CSCI 1933 Project 3

Finding the Stars: Lists, Interface and Abstract

Note: The project is due on **Monday, March 27th** 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 **Monday, March 27th** 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.

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 and LinkedList data structure. Using these you will construct a **Celestial Bodies** database to include a list of different types of Stars present in our Universe. You are require to make the database compatible for multiple type of Stars like Sequence Stars, WhiteDwarfs, and Red Giants using concept of class inheritance [2]. Finally, you are going to find the biggest and the brightest stars in our own small universe of objects and also look out for Black holes.

[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 both array and linked list data structure from a custom List interface.

[2]. Inheritance: Interface and Abstract classes:
Interface and Abstract classes are an important aspect of inheritance in Object Oriented Programming. Both plays a critical role in designing objects in the way that makes it easy to design and implement complex objects. On one hand where all methods defined in an Interface are un-implemented and required to be implemented by an inheriting class; the Abstract classes can have combinations of method definitions that are implemented and some that are not, defined as *abstract* methods. In Java an Interface class is inherited by other classes using the keyword *implements* while Abstract class requires to be inherited using *extends*. See an 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();
}

// An abstract.
abstract class Name {
        // Abstract have variables unlike interface.
        String firstName;
        String secondName;

        // An abstract method.
        abstract 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("%s %s", this.firstName, this.secondName);
        }
}

// This class extends the Name class.
class PeopleName extends Name {
        public void printName() {
                System.out.println("%s %s", 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`.

[†] specifically because of type erasure

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

```
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.

Implementation Details

- 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 a main method** that tests all functionality.

1.2 Linked List Implementation

The second part of this project will be to implement a linked list. Create a class `LinkedList` that implements all the methods in `List`. Recall again that to implement the `List` interface, `LinkedList` should be structured as follows:

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

The underlying structure of a linked list is a node. This means you will have an instance variable that is the first node of the list. The provided `Node.java` contains a generic node class that you

will use for your linked list implementation[§].

Your `LinkedList` class should have a single constructor:

