CSCI 1933 Project 2

Herding the Elephants: Lists and Interface

Note: The project is due on **Wednesday, February 28th** by **11:55 PM**.

Instructions

Please read and understand these expectations thoroughly. Failure to follow these instructions could negatively impact your grade. Rules detailed in the course syllabus also apply but will not necessarily be repeated here.

- **Due:** The project is due on **Wednesday, February 28th** by **11:55 PM**.
- **Identification:** Place you and your partner's x500 in a comment in all files you submit. For example, `//Written by shino012 and hoang159`.
- **Submission:** Submit a `zip` or `tar` archive on Moodle containing all your `java` files. You are allowed to change or modify your submission, so submit early and often, and *verify that all your files are in the submission.*

  Failure to submit the correct files will result in a score of zero for all missing parts. Late submissions and submissions in an abnormal format (such as `.rar` or `.java`) will be penalized. Only submissions made via Moodle are acceptable.
- **Partners:** You may work alone or with *one* partner. **Failure to tell us who is your partner is indistinguishable from cheating and you will both receive a zero**. Ensure all code shared with your partner is private.
- **Code:** You must use the *exact* class and method signatures we ask for. This is because we use a program to evaluate your code. Code that doesn't compile will receive a significant penalty. Code should be compatible with Java 8, which is installed on the CSE Labs computers. We recommend to not use IDEs for your implementations.
- **Questions:** Questions related to the project can be discussed on Moodle in abstract. This relates to programming in Java, understanding the writeup, and topics covered in lecture and labs. **Do not post any code or solutions on the forum**. Do not e-mail the TAs your questions when they can be asked on Moodle.
- **Grading:** Grading will be done by the TAs, so please address grading problems to them *privately*.

IMPORTANT: You are NOT permitted to use ANY built-in libraries, classes, etc. Double check that you have NO import statements in your code, except for those explicitly permitted in the File I/O section.

Code Style

Part of your grade will be decided based on the "code style" demonstrated by your programming. In general, all projects will involve a style component. This should not be intimidating, but it is fundamentally important. The following items represent "good" coding style:

- Use effective comments to document what important variables, functions, and sections of the code are for. In general, the TA should be able to understand your logic through the comments left in the code.

  Try to leave comments as you program, rather than adding them all in at the end. Comments should not feel like arbitrary busy work - they should be written assuming the reader is fluent in Java, yet has no idea how your program works or why you chose certain solutions.

- Use effective and standard indentation.
- Use descriptive names for variables. Use standard Java style for your names: `ClassName, functionName, variableName` for structures in your code, and `ClassName.java` for the file names.

Try to avoid the following stylistic problems:

- Missing or highly redundant, useless comments. `int a = 5; //Set a to be 5` is not helpful.
- Disorganized and messy files. Poor indentation of braces (`{` and `}`).
- Incoherent variable names. Names such as `m` and `numberOfIndicesToCount` are not useful. The former is too short to be descriptive, while the latter is much too descriptive and redundant.
- Slow functions. While some algorithms are more efficient than others, functions that are aggressively inefficient could be penalized even if they are otherwise correct. In general, functions ought to terminate in under 5 seconds for any reasonable input.

The programming exercises detailed in the following pages will both be evaluated for code style. This will not be strict – for example, one bad indent or one subjective variable name are hardly a problem. However, if your code seems careless or confusing, or if no significant effort was made to document the code, then points will be deducted.

In further projects we will continue to expect a reasonable attempt at documentation and style as detailed in this section. If you are confused about the style guide, please talk with a TA.

Introduction

In this project you are going to implement a list [1] interface to construct your own ArrayList data structure. Using this you will construct an **ElephantHerd** to hold a family of elephants [2].

[1]. Lists:
A List is a list of ordered items that can also contain duplicates. In Java, lists are constructed either using an array or linked list data structure. The implementations for each have certain pros and cons with respect to cost of space and runtime. In this project, you will implement lists using only an array data structure from a custom List interface.

[2]. Inheritance: Interface:
Interfaces are an important aspect of inheritance in Object Oriented Programming. All methods defined in an Interface are un-implemented and required to be implemented by an inheriting class. In Java an Interface class is inherited by other classes using the keyword *implements*. See the example code below.

