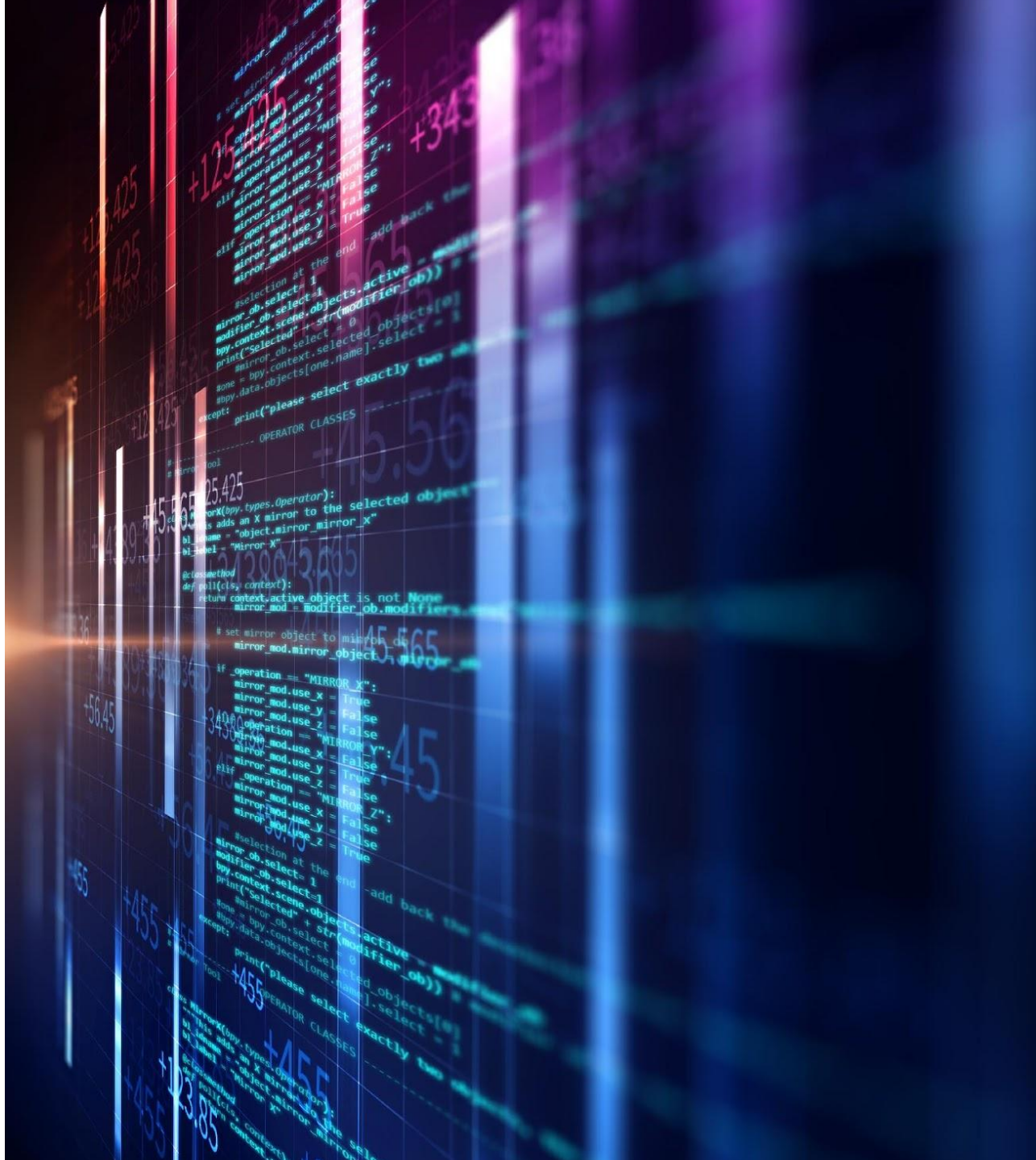
Collections & Maps

Brandon Krakowsky

Penn Engineering



Collections & Maps

Collections

- A Collection is a structured group of objects
- Java includes a *Collections Framework* which is a unified architecture for representing and manipulating different kinds of collections
  - Collections are defined in java.util
  - The *Collections Framework* is designed around a set of standard interfaces
  - There are a number of predefined implementations (i.e. classes)
- An ArrayList is one type (or implementation) of Collection
  - To be more precise, ArrayList is an implementation of List, which is a *subinterface* of Collection
- For reference, an Array is NOT a type (or implementation) of any of the Collection interfaces

