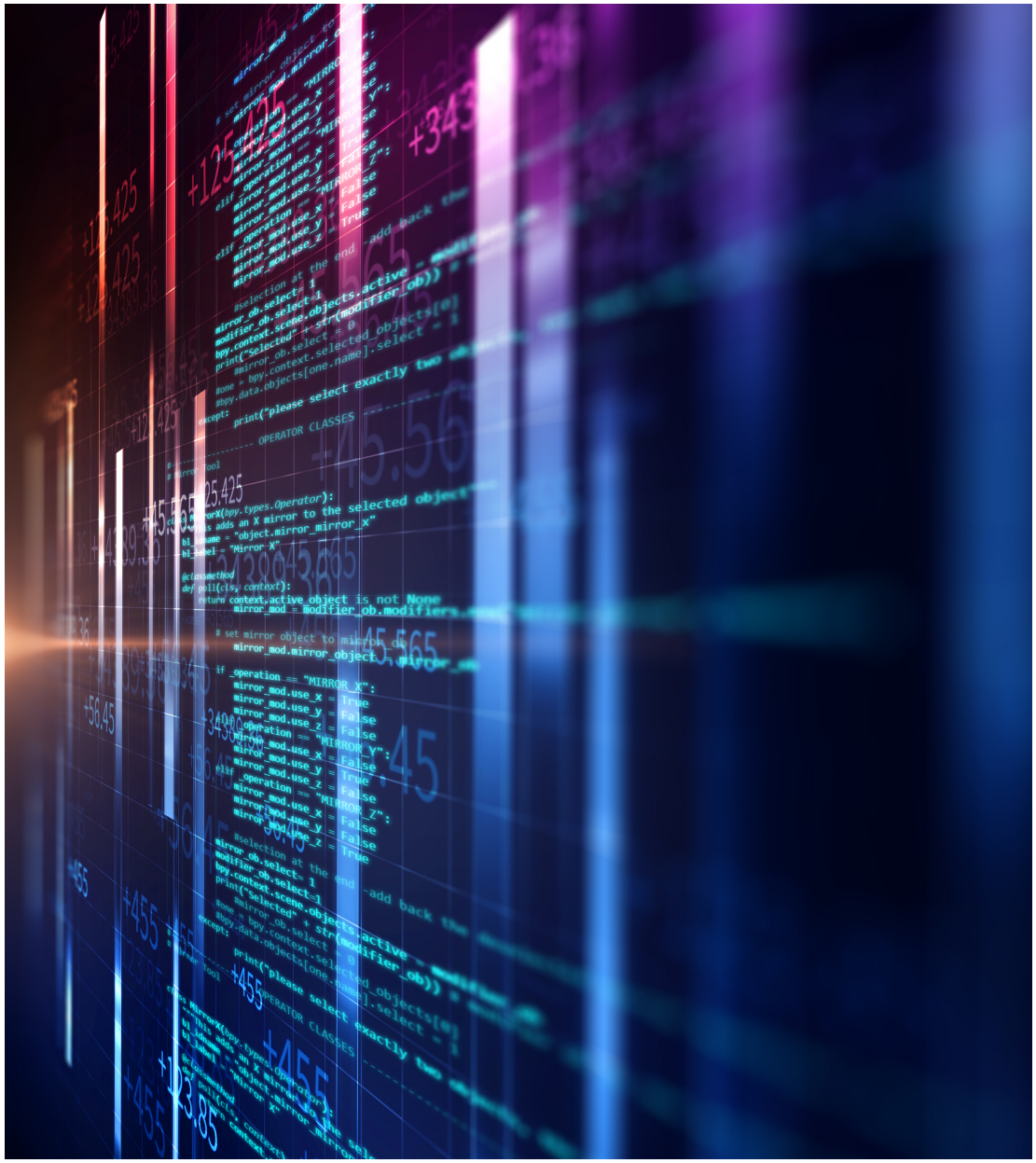
Collections & Maps

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

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* is 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();
    }
}
```

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
```

Set Toppings

```
149
150     /**
151      * Takes a map of food keys and topping values, and modifies and returns the map as follows:
152      * If the key "ice cream" is present, set its value to "cherry".
153      * In all cases, set the key "bread" to have the value of "butter".
154      *
155      * setToppings({"ice cream": "peanuts"}) {"bread": "butter", "ice cream": "cherry"}
156      * setToppings({}) {"bread": "butter"}
157      * setToppings({"pancake": "syrup"}) {"bread": "butter", "pancake": "syrup"}
158      *
159      * @param map of food items and toppings
160      * @return updated map of food items and toppings
161      */
162     public static Map<String, String> setToppings(Map<String, String> map) {
163
164         //add key (bread) and value (butter) if it's not in map
165         if (!map.containsKey("bread")) {
166             map.put("bread", "butter");
167         }
168
169         //if key is 'ice cream', set value to 'cherry'
170         if (map.containsKey("ice cream")) {
171             map.replace("ice cream", "cherry");
172         }
173
174         return map;
175     }
176
```

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
```