Java Collections Framework

In this chapter, we will look at Java library classes that are designed for storing a collection of data. These classes are part of the *Java Collections Framework*. We will see that the best way to store data depends on our purpose – how many elements we want to store, whether we need things in sorted order, and how we would like to be able to search for individual elements. In the next course, CIS 300, you will learn much more about these different ways to store data (called *data structures*) – including how to implement these Java library classes.

To use any of these classes, we need:

```
import java.util.*;
```

Generics

The Java Collections Framework makes use of a new feature in Java called *generics*. The collections classes are written so that they can hold ANY type of elements we choose to store. When we create a new collections object, we must list the type of element that we're storing. You will learn more about generics, including how to implement them in your own classes, in CIS 300.

ArrayList

The first collection class we will look at is the `ArrayList` class. This class is essentially a resizable array – the class stores the elements in an array, but if that array gets full as elements are added, it makes it bigger. When we are using the `ArrayList` class, though, we don't see anything about the array. All we see is the ability to add elements, remove elements, find elements, etc. – but we do all of this through method calls.

Creating an ArrayList

To create an `ArrayList` object, we do:

```
ArrayList<type> name = new ArrayList<type>();
```

Here, `type` is the type of elements we want to store, and `name` is the variable name we're giving to this `ArrayList`. For example, to create an `ArrayList` that holds a bunch of names, we might do:

```
ArrayList<String> namesList = new ArrayList<String>();
```

Adding elements

Once we've created a new `ArrayList` object, we can add elements to our list. This is done with the syntax:

```
name.add(elem);
```

Where `name` is the name of our `ArrayList`, and `elem` has the same type we specified when creating that list. For example, we could add names to our `namesList` object as follows:

```
namesList.add("Bob");
namesList.add("Jane");
namesList.add("Joe");
namesList.add("Mary");
```

An `ArrayList` stores elements in the same way you would expect things to be stored in the array (because the class itself uses an array to store the elements). After adding those four names, we can think of "Bob" being at index 0, "Jane" being at index 1, etc.

Changing elements

Once we have stored values in an `ArrayList`, we can change their value by specifying the index we're after and the new value we want to use. Here is the syntax:

```
name.set(index, elem);
```

Where `index` is the position we want to change (thinking of the `ArrayList` like an array), and `elem` is the new value we wish to store. For example, if we do:

```
namesList.set(1, "Allison");
```

Then the new list of values in `namesList` is: `{"Bob", "Allison", "Joe", "Mary"}`. We cannot use this syntax to change a position that has not been added yet. For example, if we try:

```
namesList.set(5, "Fred");
```

Then an exception will be thrown, as there is not currently an element at position 5.

Getting elements

We can also access elements in an `ArrayList` much like we can access elements in an array – by specifying the index we want. We can do this with the syntax:

```
name.get(index)
```

Where `index` is the position of the element we would like to access. This method call returns the element at that position, so we will usually want to store the result of the method call in a variable whose type matches the type of elements we're storing. For example, we could do:

```
String first = namesList.get(0);
```

To get the first name in `namesList`. Our `first` variable will now have the value `"Bob"`. We can also use this method to help us access every element in an `ArrayList`, using the

syntax `name.size()` to determine how many elements are in the list. For example, we can print every element in `namesList` like this:

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

This will print:

Bob
Allison
Joe
Mary

Removing elements
We can also remove elements from our list by specifying the index of the element we wish to remove. Any element that was behind the one we're removing will be shifted down to fill the empty spot. The syntax is:

```
name.remove(index)
```

Where `index` is the position with the element we wish to remove. This method does return the element that is removed, so you can optionally store the result of the method call in a variable with the same type as the elements being stored.

For example, we can do:

```
namesList.remove(1);
```

This removes `"Allison"` from our list, and shifts up `"Joe"` and `"Mary"` to fill the hole. Now the contents of the list are: Bob at index 0, Joe at index 1, and Mary at index 2.