Ref: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Types of Collections & “Collection-Like” Things (Maps)

- Java supplies several types of Collections
  - Here are some:
    - Set: Cannot contain duplicate elements, order is not important
    - SortedSet: Like a Set, but order is important
    - List: May contain duplicate elements, order is important
- Java also supplies some “collection-like” things (i.e. Maps)
  - Here are some:
    - Map: A “dictionary” that associates keys with values, where order is not important
    - SortedMap: Like a Map, where order is important

Ref: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html
https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Hierarchy of Collections

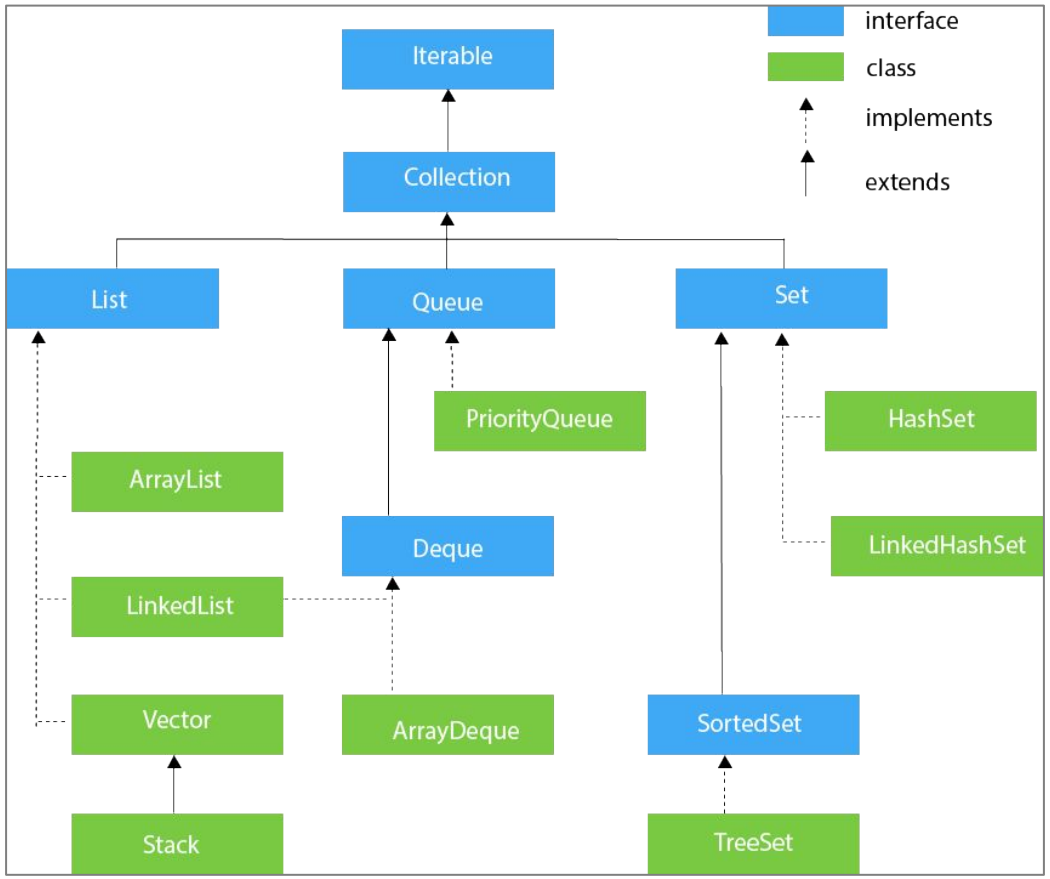


Image: https://www.javatpoint.com/collections-in-java

Hierarchy of “Collection-Like” Things (Maps)

