() method
    - Removes the last thing that we got with next()
    - Special case?
        - If next() hasn’t been called yet
        - If remove() called twice in a row
        - IllegalStateException
    - ADD TO SKELETON
    - Add code to client to remove 3’s

      ```
      if (next == 3)
              i.remove();
      ```

- How would we implement the ArrayIntList iterator?
    - What will it need to keep track of?
        - Hint: When we did the loop approach, we had a for-loop with an index
        - Position
        - Also the list itself, so we can access the list
    - Constructor
        - Parameters?
            - The list
            - (UPDATE THE ArrayIntList CODE)
        - Second use of the “this” keyword - to distinguish field from parameter
        - Where does position start?
    - Let’s look at “next”
        - We return a value - which value?
            - The one at “position”

        - How does position change?
    - How do we know if there is a next (“hasNext”)?
        - When do we reach the end of the list (same as the for-loop ending condition)
        - Compare with size
    - remove()
        - Removes the previous value
        - position - 1
        - Can call the list’s remove() method
        - Not enough - we also have to do position-- (because things shifted)
    - Robustness - add exceptions
        - Can’t remove twice in a row, can’t remove at the beginning
        - Add boolean flag
        - Throw exceptions
            - In NEXT if no HASNEXT
            - In REMOVE if not remove ok
    - “Lightweight object”
        - Doesn’t really store any data of its own
- Another issue with our code: if we run out of room!
    - Sometimes you will add enough to exceed the capacity
    - What should you do?
        - We can’t grow the array, because of how it is stored on the computer
        - Must be CONTIGUOUS - that’s how access is fast
        - Would overwrite some other objects
    - Create a new, bigger array, and copy things over
        - How much should we increase the size?
            - By 1 --> very inefficient
            - Double --> if we grow from 100 to 200, only have to copy once
            - “Amortized” - spread out over the 200 adds, the cost of growth is small
            - Actual Java ArrayList - 50%
        - Can use Arrays.copyOf()

```
public void ensureCapacity(int capacity) {
    if (capacity > elementData.length) {
        int newCapacity = elementData.length * 2 + 1;
        if (capacity > newCapacity) {
            newCapacity = capacity;
        }
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}
```

- Summary
    - private fields
    - class constants for “magic numbers”
    - initialize fields in the constructor
    - use “this()” to reduce redundancy in constructor calls
    - throw exceptions to prevent misuse of your code
    - document all preconditions (including exceptions), postconditions
    - boolean zen when dealing with boolean expressions
    - when overloading methods, have more general call more specific method
    - add private helper methods if needed
    - Can access private fields of object in methods of the same class