Auto-boxing and Auto-unboxing
In Java, variable types such as int, double, char, and boolean are called *primitive types.* Other types (which are defined by a Java class or one of our own classes), like String, StringTokenizer, Random, Person (if we had written a Person class), etc. are called *reference types*. The difference between these types is a bit beyond the scope of this class, but it has to do with how these variables are stored in memory. Memory space for primitive variables is always allocated before the program runs, but memory space for reference types (objects) is not allocated until the program is already running. Moreover, a primitive variable actually holds a value, while a reference variable holds the memory address of where that object is in memory.

In any case, Java Collections are only designed to hold reference types. However, there are special reference types in Java (called wrapper classes) whose purpose is to turn a primitive type

into a reference type. The three such classes we will look at are: `Integer`, `Double`, and `Character`. For example, we can do:

```
int x = 4;
Integer val = new Integer(x);
```

This essentially turns the primitive int 4 into a reference type that wraps around the value 4. So if we want to store primitive types like ints in an `ArrayList` (or any Java collection), we could try something like this:

```
ArrayList<Integer> nums = new ArrayList<Integer>();
nums.add(new Integer(4));
nums.add(new Integer(10));
nums.add(new Integer(2));
```

If we wanted to get the value at index 1 (the 10), we could do:

```
Integer val = nums.get(1);
```

And then if we wanted to turn it back into a regular int, we could use the `intValue` command:

```
int x = val.intValue();
```

Now `x` would have the value 10. We notice, though, that having to deal with primitive types in this way is cumbersome and annoying. To smooth this process, Java has an auto-boxing/auto-unboxing feature that automatically converts between primitive types like int, double, char and their wrapper types. For example, we could create our `nums` list from above like this:

```
ArrayList<Integer> nums = new ArrayList<Integer>();
nums.add(4);
nums.add(10);
nums.add(2);
```

When we pass in an int to be added to the list, that int is automatically converted to an `Integer` (this is called *auto-boxing*). We can also do:

```
int x = nums.get(1);
```

And the `Integer` at position 1 is automatically converted to an int (this is called *auto-unboxing*). We can do something similar with double/`Double` and char/`Character`.

Iterator

An *iterator* is a tool in Java to help us step through each element in a collection. With an `ArrayList`, it is just as easy to loop through each index (with a for loop) and call get to access each value, but the same is not true of other collections.

Suppose we have this `ArrayList`:

```
ArrayList<Double> vals = new ArrayList<Double>();
Scanner s = new Scanner(System.in);
for (int i = 0; i < 10; i++) {
     System.out.print("Enter a value: ");
     double x = Double.parseDouble(s.nextLine());
     vals.add(x);
}
```

Here is how we can access and print each value in the list using an `Iterator`:

```
Iterator<Double> it = vals.iterator();
while (it.hasNext()) {
     double cur = it.next();
     System.out.println(cur);
}
```

We will be able to use this same style of looping through elements with an `Iterator` in other Java Collections.

Example
In this section, we will write a bigger example that uses an `ArrayList`. Suppose that we want to read information about a group of people from a file. Each line in the file (`info.txt`) will have the format:

name: age

We would like to store this information, and then allow the user to repeatedly search for a user by name. Each time, we will print out the age of the person with that name. First, we need a way to represent a person:

```
public class Person {
     private String name;
     private int age;

     public Person(String n, int a) {
          name = n;
          age = a;
     }

     public String getName() {
          return name;
     }
```

```
        public int getAge() {
            return age;
        }
    }
```

