CMSC131

Java Collection Interface
and Collections Class
Mostly an "at-home reading" slide deck.
(not for use on Project 6)

`Collection<E>` and `Collections`

There are many ways to keep a related group of items together in a data structure and the standard Java libraries provide some support for this.

The Java standard install includes:

- a **generic interface** called `Collection<E>` which identifies some common methods for interacting with such data structures
- a **class** called `Collections` which includes some potentially useful static methods that can interact with `Collection`-based objects.

`Collection<E>` Interface: Methods

While not an exhaustive list of the methods that classes implementing the **`Collection<E>`** interface have to provide, these are a good list to discuss here.

```
boolean add(E newVal)
void clear()
boolean contains(Object ref)
boolean remove(Object ref)
int size()
boolean isEmpty()
Object[] toArray()
```

`Collection<E>` Examples

This is not an exhaustive list of the classes that implement the **`Collection<E>`** interface, but they are a good sampling of them.

- **`ArrayList`**
- **`PriorityQueue`**
- **`Stack`**

We've already explored the **`ArrayList`** in Java (before knowing there was something "special" about it in a larger context, so let's explore the other two…

PriorityQueue

The way in which a **PriorityQueue** data structure is used is to insert items where there is a natural ordering to them, and as you remove things, they come out based on that natural ordering, smallest first.

- It has all of the required **Collection** methods as well as others such as **remove()** which simply removes the "next smallest" item in the collection.
- However, note that none of this means the values are ***stored*** in sorted order, you just know the order in which things will come out if you call **remove()** repeatedly.

Usage Example: PriorityQueue

We could use a **PriorityQueue** to read in a list of floating point values, and then print them out in ascending order without having to call a sort method.

```
PriorityQueue<Double> myDoublePQ =
                                new PriorityQueue<Double>();
for (int index=0; index<n; index++) {
  System.out.print("Enter value #"+(index+1)+": ");
  myDoublePQ.add(sc.nextDouble());
}

while (!myDoublePQ.isEmpty()) {
  System.out.println(myDoublePQ.remove());
}
```

Pros of `Collection` interface

One of the advantages of using an interface like **`Collection`** is that if you have a method that should be able to operate on multiple objects as long as they have a required method that is specified in the interface, you can write the method to take a reference to any object that implements that interface!

Also, anything that implements **`Collection`** has what's required to use a "for-each" loop.

General-Use Methods

Imagine wanting to be able to print the sum of a collection of **Double** values, regardless of what type of collection is was…

```
public static double
  sumOfDoubles(Collection<Double> theCollection) {
     double sum = 0;
     for (Double val : theCollection) {
        sum+=val;
     }
     return sum;
}
```

Con of `Collection` interface

However, that same ability to make use of collections regardless of their specific type is also a "con" in the eyes of many.

It means that although there are very specific logical rules on how to access data structures such as a priority queue or stack, using either the **`Stack`** **`PriorityQueue`** or the **`Set`** that Java provides allows you to access it in other ways as well.

`Collections` class and `ArrayList`

The **`ArrayList`** class implements the **`List`** interface, which itself extends **`Collection`**.

This means that any method which can take either a **`List`** or a **`Collection`** as a parameter (which the methods in Collections do) can take an **`ArrayList`** object's reference.

These slides will look at methods that will:

- Sort the contents of any **`List`**.
- Return the **min** or the **max** of any **`Collection`** as well as ones that will **shuffle** the contents of it.

Collections: sort

Assume we had declared and filled:

```
ArrayList<Student> myList;
```

We could sort this list based on their name by simply calling:

```
Collections.sort(myList);
```

The Java-provided static `sort` method would use our `compareTo` method as the basis for sorting the list behind the scenes.

Collections: min, max

Using this same **`ArrayList`** object, without calling sort on it, we could print the record of the students with the alphabetically smallest and largest name:

```
System.out.println("Min: "+Collections.min(myList));
System.out.println("Max: "+Collections.max(myList));
```

Again, our `compareTo` method would be used as the basis for determining this behind the scenes. However, no change would be made to the actual list and the runtime would be faster since it doesn't need to sort to get these.

Consider the following…

Does a **Deck** of cards have a minimum value?

If not, then having **Deck** implement **Collection** would be logically unusual since a **Collection** *must* have a minimum value…

`Collections: shuffle`

If we wanted to randomly shuffle our list of students (perhaps to decide who to call on in class) we could simply call `Collections.shuffle(arrayList);`

The items in our list would be randomly shuffled into a new order. Each time this method is called, a different and statistically unpredictable ordering is the result.

NOTE: This is NOT the controlled shuffle you will be asked to implement in P6 (or the one P7 will ask you to do).