Recitation 6: Enums and Collections

Recitation TA Names Here

Old-fashioned, error prone

```java
public class PlayingCard {
    // i Hearts, 2 Spades, 3 Clubs, 4 Diamonds
    private int suit;
    // i Ace, 2, ..., 10, 11 Jack, 12 Queen, 13 King
    private int value;
    // ...
}
```

Don’t program like this! Frought with danger. Have to use integers, e.g.
if (c.suit == 1) // ...
User may forget what 1 means and make a mistake.

Better, but still problematic

```java
public class PlayingCard {
    public static final int Hearts = 1;
    public static final int Spades = 2;
    // ...
    private int suit;
    private int value;
    // ...
}
```

Well, still relying on integers, and user isn’t forced to use names, can still use integers. (Professionals won’t, beginners will)

Declare an enum, in a new file Suit.java:

```java
public enum Suit {SPADES, CLUBS, DIAMONDS, HEARTS};
```

- New enum keyword
- Can use any access modifier
- Enumerate over all possible values
- A enum is a subclass of java.lang.Enum

```java
public class Card { Suit suit; ...
}
```

Then, user writes:
```java
if (c.suit == Suit.SPADES)
```

Enums: Tidbits

An enum’s constructor is private

The ONLY objects of class Suit that can be created are:
- Suit.SPADES, Suit.CLUBS, Suit.DIAMONDS, and Suit.HEARTS.

```java
public enum Suit {SPADES, CLUBS, DIAMONDS, HEARTS};
```

Enums: Tidbits

- Suit.values() returns a Suit[] of the possible constants
- .ordinal() returns the position in the list of constants (i.e. the order declared)
- Implement Comparable using the declaration order
- .toString() returns the name of the constant
**Enums: Switch Statement**

```java
Suit s = Suit.SPADES;
switch(s) {
    case SPADES:
        case CLUBS:
            color = "black";
            break;
    case HEARTS:
    case DIAMONDS:
        color = "red";
        break;
}
```

Cases fall through until reach a break statement!

**Collections: Overview**

- Different implementations to do (generally) the same thing
  - Store data about a group of information
  - Each has benefits and drawbacks for each use case

<table>
<thead>
<tr>
<th>Lists (ArrayList, LinkedList, ...)</th>
<th>Stacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets (and sorted sets)</td>
<td>Queues</td>
</tr>
<tr>
<td>Bags (multi-set: sets with repeated values)</td>
<td>Maps (and sorted maps) (like dictionaries)</td>
</tr>
</tbody>
</table>

**Collections: ArrayLists**

- Indexed: identify each element by a number
  - 0..list.size() - 1
- Ordered (due to indexing)
- Dynamic Memory Allocation
  - An ArrayList doubles in size if it gets too big

<table>
<thead>
<tr>
<th>Useful Methods to Know</th>
<th>.add(element)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.get(index)</td>
<td>.contains(element)</td>
</tr>
<tr>
<td>.remove(index)</td>
<td>.size()</td>
</tr>
</tbody>
</table>

**Aside: ArrayLists vs. Arrays**

- Both are indexed and ordered
- Syntax differences:
  - list.get(2) vs. arr[2] when getting an element
  - list.add(element) vs. arr[<index>] = element for adding an element
- Dynamic Memory Allocation: arrays have fixed amount of space
- Know the max number of elements in the list? Use an array.
- Otherwise, use an ArrayList

**Collections: HashSets**

- Unordered and unindexed
- No duplicate elements
  - Adding duplicates to a set does nothing
- Very fast for adding, removing, and contains operations!

<table>
<thead>
<tr>
<th>Useful Methods to Know</th>
<th>.add(element)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.contains(element)</td>
<td>.remove(element)</td>
</tr>
<tr>
<td>.size()</td>
<td>.isEmpty()</td>
</tr>
</tbody>
</table>

You will learn all about hash sets later in the course! For now, just use HashSet as a nice implementation of a set.
**Collections: LinkedLists**
- Ordered, but not quite indexed like an ArrayList
- Start at the head or tail and traverse through the List
- You implement this in A3!