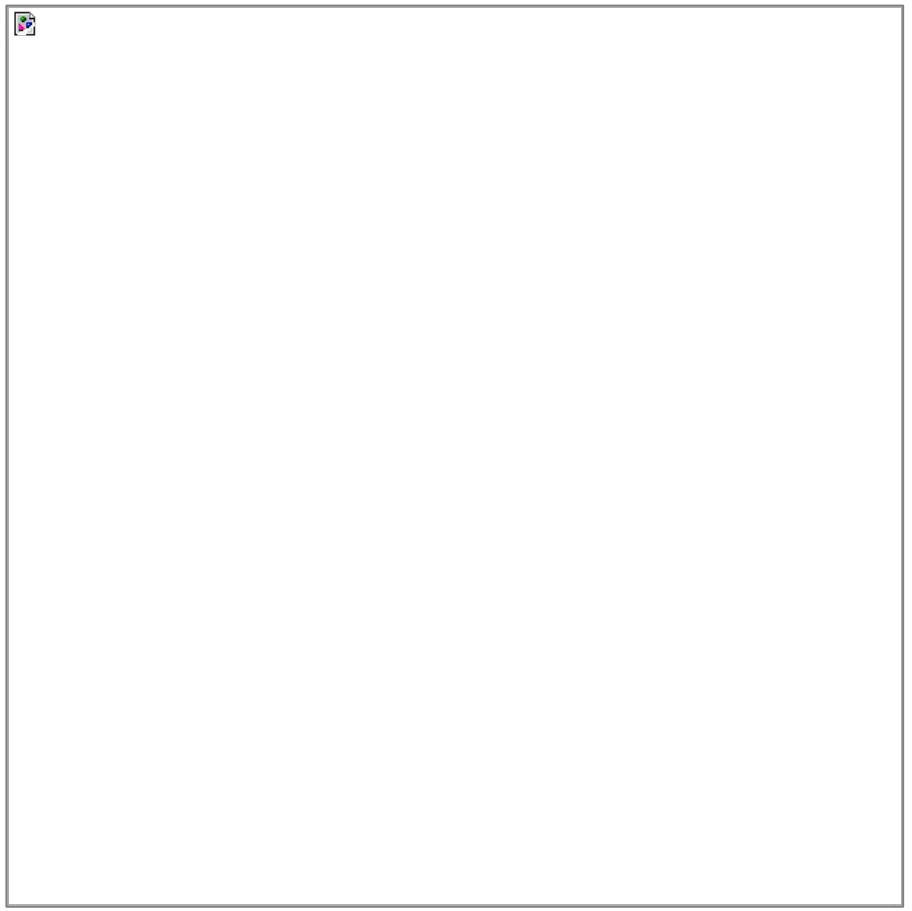


Image: https://www.javatpoint.com/java-map

Methods in the Collection Interface

- The Collection interface is the root interface of the Collection hierarchy
- All *subinterfaces* include the methods in the Collection interface
- Here are some, but not all of the methods:

```
boolean add(E o)
boolean contains(Object o)
boolean remove(Object o)
boolean isEmpty()
int size()
Object[] toArray()
Iterator<E> iterator()
```

Implementations

Each interface has at least one implementation. Here are some:

- List – ArrayList, LinkedList
- Deque (Double-ended queue) – ArrayDeque, ConcurrentLinkedDeque, LinkedList
- Set – HashSet, LinkedHashSet, TreeSet
- Map – HashMap, TreeMap

List Interface

- Elements are stored in the order they are added
- Access to elements is through its index position in the list
- Some additional specialized methods of the List interface:

```
void add(int index, E element)
E get(int index)
void set(int index, E element)
ListIterator<E> listIterator()
```

Deque Interface

- An ordered sequence like a List
- Element access only at the front or the rear of the Collection
- Can be used as both a stack (LIFO) and a queue (FIFO)
- Some additional specialized methods of the Deque interface:

```
void addFirst(E o)
void addLast(E o)
E getFirst()
E getLast()
E removeFirst()
E removeLast()
```

Set Interface

- Models the mathematical set
- Does not allow duplicates
- No additional methods other than those in the Collection interface (*add*, *remove*, *size*, etc.)