1 List: An interface

A List must consist of specific methods irrespective of underlying data structure. These methods are defined as part of an interface that you are required to inherit in your array list and linked list implementations. **Refer to List.java** for methods and their definitions. Note that methods have generic types* and you are required to implement your inherited classes as generic types too (continue reading to see what it means...).

*A generic type is a generic class or interface that is parameterized over types. In the context of `List`, `T` is the type of the object that is in the list, and note that `T` extends `Comparable`.

Inheritance Java Example:

```
// An interface.
interface IName {
        public void printName();
}

// This class implements the Name interface.
class PeopleName implements IName {
        String firstName;
        String secondName;

        // Need to implement printName().
        public void printName() {
                System.out.println(this.firstName + " " + this.secondName);
        }
}
```

1.1 Array List Implementation

The first part of this project will be to implement an array list. Create a class `ArrayList` that implements all the methods in `List` interface. Recall that to implement the `List` interface and use the generic compatibility with your code, `ArrayList` should have following structure:

```
public class ArrayList<T extends Comparable<T>> implements List<T> {
  ...
}
```

The underlying structure of an array list is (obviously) an array. This means you will have an instance variable that is an array. Since our implementation is generic, the type of this array will be `T[]`. Due to Java's implementation of generics[†], you **CANNOT** simply create a generic array with:

```
T[] a = new T[size];
```

Rather, you have to create a `Comparable` (since `T` extends `Comparable`)[‡] array and *cast* it to an array of type `T`.

```
T[] a = (T[]) new Comparable[size];
```

Your `ArrayList` class should have a single constructor:

```
public ArrayList() {
  ...
}
```

that initializes the underlying array to a length of 2.

[†] specifically because of type erasure

[‡] had `T` not extended `Comparable`, you would say `T[] a = (T[])new Object[size];`

Implementation Details

- In addition to the methods described in the `List` interface, the `ArrayList` class should contain a private class variable `isSorted`. This should be initialized to `true` in the constructor (because it is sorted if it has no elements) and updated when the list is sorted, or more elements are added or set. For the purposes of this class, `isSorted` is only true if the list is sorted in ascending order.
- When the underlying array becomes full, both `add` methods will automatically add more space by creating a *new* array that is **twice** the length of the original array, copying over everything from the original array to the new array, and finally setting the instance variable to the new array. *Hint: You may find it useful to write a separate* private *method that does the growing and copying*
- When calling either `remove` method, the underlying array should *no longer have that spot.* For example, if the array was `["hi", "bye", "hello", "okay", ...]` and you called `remove` with index 1, the array would be `["hi", "hello", "okay", ...]`. Basically, the only `null` elements of the array should be *after* all the data.
- Initially and after a call to `clear()`, the `size` method should return 0. The "size" refers to the number of elements in the *list* , NOT the length of the *array.* After a call to `clear()`, the underlying array should be reset to a length of 2 as in the constructor.

After you have implemented your `ArrayList` class, **include junit tests** that test all functionality.

2 An Elephant Herd

You will use array list and linked list implementations to now construct a herd of elephants. Elephants have a *name*, *age* and *height*.

You will use the ArrayList data structure to construct this Elephant Herd. You are provided with `Elephant.java` which implements Comparable (Refer to the **Elephant.java** file for details) which is a class with three properties: `name, age, and height`, setters and getters, a `compareTo()` and a `toString()` method.

2.1 The Herd

Create a class `ElephantHerd`. To create this herd you will use your `ArrayList` class as the underlying object list. The type for the object in the list will be `Elephant`. Your `ElephantHerd` should include the following methods:

- `private List<Elephant> list` – Your underlying list of Elephants.
- `public ElephantHerd()` – This constructor will initialize the underlying list.
- `public boolean add(Elephant ellie)` – This will add `ellie` to the end of the list and return `true` if successful, `false` otherwise.
- `public Elephant find(String name)` – This will try to find an elephant with name field that *contains* `name`. Note that the `name` need not be exactly the same as the name of elephant. You can use the built in String method `public boolean contains(String anotherString )`[§]. Return `null` if no Elephant was found.
- `public Elephant remove(int index)` – This will remove the elephant object currently at index `index`, if `index` is out of bounds, return `null`.
- `public void sort()` – This will sort the list in order of height, from tallest to shortest. Note that you cannot just use the ArrayList sort method that you wrote earlier, because that method sorts based on the results of `compareTo()`, not on the basis of height.
- `public Elephant[] getTopKLargestElephants(int k)` – This will return an *array* of length k containing the top-k elephants sorted by their height, from tallest to shortest. If the list is empty, return null. If the number of elephants (M) in the list is smaller than k, then return an array of length M.