<table>
<thead>
<tr>
<th>Useful Methods to Know</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>.add(element)</td>
<td></td>
</tr>
<tr>
<td>.get(index)</td>
<td>.remove()</td>
</tr>
<tr>
<td>.size()</td>
<td>.prepend(element)</td>
</tr>
</tbody>
</table>

**Collections: Stacks**
- Ordered, but not indexed
- Last in, first out ordering (LIFO)
- Add to the top, remove from the top

<table>
<thead>
<tr>
<th>Useful Methods to Know</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>.push(element)</td>
<td></td>
</tr>
<tr>
<td>.pop()</td>
<td>.empty()</td>
</tr>
<tr>
<td>.peek()</td>
<td></td>
</tr>
</tbody>
</table>

**Collections: Queues**
- Ordered, but not indexed
- First in, first out ordering (FIFO)
- Add to the top, remove from the bottom

<table>
<thead>
<tr>
<th>Useful Methods to Know</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>.add(element)</td>
<td></td>
</tr>
<tr>
<td>.poll()</td>
<td>.isEmpty()</td>
</tr>
<tr>
<td>.peek()</td>
<td></td>
</tr>
</tbody>
</table>

**Collections: HashMap**
- Indexed by keys, ordering depends on implementation
- Key-value pairs (in dictionary: word-meaning pairs)
- Like a dictionary in Python

<table>
<thead>
<tr>
<th>Useful Methods to Know</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>.put(key, value)</td>
<td></td>
</tr>
<tr>
<td>.get(key)</td>
<td>.containsKey(key)</td>
</tr>
<tr>
<td>.keySet()</td>
<td>.remove(key)</td>
</tr>
</tbody>
</table>

**Important Interfaces & Classes**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection&lt;E&gt;</td>
<td>add(E), contains(Object), isEmpty(), remove(Object), size(), ...</td>
</tr>
<tr>
<td>List&lt;E&gt;</td>
<td>add(int, E), get(int), indexOf(E), add(int, E), ...</td>
</tr>
<tr>
<td>Set&lt;E&gt;</td>
<td>add(E), contains(Object), isEmpty(), remove(Object), size()</td>
</tr>
<tr>
<td>Map&lt;K, V&gt;</td>
<td>put(K, V), get(Object), ...</td>
</tr>
</tbody>
</table>

No new methods in Set<E>, just changes specifications.
Iterating Without Indices: For-each Loop

```java
HashSet<E> set = new HashSet<E>();
// .. store values in the set ..

// for (EleType varName : Collection) { ... }
for (E element : set) {
    // process each element
    System.out.println(element);
}
```

Collection Problems & Practice

1. Remove duplicates from an array
2. Find all negative numbers in an array
3. Create a random note
4. Implement a Stack with a max API
5. Braces parsing

---

Remove Duplicates

```java
/**
 * [removeDups] removes all duplicates from an array of integers.
 */
public static Integer[] removeDups(int[] arr) {
    // TODO: Implement me!
}
```

Find Negative Numbers

```java
/**
 * [findNegNums] finds all negative numbers in an array and returns those integers
 */
public static Integer[] findNegNums(int[] arr) {
    // TODO: Implement me!
}
```

Create Ransom Note

```java
/**
 * [isRansomNote] is true if you can use the letters in the magazine to create a ransom note.
 */
public static boolean isRansomNote(String note, String magazine) {
    // TODO: Implement me!
}
```

Stack with Max() function in O(1) time

```java
/**
 * MaxStack has normal Stack functionality, but also includes a .max() function that returns
 * the max value in the stack in constant time.
 */
public class MaxStack {
    // TODO: Implement me!
}
```
Braces Parsing

/**
 * isValidParen is true if the format of square and parenthesis are oriented correctly.
 * Ex: "{}()" -> true, "{{}}" -> false,
 * "{}" -> false, "))" -> false
 */
public static boolean isValidParen(String str) {
    // TODO: Implement me!
}