Map Interface

- Not part of the Collection interface hierarchy – it does not inherit Collection methods
- Maps keys to values
- Also known as a dictionary or associative array
- Cannot contain duplicate keys
- Each key can map to at most one value

Important methods:

```
V put(K key, V value)
V get(Object key)
```

General Rules for Selecting an Implementation

- List – If you need fast access to random elements in the list, choose the ArrayList. If you will frequently remove or insert elements to the list, choose a LinkedList.
- Deque – If you only need access at the ends (beginning or end) of the sequence. Use an ArrayDeque if you don't need a thread safe implementation. Otherwise, choose a ConcurrentLinkedQueue.
- Set – Use a TreeSet if you need to traverse the set in sorted order. Otherwise, use a HashSet, it's more efficient.
- Map – Choose TreeMap if you want to access the collection in key order. Otherwise, choose HashMap.

Iterator

All Collection implementations have an *Iterator* object which can be used to loop through the collection

- Use an *iterator* instead of a for loop to modify the collection while traversing

```
//get iterator object from treeset (ordered set)
Iterator<String> it = treeSet.iterator();

//modify (remove) the values using the iterator while traversing the
treeset
while(it.hasNext()) {
    if(it.next().equals(“red”)) {
        it.remove();
    }
}
```

ConcurrentModificationException

- A *ConcurrentModificationException* may be thrown when a collection is modified while traversing, by any means other than through its *iterator*

THIS IS NOT OK

```
//modifying while traversing without
//iterator
for(String s : treeSet) {
    if(s.equals(“red”)) {
        //throws Exception
        treeSet.remove(“red”);
    }
}
```

THIS IS OK

```
//modifying while traversing with
//iterator
Iterator<String> it = treeSet.iterator();
while(it.hasNext()) {
    if(it.next().equals(“red”)) {
        it.remove();
    }
}
```

Comparable Interface

Some Collection implementations are stored in sorted order (e.g. TreeSet)

- To store an object in a sorted Collection, the Object must have a natural order OR it must implement the *Comparable* interface
- For example, here's a TreeSet containing Integers

```
TreeSet<Integer> ages  = new TreeSet<Integer>();

ages.add(2);
ages.add(2);
ages.add(1);

System.out.println(ages);
```

- This will "naturally" sort the numeric values and print the following

```
[1, 2]
```

Comparable Interface

Some Collection implementations are stored in sorted order (e.g. TreeSet)

- To store an object in a sorted Collection, the Object must have a natural order OR it must implement the *Comparable* interface
- The *Comparable* interface has one method *compareTo* that must be implemented
  - Strings implement *compareTo*, so Collections like TreeSets know how to sort them
- Here's a TreeSet containing Strings

```
TreeSet<String> names  = new TreeSet<String>();

names.add("Ted");
names.add("Ted");
names.add("Ryan");

System.out.println(names);
```

- This will sort the String values alphabetically and print the following

```
[Ryan, Ted]
```

Comparable Interface

- Here's an example Author class that implements Comparable and *compareTo*

```
public class Author implements Comparable<Author> {

    String firstName;
    String lastName;

    @Override
    public int compareTo(Author other) {
        //compare the names of authors
        //returns negative number if this is supposed to be less than the other
        //returns positive number if this is supposed to be greater than the other
        //otherwise returns 0 if they are supposed to be equal
        int last = this.lastName.compareTo(other.lastName);
        return last == 0 ? this.firstName.compareTo(other.firstName) : last;
    }
}
```