Now, we will write a class with a main method that creates an `ArrayList` of type `Person`. For each line in the file, we will create a new `Person` object and store it in our list. Then we will let the user search the list by name:

```
import java.util.*;
import java.io.*;
public class PersonLookup {
    public static void main(String[] args) {
        ArrayList<Person> list = new ArrayList<Person>();
        try {
            Scanner inFile = new Scanner(new
                                    File("info.txt"));
            while (inFile.hasNext()) {
                String line = inFile.nextLine();
                String[] pieces = line.split(": ");
                int age = Integer.parseInt(pieces[1]);
                Person p = new Person(pieces[0], age);
                list.add(p);
            }

            inFile.close();
        }
        catch (IOException ioe) {
            System.out.println("Trouble reading file");
        }

        Scanner s = new Scanner(System.in);
        while (true) {
            System.out.print("Enter name (Enter to quit): ");
            String name = s.nextLine();

            if (name.length() == 0) break;

            Iterator<Person> it = list.iterator();
            while (it.hasNext()) {
                Person p = it.next();
                if (p.getName().equals(name)) {
                    System.out.println(p.getAge());
                }
            }
        }
```

```
    }
}
```

TreeSet

The next collection class we will look at is the `TreeSet` class. You will learn much more about the structure of this class (and tree structures in general) in CIS 300, but for now, all you need to know is `TreeSets` are an efficient way to store data that needs to be sorted. This data is NOT stored in a linear way (like an array, with one element after the other), so it will be trickier for us to look at all the elements.

Creating a TreeSet

To create a `TreeSet` object, we do:

```
TreeSet<type> name = new TreeSet<type>();
```

Here, `type` is the type of elements we want to store, and `name` is the variable name we're giving to this `TreeSet`. For example, to create a `TreeSet` that holds a bunch of names in sorted order, we might do:

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

Adding elements

Once we've created a new `TreeSet` object, we can add elements to it. This is done with the syntax:

```
name.add(elem);
```

Where `name` is the name of our `TreeSet`, and `elem` has the same type we specified when creating that tree. For example, we could add names to our `namesTree` object as follows:

```
namesTree.add("Fred");
namesTree.add("Amy");
namesTree.add("Sam");
namesTree.add("Katelyn");
```

Again, a `TreeSet` does not store elements the way you would expect them to be stored in arrays (with indices corresponding to the different elements). Instead, the elements are stored in sorted order. To get them back, we will need to use an `Iterator`.

Iterator

We can use an iterator to access all the elements in our `TreeSet` in sorted order. Suppose we have the `namesTree` TreeSet from above. We can access and print each name like this:

```
Iterator<String> it = namesTree.iterator();
while (it.hasNext()) {
     String cur = it.next();
     System.out.println(cur);
}
```

This will print:

Amy
Fred
Katelyn
Sam

Removing elements
We can remove elements from our `TreeSet` by passing in which object we want to remove. The remaining elements in the `TreeSet` are shifted around to still be in sorted order. The syntax is:

```
name.remove(obj)
```

Where `obj` is the element we wish to remove. This method does return the whether or not it found that element (boolean) so you can optionally store the result of the method call.

For example, we can do:

```
namesTree.remove("Katelyn");
```

To remove "Katelyn". If we were to print each element again, using an iterator, it would print:

Amy
Fred
Sam

Notice that it would shift around the elements to still be sorted after removing "Katelyn".

HashMap
The last collection class we will look at is the `HashMap` class. You will again learn much more about the structure of this class (and hash tables in general) in later classes, but for now, the purpose is to store information by *key*. For example, you might want to store information about a bunch of accounts, and you want to be able to look up information on a particular account using an account number (the *key*). Arrays are actually a form of this structure, but the key is always an array index. Here, we can use ANYTHING as our "lookup" key. Like with `TreeSets`, this data is NOT stored in a linear way, so it will again be trickier for us to look at all the elements.

Creating a HashMap
To create a `HashMap` object, we do:

```
HashMap<keyType, valueType> name = new HashMap<keyType, valueType>();
```

Here, `keyType` is the type of the keys, `valueType` is the type of values that we want to store, we want to store, and `name` is the variable name we're giving to this `HashMap`. For example, to create a `HashMap` that allows us to look up state names (the *values*) by their two-letter abbreviations (the *keys*), we would do :

```
HashMap<String, String> states = new HashMap<String, String>();
```

Adding elements
Once we've created a new `HashMap` object, we can add (key, value) pairs to it. This is done with the syntax:

```
name.put(key, value);
```

Where `name` is the name of our `HashMap`, `key` is the key for the element (and has the same type as `keyType` when we created it), and `value` is the value for the element (and has the same type as `valueType` from above). For example, we could add state information to our `states` object as follows:

```
states.put("KS", "Kansas");
states.put("NE", "Nebraska");
states.put("TX", "Texas");
states.put("OR", "Oregon");
states.put("AZ", "Arizona");
```

Again, a `HashMap` does not store elements the way you would expect them to be stored in arrays (with indices corresponding to the different elements). Instead, the elements are stored in a way that makes it very fast for them to be looked up by key.