After you have implemented your `ElephantHerd` class, **write junit tests** that test all functionality.

[§] The actual signature of `contains` is `public boolean contains(CharSequence s)` but you don't have to worry about that

3 File Input

Now that you have created your ElephantHerd, it is time to make a convenient way to input the data for the herd. You will do this by creating an ElephantReader class which will be able to read data from a file into an ElephantHerd object and to write data from the herd to a file. To do this, you will need to import File, Scanner, and PrintWriter. These are the only imports allowed.

To read the data, you will create a File object, and then use a Scanner to parse the data. The following code gives examples of how to read and write to files called "fileName".

```
// assume our filename is stored in the string fileName
Scanner s = null; // declare s outside try-catch block
try {
        s = new Scanner(new File(fileName));
} catch (Exception e) { // returns false if fails to find fileName
        return false;
}
// Now use s in the same way we used Scanners previously for user input to
    write to an arbitrary textfile, do the following:
// assume our filename is stored in the string fileName
PrintWriter p = null; // declare p outside try-catch block
try {
        p = new PrintWriter(new File(fileName));
} catch (Exception e) {
        return false;
}
```

At this point, it is not critical that you understand exactly how the try/catch block works, but know that the contents of the "try" portion are what could throw an Exception, while the contents of the "catch" block are what you want the program to do if the Exception is thrown.

This class only contains two method:

- public static boolean readElephants(ElephantHerd e, String fileName) – This method removes all previous elephants in e and replaces them with elephants from the file of the given name. If there is an error, or e is null, return false. Otherwise, return true. You can assume that the file is formatted such that there is data for one elephant per line. The data will be in the form of "name age height". An example text file is provided for testing.
- public static boolean writeElephants(ElephantHerd e, String fileName) – This method will write all elements of e to a file of the given name. If there is an error, or e is null or

empty, return false. Otherwise, return true. The file should be written using the `toString` function in `List`. This should give the same format as the file being read, so a written file can be reloaded later.

4 Iterators (Honors)

Note: This section is **required** for students in Honors section only. Optional for others but no extra credit.

An iterator is an object that traverses a list, going through each element exactly once.

This section will require you to write another class, and to make modifications to the `ArrayList` class.

You will write a `ArrayListIterator` class which will iterate over a list. This iterator should implement java's iterator interface in addition to the `List<T>` interface. Make sure to import `java.util.Iterator`. This class will have two functions and a constructor. It will also need class variables to store a pointer to its ArrayList and the current index.

1. `ArrayListIterator(ArrayList a)` – the constructor. This constructor will never be directly called by the user. Instead, it will be called in the `iterator()` function in the ArrayList class.
2. `hasNext()` – This will return `true` if there is another object to iterate on, and `false` otherwise.
3. `next()` – This will return the next object if there is one, and `null` otherwise.

The first line of the `ArrayListIterator` class should be as follows:

```
private class ArrayListIterator<T extends Comparable<T>> implements Iterator<T>
```

Note that in order for a class to be private, it must be in the same document as another class, and within the curly braces of that class. This means that `ArrayListIterator` should be in the `ArrayList.java` file, and should be in the curly braces of `ArrayList`, with the methods of `ArrayList`.

You will also need to make some modification to the `ArrayList` class. First, the class now needs to implement `Iterable`.

```
public class ArrayList<T extends Comparable<T>> implements Iterable<T>, List<T>
```

Secondly, you will need to add the method `public Iterator<T> iterator()`. This method should return an ArrayListIterator object by calling the ArrayListIterator constructor and passing itself to the constructor (via the this keyword).

Make sure to create junit tests to ensure that your iterator functions as desired. Example code using the iterator is provided below:

Iterator Example:

```
// the main method of a class using the ArrayListIterator
ArrayList<String> arr = new ArrayList<String>();
arr.add(”hello”);
arr.add(”how”);
arr.add(”are”);
arr.add(”you?”);
Iterator<String> it = arr.iterator();
System.out.println(it.next()); //prints ”hello”
System.out.println(it.hasNext())); //prints true

while (it.hasNext())
{
        System.out.print(it.next()); //prints ”how are you?”
}
```