Comparable Interface

- Now this TreeSet containing Authors knows how to sort them

```
TreeSet<Author> authors  = new TreeSet<Author>();

Author author1 = new Author();
author1.firstName = “William”;
author1.lastName = “Shakespeare”;

Author author2 = new Author();
author2.firstName = “Brandon”;
author2.lastName = “Krakowsky”;

authors.add(author1);
authors.add(author2);
```

- authors will automatically sort author2 before author1

Sorting & Searching Collections

The Collections class has some convenient *static* methods for working with collections

- These include sorting and search methods that use optimized algorithms
- Many of the methods require the objects in the collection to implement the Comparable interface
- Some examples:

  `Collections.sort(arrayList)` – sorts list using *merge sort* algorithm

  `int position = Collections.binarySearch(arrayList, “red”)` – returns position in list where object is found

Arrays Class

For reference, the Arrays class also has *static* methods for working with arrays

- For example:

  `List list = Arrays.asList(array)` – returns List from array

  `Arrays.sort(array)` – sorts array using *quick sort* algorithm

Exercises with Collections

CollectionsClass

CollectionsClass.java

```
 1 import java.util.ArrayList;
13
14 /**
15  * Class with various methods for using different kinds of Collections.
16  */
17 public class CollectionsClass {
18
```

Remove from a List

```java
/**
 * Takes an ArrayList of integers and two integer values min and max
 * as parameters and removes all elements with values in the range min through
 * max (inclusive).
 *
 * For example, if an ArrayList named 'list' stores
 * [7, 9, 4, 2, 7, 7, 5, 3, 5, 1, 7, 8, 6, 7], the call of
 * removeRange(list, 5, 7) should change the list to [9, 4, 2, 3, 1, 8].
 *
 * Uses Iterator.
 *
 * @param list of values
 * @param min of range
 * @param max of range
 */
public static void removeRange(ArrayList<Integer> list, int min, int max) {

    //Create iterator and use it to remove items in place
    //Avoid ConcurrentModificationException
    Iterator<Integer> iterator = list.iterator();
    while (iterator.hasNext()) {
        Integer next = iterator.next();
        if (next >= min && next <= max) {
            iterator.remove();
        }
    }
}
```

Remove from a List

```
310
311     public static void main(String[] args) {
312
313         //removeRange
314         //create array of Integers
315         Integer[] removeRangeArray = {7, 9, 4, 2, 7, 7, 5, 3, 5, 1, 7, 8, 6, 7};
316         ArrayList<Integer> list = new ArrayList<Integer>();
317
318         //add all items from Integer array to arraylist
319         list.addAll(Arrays.asList(removeRangeArray));
320         CollectionsClass.removeRange(list, 5, 7);
321
322         //expected output [9, 4, 2, 3, 1, 8]
323         System.out.println("removeRange: " + list);
324         System.out.println();
325
```

Add to a List

```
47      /**
48       * Takes an ArrayList of strings as a parameter and modifies the list
49       * by placing a "*" in between each element, and at the start
50       * and end of the list.
51       *
52       * For example, if a list named 'list' contains
53       * ["the", "quick", "brown", "fox"],
54       * the call of addStars(list) should modify it to store
55       * ["*", "the", "*", "quick", "*", "brown", "*", "fox", "*"].
56       *
57       * @param list of values to add stars
58       */
59      public static void addStars(ArrayList<String> list) {
60
61          //copy all values in arraylist to array
62          //Note: toArray takes an empty array into which the values are to be stored
63          String[] array = list.toArray(new String[list.size()]);
64
65          //empty original arraylist
66          list.removeAll(Arrays.asList(array));
67
68          //add stars and values back into the original arraylist
69          list.add("*");
70          for (String s : array) {
71              list.add(s);
72              list.add("*");
73          }
74      }
75
```

