ArrayLists and Sorting
Array Lists

The Java Collections Framework gives implementations for some standard data structures. We will do our own versions of several of these as we study them.

In Lab 2 we will re-implement the ArrayList class.
Example 1
I want to write a program that reads a bunch of numbers (integers) from the user, then at the end prints them out. To do this in C we would use an array, and hope it is big enough to hold the data. If we run out of room we could make a bigger array.

Python would solve this with a list. In Java we have **ArrayLists** to hold the data. ArrayList is a generic class that takes a type parameter:

```java
public class ArrayList<E> {
    ....
}
```
There are many useful methods of class ArrayList, but we will focus here on a few. If L is an ArrayList, then

- L.size() is the number of entries in L
- L.add(x) appends x to the end of the list
- L.add(i, x) adds x to the list at position i, shifting the tail of the list back one to make room.
- L.get(i) is the element at position i.
This makes our program very simple. We start by creating the ArrayList:

```java
ArrayList<Integer> L = new ArrayList<Integer>();
```

Each time we get a new data value $x$ we add it to the list:

```java
L.add(x);
```
At the end we print the list: using either
for (int x: L )
    System.out.println(x)

or, if we want to use indices
for (int i = 0; i < L.size(); i++)
    System.out.println( L.get(i) );

For the complete code see SimpleList.java
Example 2

I want to write a program that will read a sequence of names, sort it, and print it back in alphabetical order. Typical names might be

bob
Fred Flintstone
John Frederick Oberlin
Charles Philip Arthur George Windsor

Naturally, we want to sort by last name.
What class structures will we use for this?
I like this structure:

First, we’ll have a Name class to hold the names. This has two String class variables: first and last (which might more properly be called givenNames, familyName). We will make a constructor that takes a string holding the full name and splits it up into those fields.
Then we need to store a whole sequence of these. This seems like another good use of ArrayLists, so we’ll make an ArrayList<Name> object to hold the list.

At this point the program is much like our integer list program. The one difference is that we need to sort the list.
There are two ways to get Java to handle the sorting. One way is to have the Name class implement the Comparable<Name> interface, which just means that we give it a compareTo method. This takes a Name n as an argument and returns -1, 0, or 1 if this < n, this == n, or this > n.
The *Collections* class has a static method `sort( )` that can be applied to any list whose base type implements the `Comparable` interface. So we add method `compareTo( )` to `Name`, we note that `Name` implements the `Comparable` interface, and then we call `Collections.sort(L)`, where `L` is the list of names to be sorted.
The other way is to make a class that implements the Comparator interface. This class needs a method \( \text{Compare}(n1, n2) \) where \( n1 \) and \( n2 \) have the base type of the list being sorted. \( \text{Compare} \) returns \(-1, 0 \) or \(1\) if \( n1 < n2 \), \( n1 == n2 \) or \( n1 > n2 \).

The Collections class has a method \( \text{sort}(L, \text{comp}) \) where \( \text{comp} \) is an object of the class that implements the Comparator interface. This time we don't need to make any changes to the Name class; we add the comparator class as a nested class inside our application program.
See the NameListComparable and NameListComparator programs.
Example 3

I want to write an inventory program. This will read in lines from the user that are formatted as

<object> <count>

For example,

hammer 23

This indicates that we have found 23 hammers. If we see another line

hammer 5

then we need to update our count of hammers to 28.
What we are doing here is associating Strings with ints. We could do this by keeping a list of strings and a corresponding list of ints but that would be a pain to maintain. In Python you would use a Dictionary to hold this data. In Java such an associative structure is called a map.
The Collections Framework has two map structures -- HashMaps and TreeMap. We will implement both this semester, so you'll be able to see the differences. For now we'll just do this with a HashMap.
You can think of HashMaps in the way you think of dictionaries in Python -- they associate values to keys. The *key* field is like an index; in our inventory program it will be the name of the item, like "hammer". The *value* field is the value being associated with the corresponding key; in our program this is the count of how many such items we have.
HashMaps take two class parameters -- one class for the key and one for the values, as in

\[
\text{HashMap}<\text{Key, Value}>
\]

For our inventory program the structure is

\[
\text{HashMap}<\text{String, Integer}>
\]

We construct our inventory structure with

\[
\text{HashMap}<\text{String, Integer}>\text{inventory} = \text{new HashMap}<\text{String, Integer}>();
\]
There is a *put* method for inserting into the HashMap:

```java
inventory.put("hammer", 5)
```
associates the number 5 to the string "hammer".
Similarly, there is a *get* method for finding the value associated with a particular key:

```python
inventory.get("hammer")
```

tells you the number associated with string "hammer".
Just as with Python, there is a runtime error if you try to get the value associated with a string that is not one of the current keys, so it is necessary to check that something is a key before using it as an argument to get. After getting *name* and *count* values from the input, here is the code for updating the HashMap:

```java
    if (inventory.containsKey(name)) {
        int current = inventory.get(name);
        inventory.put(name, current+count);
    }
    else
        inventory.put(name, count);
```
In Python there is a method for obtaining a list of the current keys in a HashMap; in Java the keySet() method gives a set of the current keys. You can't iterate through sets with a for-loop the way you can with lists, but you can obtain an iterator structure that will do this. You will implement iterators in Lab 4. Here is the code for using an iterator to print the data in our HashMap:
Set keys = inventory.keySet();
Iterator<String> iter = keys.iterator();
while (iter.hasNext()) {
    String name = iter.next();
    Integer count = inventory.get(name);
    System.out.printf("%-10s %3d \n", name, count);
}