Use of the ArrayList class

The ArrayList class is very similar to the Vector class. It also manages a collection of objects, and as the name indicates, does so with an array implementation. This is also a template class, so you can declare an ArrayList of any sort of Object. Here is an example declaring an empty ArrayList of String:

ArrayList<String> stooges = new ArrayList<String>();

The same sort of issues (dealing with casting) occur when you try to retrieve an element from an ArrayList that is of a different type than the template. (For this example, if we stored an object from a class that inherited from String, we would have to cast a item retrieved from this ArrayList to the proper type.)

As with the Vector, an iterator can be used with an ArrayList as well. Here's the example from the code shown in class:

```
for (String s: stooges)
  System.out.println(s);
```

Here's stooges is the name of the ArrayList reference that we are iterating through.

Here are a few of the methods from the ArrayList class:

```
// Adds v to the end of this list and returns true.
public boolean add(T v);

// Insert value v intp the list such that v has index i. Any
// preexisting elements with indices I or greater are shifted
// backwards by one element position.
public void add(int I, T v);

// Removes all elements from the list.
public void clear();

// Returns whether the list has an element with value v.
public boolean contains(Object v);

// If i is a valid index, it returns the ith element; otherwise
// an exception is generated.
public T get(int i);

// If i is a valid index, it removes the ith element from the list by
// shifting forward elements i+1 and on. In addition, the
// removed element value is returned. Otherwise an exception is
// generated.
public T remove(int i);

// If v is in the list, method removes v from this list, shifts
// items to the right, returns true. Otherwise, returns false.
public boolean remove(T v);

// Sets the item at index to v.
public void set(int index, T v);

// Returns the size of this list.
public int size();
```

Collections Algorithms

For the various collection classes, there exist some common methods that perform operations on those different types of collections. (Namely, these Collection methods are such that you can pass in different types of collections - either a Vector, ArrayList or one of Java's other predefined collections, and the method will perform the appropriate operation on that collection.)

It's important to note that these methods are static - thus, they do not operate on any specific type of object. Instead, they belong to the class Collections and can be called in the following fashion:

```
Collections.max(redteam);
```

In order for these methods to work, it's important that the Collection of Objects is actually a collection of a type that implements Comparable. Comparable is an abstract class that other classes can implement. In order to implement Comparable, a class must contain a compareTo method. Here's the signature necessary for a compareTo method:

```
public int compareTo(T v);
```

where T is the class in which the method resides. (Thus, this method is NOT a template method.)

Furthermore, the method much return a negative integer if the current object comes before v in order, a positive integer if the current object comes after v in order, and 0 if the two objects are equal.

Collections Method Signatures

```
// Returns the number of elements of collection c equal to the
// the object v.
public static int frequency(Collection<?> c,
Object v);

// Returns the maximum element of collection c with respect to
// its natural ordering.
public static <T, Comparable<? Super T>> T
max(Collection<? Extends T> c)

// Returns the minimum element of collection c with respect to
// its natural ordering.
public static <T, Comparable<? Super T>> T
min(Collection<? Extends T> c)

// Reverses the order of the elements in list a.
public static void reverse(List<?> a);

// Pseudorandomly permutes list a.
public static void shuffle(List<?> a);

// Sorts list a into nondescending order using the natural
// ordering.
public static <T extends Comparable<? Super
T>> void sort(List<T> a);
```

In order to call these methods, if we have a collection of Comparable objects, the work is fairly easy. Consider the example where redteam is an ArrayList<String> reference. They we can do the following:

```
Collections.sort(redteam);
```

Collections Example: Writing a Static Method

A static method in Java is very much like a regular function in C. The method performs some general task and takes in all necessary information in its parameter list. In Java, all methods reside in classes, and static methods belong to the class. As discussed before, to call a static method, we do the following:

```
Classname.methodName(parameters);
```

The one exception to this is when the method call is made within the class in which the method is defined. In this case, the class name can be omitted and the call can be made as follows:

```
methodName(parameters);
```

Another important detail to discuss is how parameters are passed in Java. Luckily, the rules are simple:

1) All primitives are passed by value. (So it's impossible to write a swap method that swaps the values of two int variables, for example.)

2) All non-primitives are passed by reference. (So, all changes made to any object inside of a method are reflected in the object passed in as an actual parameter when that method call completes.)

In the example shown in class, an ArrayList<String> reference is passed in, and the method fills up the corresponding object with the information gathered from the user.

An Array of Objects

The way to declare a regular array of objects is as follows:

```
Class[] var_name = new Class[SIZE];
```

What this declaration does is allocate space for SIZE number of references, but NOT SIZE number of objects of type Class.

For example, consider the PhoneBookEntry class that starts as follows:

```
public class PhoneBookEntry {

  private String first_name;
  private String last_name;
  private int number;

  public PhoneBookEntry(String name_f,
                        String name_l,
                        int num) {

    first_name = name_f;
    last_name = name_l;
    number = num;
  }

  …
}
```

The following declaration does NOT create 5 PhoneBookEntry objects:

```
PhoneBookEntry[] mine = new
PhoneBookEntry[5];
```

What it allocates is five array locations. Each of those array locations holds a PhoneBookEntry reference, that has yet to reference a PhoneBookEntry object. In order to really fill in this sort of structure, something like the following must be done:

```
for (i=0; i<size; i++) {
  String name_f = infile.next();
  String name_l = infile.next();
  int num = infile.nextInt();
  mine[i] = new PhoneBookEntry(name_f,
                        name_l, num);
}
```

It is assumed that infile is a previously defined Scanner reading from an appropriate location and that i is declared as an integer.

In this code segment, we create one object at a time and have each subsequent reference reference the newly created object.

compareTo method for PhoneBookEntry

```
public int compareTo(PhoneBookEntry e)
{

  // Compare last names.
  int temp = (this.last_name).compareTo
                  (e.last_name);

  // If these differ, return the
  // comparison between these strings.
  if (temp !=0) return temp;

  // Compare first names.
  temp = (this.first_name).compareTo
                  (e.first_name);

  // Checks the case that first names
  // differ while last names don't.
  if (temp !=0) return temp;

  // Breaks ties when first and last
  // names match.
  return this.number - e.number;
}
```