Add to a List

```
199
200         //addStars
201         //create array of Strings
202         String[] addStar = {"the", "quick", "brown", "fox"};
203
204         //add all items from String array to arraylist
205         ArrayList<String> sList = new ArrayList<String>();
206         sList.addAll(Arrays.asList(addStar));
207
208         CollectionsClass.addStars(sList);
209
210         //expected output ["*", "the", "*", "quick", "*", "brown", "*", "fox", "*"]
211         System.out.println("addStars: " + sList);
212         System.out.println();
213
```

Count Words

```
63      /**
64       * The classic word-count algorithm: given an array of strings,
65       * return a Map<String, Integer> with a key for each different string,
66       * with the value the number of times that string appears in the array.
67       *
68       * wordCount(["a", "b", "a", "c", "b"])  {"a": 2, "b": 2, "c": 1}
69       * wordCount(["c", "b", "a"])  {"a": 1, "b": 1, "c": 1}
70       * wordCount(["c", "c", "c", "c"]) {"c": 4}
71       *
72       * Uses HashMap
73       *
74       * @param strings to count
75       * @return map of word counts, where key is word and value is count
76       */
77      public static Map<String, Integer> wordCount(String[] strings) {
78
79          //create a hashmap (has no order)
80          Map<String, Integer> map = new HashMap<String, Integer>();
81
```

Count Words

```
82          //iterate over given array
83          for (String s : strings) {
84
85              //if map does not contain string as a key
86              if (!map.containsKey(s)) {
87
88                  //add key with default value 1
89                  map.put(s, 1);
90              } else {
91
92                  //replace the old count with incremented count
93                  map.replace(s, map.get(s) + 1);
94              }
95          }
96
97          return map;
98      }
```

Count Words

```
339         //wordCount
340         String[] s = {"a", "b", "a", "c", "b"};
341         Map<String, Integer> ret = CollectionsClass.wordCount(s);
342
343         //expected: {a=2, b=2, c=1}
344         System.out.println("wordCount: " + ret);
345         System.out.println();
```

Count Unique Words

```
110     /**
111      * Takes an array of Strings as a parameter and returns a count of the
112      * number of unique words in the array.
113      *
114      * DOES consider capitalization and/or punctuation; for example,
115      * "Hello", "hello", and "hello!!" are considered different words.
116      *
117      * Uses HashSet.
118      *
119      * @param words to count
120      * @return count of unique words
121      */
122     public static int countUniqueWords(String[] words) {
123
124         //create hashset (has no order)
125         Set<String> hashSetWords = new HashSet<String>(Arrays.asList(words));
126
127         return hashSetWords.size();
128     }
```

Count Unique Words

```
347         //countUniqueWords
348         String[] countUniqueWordsArray = {"hello", "izzy", "and", "Elise", "Hello"};
349
350         //expected: 5
351         System.out.println("countUniqueWords: "
352                 + CollectionsClass.countUniqueWords(countUniqueWordsArray));
353         System.out.println();
354
```

Count Unique Words (Case Insensitive)

```
129
130     /**
131      * Takes an array of Strings as a parameter and returns a count
132      * of the number of unique words in the array.
133      *
134      * DOES NOT consider capitalization; for example,
135      * "Hello" and "hello" should NOT BE considered different words for this problem.
136      *
137      * Uses TreeSet.
138      *
139      * @param words to count
140      * @return count of unique words
141      */
142     public static int countUniqueWordsCaseInsensitive(String[] words) {
143
144         //create treeset (like hashset, but ordered)
145         //Note: String.CASE_INSENSITIVE_ORDER makes internal comparison use equalsIgnoreCase
146         Set<String> treeSetWords = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
147         treeSetWords.addAll(Arrays.asList(words));
148
149         return treeSetWords.size();
150     }
```

Count Unique Words (Case Insensitive)

```
354         //countUniqueWordsCaseInsensitive
355         //expected: 4
356         System.out.println("countUniqueWordsCaseInsensitive: "
357                 + CollectionsClass.countUniqueWordsCaseInsensitive(countUniqueWordsArray));
358         System.out.println();
359
```