```
public LinkedList() {
  ...
}
```

that initializes the list to an empty list.

Implementation Details

- Initially and after a call to `clear()`, the size should be zero and your list should be empty.
- In `sort()`, **do not use an array or ArrayList** to sort the elements. You are required to sort the values using only the linked list data structure. You can move nodes or swap values but you cannot use an array to store values while sorting.
- Depending on your implementation, remember that after sorting, the former first node may not be the current first node.

After you have implemented your `LinkedList` class, **include a main method** that tests all functionality.

[§] You may implement your linked list as a *headed* list, i.e., the first node in the list is a 'dummy' node and the second node is the first element of the list, or a non-headed list, i.e., the first node is the first element of the list. Depending on how you choose to implement your list, there will be some small nuances.

2 A Celestial Database

You will use array list and linked list implementations to now construct a celestial database. For this project, this database will include a list of stars from our Universe.

Stars are one of the most common form of celestial bodies present in our Universe. They come in various forms and often characterized by their *mass*, *size* and *lifespan*. For this project, we will focus only on mass and size. Among many types or phases of stars that exists we will include only three types, namely Sequence Stars, Red Giants, and White Dwarfs. Sun, the closest star and the center of our solar system is one of the Sequence stars, characterized by average mass and size. The Red Giants, reddish or orange in color, are generally 100 times larger than size of Sequence Stars and often seen as the starting of dying phase of star. White Dwarfs are the remnant of an average-sized star that passed through the red giant stage of its life.

You will use the ArrayList or LinkedList data structure to construct this Celestial database of Stars. You are provided with `Star.java` (**Refer to Star.java** file for details) which is an abstract class with three properties: `name, mass, and size`, setter and getter, a compareTo function, and two `abstract` methods:

- `public abstract boolean isBlackHole();` — Return `true` if star is a blackhole, otherwise `false`.
- `public abstract String toString();` — Print description of each star as SequenceStar, RedGiant or WhiteDwarf with their respective mass and sizes.

You are required to extend the base `Star` class to create three new classes for each: Sequence, RedGiant, and WhiteDwarf. Each of the three classes must have constructor (Hint: You cannot instantiate an abstract class from its constructor. See the use of `super()`.) and override the abstract methods `isBlackHole()` and the respective `toString()` method as follows:

Sequence

- `public Sequence(String name, double mass, double size)`
- `public boolean isSun()`: Return `true` if mass == 2 ($\times 10^{30}$ KG) and size == 430 ($\times$1000 miles), otherwise `false`.
- `public String toString()`: Print description for the star in the following format: "$<$ $Name$ $>$: A Sequence Star with mass = $< XXX.YY >$ KG; Size = $< XXX.YY >$ miles".
- `public boolean isBlackHole()`: Return `true` if the mass of the star is greater than 1000 units ($\times 10^{30}$ KGs) and size less than 50 ($\times$1000 miles), otherwise `false`.

RedGiant

- `public RedGiant(String name, double mass, double size)`
- `public boolean isSuperGiant()`: Return `true` if mass $>$ 100 and size $>$ 100, otherwise `false`.
- `public boolean isBlackHole()`: Return `true` if star is a Super RedGiant, otherwise `false`.
- `public String toString()`: Print description for the star in following format: "$<Name>$: A RedGiant (replace with SuperGiant if above condition is met) with mass = $<XXX.YY>$ KG and size = $<XXX.YY>$ miles.

WhiteDwarf

- `public WhiteDwarf(String name, double mass, double mass)`
- `public boolean isBlackHole()`: Return `false` (always).
- `public String toString()`: Print description for the star in following format: "$<Name>$: A WhiteDwarf with mass = $<XXX.YY>$ KG and size = $<XXX.YY>$ miles.

2.1 The Database

Now that you have your 3 types of stars, create a class `CelestialDatabase`. To create this database you will use your `ArrayList` and `LinkedList` classes as the underlying object list. The type for the object in the list will be `Star`. Your `CelestialDatabase` should include the following methods:

- `private List<Star> list` – Your underlying list of Stars.
- `public CelestialDatabase(String type)` – This constructor will initialize the underlying list based on what the value of `type` is. If `type.equals("array")`, your underlying list should be a `ArrayList`. If `type.equals("linked")`, your underlying list should be a `LinkedList`. You may assume no other strings will be used with this constructor.

  Hint: Both `List<Star> l = new ArrayList<Star>();` and `List<Star> l = new LinkedList<Star>();` are valid in Java since `ArrayList` and `LinkedList` both implement `List`.
- `public boolean add(Star s)` – This will add `c` to the end of the list and return `true` if successful, `false` otherwise.
- `public Star find(String name)` – This will try to find a star with name field that *contains* `name`. Note that the `name` is not necessary to be exactly same to the name of Star. You can use

the built in String method `public boolean contains(String anotherString)`[¶]. Return `null` if no Star was found.

- `public Star findSun()` – This will try to find a Sequence Star that matches the characteristics of Sun (see `isSun()`). Return `null` if no match was found.
- `public Star remove(int index)` – This will remove the star object currently at index `index`, if `index` is out of bounds, return `null`.
- `public Star get(int index)` – This will return the star object currently at index `index`. If `index` is out of bounds, return `null`.
- `public void sort()` – This will sort the list in order of mass based on `compareTo`.
- `public Star[] getStarsByType(String type)` – This will return an *array* of `Star` objects that have the type `type`. You can assume that `type` will only take on the values `"sequence"`, `"redgiant"`, or `"whitedwarf"`.
  Hint: instanceof
- `public Star getHeaviestStar()` – This will return the star with the biggest mass. In the case where no objects are in the list, return `null`.
- `public Star getHeaviestRedGiant()` – This will return the one of the RedGiant star with the biggest mass. In the case where no `RedGiant` objects are in the list, return `null`.
- `public Star getLargestStar()` – This will return the star with the largest size. In the case where no `Star` objects are in the list, return `null`.
- `public Star[] listBlackHoles()` – This will return a list of stars that are black holes. In case there is no blackhole, return `null`.
- `public Star[] getTopKHeaviestStar(int k)` – This will return an *array* of length k containing the top-k stars among all types sorted by their mass. If no object in the list, return null. In case, if number of stars (M) in the list is smaller than k, then return an array of length M.
- `public Star[] getTopKLargestStar(int k)` – This will return an *array* of length k containing the top-k stars among all types sorted by their size. If no object in the list, return null. In case, if number of stars (M) in the list is smaller than k, then return an array of length M.
- `public int[] countStars()` – This will return an array of three integers where first is the count of Sequence stars (`Sequence` object) in the list, second is count of RedGiant stars, and third one is count of WhiteDwarfs in the database. For example if there exists a list of two Sequence Stars, one RedGiant, and zero WhiteDwarfs then output should be: `2, 1, 0`

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

After you have implemented your `CelestialDatabase` class, **include a main method** that tests all functionality. For the methods, where return type is a `Star` object or `Star[]` array use corresponding `toString()` method to print the details of the Star.

3 Analysis

Now that you have implemented and used both an array list and linked list, which one is better? Which methods in `List` are more efficient for each implementation?

For each of the 13 methods in `List`, compare the runtime (Big-O) for each method and implementation. Include an `analysis.txt` with your submission structured as follows:

Method	ArrayList Runtime	LinkedList Runtime	Explanation
`boolean add(T element)`	`O(...)`	`O(...)`	...
`boolean add(int index, T element)`	`O(...)`	`O(...)`	...
⋮	⋮	⋮	⋮

Your explanation for each method only needs to be a couple sentences briefly justifying your runtimes.

4 Dynamic Resizing Array (Honors)

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

In the List interface, you implemented a method `boolean add(T element);` to add any element to the array. In case when the array is full, you double the length of array. We ask you to answer two related questions:

1. Find and explain the *average* runtime analysis of sequence of `add()` operations. Include this in your analysis.txt file.
2. Reimplement the methods `boolean remove(T element)` and `T remove(int index)` for ArrayList to shrink the length of array to half of the original array when number of elements (or size of array) are less than quarter (1/4th) the length of original array, after removing an element. You can keep this same for your implementations in Section 1.1.