If we want to change the value associated with a particular key, we can call put with that key and just pass in a different value. It will overwrite the original value stored with that key. Note that this means that each key in a hash map must be unique.

Looking up elements
Once we've created and filled a `HashMap`, we can look up the values by key.

```
valueType val = name.get(key);
```

Where `name` is the name of our `HashMap`, `key` is the key for the element we're looking up (and has the same type as `keyType` when we created it), and `valueType` is the type of values that are stored. For example, we could get back state names from our `states` object as follows:

```
String abbrev = states.get("KS");
```

Now, `abbrev` with be "Kansas" – the value stored with the key "KS". If we try to get a value from a key that doesn't exist:

```
String anotherAbbrev = states.get("MD");
```

Then we will get back `null`.

Getting back all elements

To get back each (key,value) pair in a `HashMap`, we must first get back an `Iterator` that lets us step through each key. We will then call get with each key to get back the corresponding values. Suppose we have the `states` `HashMap` from above. We can access and print each (key,value) pair like this:

```
Iterator<String> keysIt = states.keySet().iterator();
while (keysIt.hasNext()) {
     String curKey = keysIt.next();
     String curValue = states.get(curKey);
     System.out.println(curKey + ": " + curValue);
}
```

This will print:

TX: Texas
AZ: Arizona
KS: Kansas
OR: Oregon
NE: Nebraska

Notice that this information is not being printed in the order we added it, nor is it printed in any kind of sorted order. This has to do with how things are stored in a hash map, and you will learn more about it in CIS 300. For now, just know that while getting elements out of a hash table is very fast, things are not sorted or even stored in their original order.

Removing elements

We can remove elements from our `HashMap` by passing in the key of the (key,value) pair that we want to remove. The syntax is:

```
name.remove(key)
```

Where `key` is the key of the pair we wish to remove. This method does return the value associated with that key, so you can optionally store the result of the method call.

For example, we can do:

```
states.remove("KS");
```

To remove the pair ("KS", "Kansas") If we were to print each pair again, using an iterator, it would print:

TX: Texas
AZ: Arizona
OR: Oregon
NE: Nebraska

In this example, the remaining elements were shifted up to cover the hole left by Kansas, but this is not always the case. You should think of hash maps as storing elements in a fairly random order (it is not actually random, but it looks that way).

Example: Word frequency
Finally, we will look at an application for which a hash table is appropriate. Suppose we have an input file that contains several paragraphs of text. We want to use a hash map to help count the frequency of each word in the document. Here's how:

```
public void countOccurrences(String filename) {
      Scanner inFile = new Scanner(new File(filename));
      HashMap<String, Integer> h = new HashMap<String, Integer>();

      while (inFile.hasNext()) {
            String line = inFile.nextLine();
            String delims = ",:.;()?! ";
            StringTokenizer tok = new StringTokenizer(line, delims);

            while (tok.hasMoreTokens()) {
                  String word = tok.nextToken();
                  //if we haven't seen it yet, give it a count of 1
                  if (h.get(word) == null) {
                        h.put(word, 1);
                  }
                  //otherwise, make its count one bigger
                  else {
                        int count = h.get(word);
```

```
                        h.put(word, count+1);
                    }
            }

            inFile.close();

            //now, print all the word frequencies
            Iterator<Integer> counts = h.keySet().iterator();
            while (counts.hasNext()) {
                  String s = counts.next();
                  int num = h.get(s);
                  System.out.println(s + ": " + num);
            }
}
```