Remove Duplicates

```
151
152     /**
153      * Takes as a parameter an ArrayList of integers, and modifies it
154      * by removing any duplicates.
155      *
156      * Note that the elements of the list are not in any particular order, so the
157      * duplicates might not occur consecutively. This method retains the original
158      * relative order of the elements.
159      *
160      * For a given list [4, 0, 2, 9, 4, 7, 2, 0, 0, 9, 6, 6],
161      * the call of removeDuplicates(list) should modify it to store [4, 0, 2, 9, 7, 6].
162      *
163      * Use LinkedHashSet.
164      *
165      * @param list of ints to remove duplicates from
166      */
167     public static void removeDuplicates(ArrayList<Integer> list) {
168
169         //create LinkedHashSet (like hashset, but maintains insertion order)
170         Set<Integer> linkedHashSet = new LinkedHashSet<Integer>(list);
171
172         list.removeAll(list);
173         list.addAll(linkedHashSet);
174     }
175
```

Remove Duplicates

```
356         //removeDuplicates
357         Integer[] removeDuplicatesArray = {4, 0, 2, 9, 4, 7, 2, 0, 0, 9, 6, 6};
358         ArrayList<Integer> list2 = new ArrayList<Integer>();
359         list2.addAll(Arrays.asList(removeDuplicatesArray));
360         CollectionsClass.removeDuplicates(list2);
361
362         //expected: {4, 0, 2, 9, 7, 6}
363         System.out.println("removeDuplicates: " + list2);
364         System.out.println();
365
```

Set Toppings

```java
    /**
     * Takes a map of food keys and topping values, and modifies and returns the map as follows:
     * If the key "ice cream" is present, set its value to "cherry".
     * In all cases, set the key "bread" to have the value of "butter".
     *
     * setToppings({"ice cream": "peanuts"}) {"bread": "butter", "ice cream": "cherry"}
     * setToppings({}) {"bread": "butter"}
     * setToppings({"pancake": "syrup"}) {"bread": "butter", "pancake": "syrup"}
     *
     * @param map of food items and toppings
     * @return updated map of food items and toppings
     */
    public static Map<String, String> setToppings(Map<String, String> map) {

        //add key (bread) and value (butter) if it's not in map
        if (!map.containsKey("bread")) {
            map.put("bread", "butter");
        }

        //if key is 'ice cream', set value to 'cherry'
        if (map.containsKey("ice cream")) {
            map.replace("ice cream", "cherry");
        }

        return map;
    }
```

Set Toppings

```
370         //setToppings
371         //create hashmap with food items
372         Map<String, String> food = new HashMap<String, String>();
373         food.put("ice cream", "peanuts");
374
375         Map<String, String> m = CollectionsClass.setToppings(food);
376         //expected: {bread=butter, ice cream=cherry}
377         System.out.println("setToppings: " + m);
378         System.out.println();
379
```

Friend List

```
216
217     /**
218      * Takes a Map<String, String> as a parameter and
219      * reads friend relationships and stores them into a compound
220      * collection that is returned.
221      *
222      * Creates a new map where each key is a person's name from the original Map,
223      * and the value associated with that key is a set of all friends of that person.
224      *
225      * Friendships are bi-directional:
226      * If Marty is friends with Danielle, then Danielle is friends with Marty
227      *
228      * The Map parameter contains one friend relationship per key/value pair,
229      * consisting of two names.  For example, if the map parameter friendMap
230      * looks like this: {Marty: Cynthia, Danielle: Marty}
231      *
232      * Then the call of friendList(friendMap) should return a map with the following
233      * contents: {Cynthia:[Marty], Danielle:[Marty], Marty:[Cynthia, Danielle]}
234      *
235      * Uses a TreeMap of TreeSets.
236      *
237      * @param friendMap of friendships
238      * @return map where each key is a person's name and the value is the set of all friends
239      */
240     public static TreeMap<String, TreeSet<String>> friendList(Map<String, String> friendMap) {
241
242         //create a treemap of treesets (like hashsets, but ordered)
243         TreeMap<String, TreeSet<String>> treeMap = new TreeMap<String, TreeSet<String>>();
244
```

