Lecture 5: Sets/Maps

- A problem
  - I want to count the # of unique words in a large corpus
    - Ex. Shakespeare's collected works
  - Let user search the words to see if a particular word appears in the corpus
  - Stack/queue doesn't seem particularly suited to this problem - let's use a list

```
List<String> words = new ArrayList<String>();
while (input.hasNext()) {
    String word = input.next();
    if (!words.contains(next)) {
        words.add(word);
    }
}
```

  - This implementation takes about 45 seconds to process Shakespeare
    - What is so slow?
    - The contains() operation has to look through the entire list
- Using an ArrayList isn't good - too slow!
  - We want a structure that has a faster contains() operation
  - Also, do we care about the ordering of the words?
    - So we probably don't need indices like in a list
- A new ADT: "set"
  - An UNORDERED collection of UNIQUE values
    - No indices
    - You can't control the ordering
  - Behaviors
    - add(value)
    - remove(value)
    - contains(value)
  - Implementation types
    - TreeSet
    - HashSet
    - (how they are implemented later in the quarter)
- Let's modify our code to work with a TreeSet

```
Set<String> words = new TreeSet<String>();
while (input.hasNext()) {
    String word = input.next();
    if (!words.contains(next)) {
        words.add(word);
    }
}
```

- In fact, you don't even need the inner if-statement because sets don't allow duplicates!
  - If you try to add something twice, nothing happens the second time
- Now let's try a HashSet
  - HashSet is faster

- Improvements? IGNORE CASE, PUNCTUATION

- I also want to examine the words to see what words are in Shakespeare - print them out!
    - How would you do this with a list?

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

    - But a set doesn't have indices - you can't do a get
    - How else can you examine the things in a collection
        - Iterator!
        - Sets also have an iterator

```
Iterator<String> i = words.iterator();
while (i.hasNext()) {
    System.out.println(i.next());
}
```

- What do you notice about the words when we use a TreeSet?
    - They are in alphabetical order!
    - But when we change to a HashSet, they are in random order
- Tradeoff between TreeSet and HashSet
    - TreeSet - sorted order, but slower
    - HashSet - unknown order, but faster
    - Which you use depends on what you want to do
- Iterators can be kind of ugly
    - This extra object hanging around - Iterator<E>
    - Java has a shorthand for using an iterator so you never actually see the iterator
    - FOR-EACH LOOP

```
for (String word : words) {
    System.out.println(word);
}
```

    - "for each String word that is in words..."
        - The choice of "word" is arbitrary - can be any variable name (e.g. "s")
        - Every time through the loop, Java sets the "word" variable to the next thing in the list
        - (basically calls iterator.next behind the hood)
        - Much simpler to use
    - FOR-EACH also works on ArrayLists and arrays
        - But should only use if you don't care about the index
        - (translate FOREACHEXAMPLE to for-each loops)
    - If you CAN use a for-each, you SHOULD use a for-each
- Restrictions of for-each
    - You cannot change the collection with a for-each
    - Re-assigning the variable doesn't change the list
        - (show example by changing each string in Shakespeare that starts with s to "HAX0R")
    - If you want to remove, use an iterator
        - (show an example by deleting strings that start with s)

- A related problem: Count the number of occurrences of each word
    - How could we do this with the structures that we've learned so far?
        - Parallel arrays or ArrayLists - index matches in both the word and counts lists
            - (not very object-oriented - word and count logically belong together)
        - Create a list of small objects (each object containing the word and the count)
            - But we saw lists are slow
        - Can't use a set because we can't guarantee the ordering would match for counts and words
    - Java provides us with an alternative: the MAP
- Another ADT: Map<E>
    - A Map stores a collection of key-value pairs
        - We want word/count pairs - what is the count for each word?
        - keys=words, values=counts
    - Real life examples
        - We use SSN to identify information with a person
        - The registrar uses your UW student # to get information about you
    - Only one value per key
        - For example, if the registrar looks up your student # and finds 3 different students, something would be wrong
        - The key UNIQUELY identifies the value
        - The keys form a SET
        - If you try to add the same key to the map twice, you will overwrite the old value
    - Look at the interface
        - Note - instead of ONE type, you have to give it TWO types
        - New operations - you can look at the KEYS, and you can look at the VALUES
        - Note that Collection<E> encompasses all the structures we've been looking at - List, Set, Stack, Queue
        - IMPLEMENTATION types:
            - TreeMap
            - HashMap
            - Same differences as before - TreeMap keys in sorted order
- Using a map to count occurrences
    - What are the key and value types that we want to use?
        - Keys = words --> String
        - Values = counts --> Integer
    - We'll start with a TreeMap
    - What do we want to do the first time we see a word?

```
Map<String, Integer> counts = new TreeMap<String, Integer>();
while (input.hasNext()) {
    String word = input.next().toLowerCase();
    counts.put(word, 1);
}
```

    - This is the right start, but not quite - it will keep track of the unique words, but all will have a

count of 1.

- What do we want the count to be if a word HAS been seen before?
```
counts.get(word) + 1
```
- How do we know which version to use?
  - An if-else, using the containsKey method to see if we've seen the word before

```
Map<String, Integer> counts = new TreeMap<String, Integer>();
while (input.hasNext()) {
    String word = input.next().toLowerCase();
    if (!counts.containsKey(word)) {
        counts.put(word, 1);
    } else {
        counts.put(word, counts.get(word) + 1);
    }
}
```

- Each time we call put subsequent to the 1st time, the count in the map goes up
  - Wipe out the old association in the map with the new "put"
  - This is a very common way of doing things

- Now we want to print the results
  - But only words that appear more than 1000 times
  - How do you iterate over a map? There are 2 parts to each key/value pair
  - We usually use a for-each loop over the keySet of the map
  - Then call get to get the value associated with the key

```
for (String word : counts.keySet()) {
    int count = counts.get(word);
    if (count > 1000) {
        System.out.println(count + "\t" + word);
    }
}
```

- Another example: find the words that appear in both Shakespeare AND Moby Dick

```
Set<String> common = new TreeSet<String>();
for (String word1 : words1) {
    if words2.contains(word1) {
        common.add(word1);
    }
}
```

- Another example: reverse the map - find all words that have the same # of occurrences
  - What is the type of the new map? Map<Integer, Set<String>>

```
Map<Integer, Set<String>> reversed =
        TreeMap<Integer, Set<String>>();
for (String word : counts.keySet()) {
    int count = counts.get(word);
    if (!reversed.containsKey(count)) {
        reversed.put(count, new TreeSet<String>());
    }
    reversed.get(count).add(word);
}
```