Friend List

```
245         //iterate over entrySet (key/value pairs) for friendship map
246         for (Entry<String, String> friendShip : friendMap.entrySet()) {
247 
248             //get key
249             String friend1 = friendShip.getKey();
250 
251             //get value
252             String friend2 = friendShip.getValue();
253 
254             //if the tree map doesn't contain friend1
255             if (!treeMap.containsKey(friend1)) {
256                 //put friend1 with empty treeset value
257                 treeMap.put(friend1, new TreeSet<String>());
258             }
259 
260             //then get friend1 and add friend2 to treeset
261             treeMap.get(friend1).add(friend2);
262 
263             //check if friend2 is in tree map
264             if (!treeMap.containsKey(friend2)) {
265                 //if not, put friend2 with empty treeset value
266                 treeMap.put(friend2, new TreeSet<String>());
267             }
268 
269             //then get friend2 and add friend1 to treeset
270             treeMap.get(friend2).add(friend1);
271 
272         }
273 
274         return treeMap;
275     }
```

Friend List

```
376         //friendList
377         //create hashmap of friendships
378         HashMap<String, String> friends = new HashMap<String, String>();
379         friends.put("Marty", "Cynthia");
380         friends.put("Danielle", "Marty");
381
382         TreeMap<String, TreeSet<String>> friendList = CollectionsClass.friendList(friends);
383
384         //expected: {Cynthia=[Marty], Danielle=[Marty], Marty=[Cynthia, Danielle]}
385         System.out.println("friendList: " + friendList);
386         System.out.println();
387
```

Union Sets

```
178     /**
179      * Takes as a parameter, a HashSet of TreeSets of integers,
180      * and returns a TreeSet of integers representing the union of all of the sets of
181      * ints.  A union is the combination of everything (without duplicates) in each set.
182      *
183      * For example, calling the method on the following set of sets:
184      * {{1, 3}, {2, 3, 4, 5}, {3, 5, 6, 7}, {42}}
185      * Should cause the following set of integers to be returned:
186      * {1, 2, 3, 4, 5, 6, 7, 42}
187      *
188      * Uses HashSet<TreeSet<Integer>>.
189      *
190      * @param sets to union
191      * @return union of all sets
192      */
193     public static TreeSet<Integer> unionSets(HashSet<TreeSet<Integer>> sets) {
194
195         //create treeset to store union of individual treesets
196         TreeSet<Integer> treeSet = new TreeSet<Integer>();
197
198         //iterate over hashset of treesets
199         for (TreeSet<Integer> ts : sets) {
200             //add each int from treeset
201             treeSet.addAll(ts);
202         }
203
204         return treeSet;
205     }
206
```

Union Sets

```
388         //unionSets
389
390         //create hashset of treesets (like hashsets but ordered)
391         HashSet<TreeSet<Integer>> hashSet = new HashSet<TreeSet<Integer>>();
392
393         //create individual treesets with int values and add to hashset
394         TreeSet<Integer> ts = new TreeSet<Integer>();
395         Integer[] arr1 = {1, 3};
396         ts.addAll(Arrays.asList(arr1));
397         hashSet.add(ts);
398
399         ts = new TreeSet<Integer>();
400         Integer[] arr2 = {5, 4, 3, 2};
401         ts.addAll(Arrays.asList(arr2));
402         hashSet.add(ts);
403
404         ts = new TreeSet<Integer>();
405         Integer[] arr3 = {3, 5, 6, 7};
406         ts.addAll(Arrays.asList(arr3));
407         hashSet.add(ts);
408
409         ts = new TreeSet<Integer>();
410         ts.add(42);
411         hashSet.add(ts);
412
413         TreeSet<Integer> treeSet = CollectionsClass.unionSets(hashSet);
414
415         //expected: {1, 2, 3, 4, 5, 6, 7, 42}
416         System.out.println("unionSets: " + treeSet);
```