CS 310: Assignment 1

Spring 2017

Topic: Interfaces and Lists

Points Possible: 150

Due By: 2/21/2017 2359

Late Work: Not accepted

Overview

Objectives

- Use industry-standard development tools.
  - Java 1.8
  - Maven
  - IDEA Community Edition (*optional*)
- Implement several major classes
  - Compliant java.util.List by extending the java.util.AbstractList
  - Generic, Comparable objects
- Begin advanced development of the student's Java skills.
  - Work with a multi-class design
  - Develop a working driver application
  - Generics in storage class objects
  - Extract data from a CSV file

Arrays frequently offer a convenient mechanism for grouping together a collection of objects, and they remain central to computer science, for they offer reliable, fast access to any element in the collection. That said, they possess a major limitation: a fixed size. When using arrays, one must specify the array's size, or capacity, prior to instantiation. From the moment it comes into existence, the size the array requires remains constant. This works well for many applications, but it falls short as a general solution.

The `ArrayList` addresses this deficiency by including automatic resizing logic in the data structure itself. Thus, as the user adds elements to the collection, it dynamically increases or decreases its size to accommodate the change. The `ArrayList` grows and shrinks to roughly encompass the elements inside the collection. `ArrayLists` never run out of space (unless the host does).

For this assignment, each student shall independently extend Java's `AbstractList` creating a custom `ArrayList.` To drive the data structure, however, students must develop a working driver which simulates a random combat between a list of opponents provided in a CSV file.

Operation

The user shall call your program from the command line as such:

`java -jar assign1-1.0-SNAPSHOT.jar [IN-`*`File`*`] [`*`OUT-File`*`]`

This application runs from the command line. It accepts two runtime parameters: the path to the input file to use, and the path to the output file to create. Upon execution, the program reads in all of the creatures in the input CSV file. It then creates unique instances of each one building up a total creature list.

The driver program then simulates a total elimination war within the group. Each round the application randomly creates a list of pairs of opponents from the surviving creatures. The stronger of the two

survives the round while the weaker enters a the list of vanquished opponents. A creature fights only one opponent per round. When there are an even number of opponents, every creature fights once in the round. If there are an odd number of survivors, the strongest survivor automatically advances to the next round.

This process continues until only a single creature remains.

Input File Format

The input file contains the comma-delineated information used to generate the creatures for the battle (see references). In summary, the fields are:

```
[id #],[name],[type 1],[type 2],[total],[hp],[attack],[defense],
[sp atk],[sp def],[speed]
```

Neither the creature object nor the application shall store the identification number, for it holds no meaning in our use. Storing the total of the remaining stats, which one may compute on demand, requires additional memory for every creature object. Instead, provide a means of supplying this total to a caller by calculating it on demand.

Combat

This application simulates combat through an equation. It uses the following characteristics to establish the creature's combat score:

```
Speed/2 * (attack + sp attack/2) + (def + sp def/2)
```

The winner of an engagement simply represents the opponent who maximizes this value. In the event of a tie, the creature with the the greatest *natural order* advances. If still tied, fall back to the creature's name alphabetically.

Round 0 (initial):

```
Vanquished: <empty>
Survivors: A, B, C, D, E, F
```

Round 1:

```
Pairs: (A,C),(B,D),(E,F)
Vanquished: A, B, E
Survivors: C, D, F
```

Round 2:

```
Pairs: (C,D)
Vanquished: A, B, E, C
Survivors: D, F
```

Round 3 (final):

```
Pairs: (D,F)
Vanquished: A, B, E, C, D
Survivors: F
```

CS 310: Data Structures
Spring 2017

Legendary Creatures

The input file includes a field identifying legendary creatures. Objects with this field set to true only enter the list of vanquished opponents after falling in a total of ***three*** rounds of combat. Thus, to defeat these opponents for good, all legendary creatures need to lose multiple fights.

They just keep coming back.

Experience

Victorious creatures grow in strength. After each battle the winner's `attack` and `defense` ratings both increase by half the opponent's total strength. The creatures quickly advance in power.

All *non-legendary* creatures who win three battles become ***heroic*** and, as such, shall then be considered ***legendary*** for vanquishing and experience point purposes. Heroic or promoted creatures should indicate as such when printed as a string.

Output File

Each time the application runs, it should override the previous contents of the output file (if it already exists). The output file shall include an introductory description of the round including statistics about how many creatures enter the battle, the current round number, and displaying a leaderboard listing the currently surviving, five strongest creatures.

It should then indicate the results of each battle in the round. One per line.

Thus, after one round of combat a file might contain:

```
Round 1:
Combatants: 401
– Leaders –
Creature F      [Type: Grass/Poison][HP: 1000]
Fire Elemental  [Type: Fire][HP: 700]
Water Elemental [Type: Water][HP: 355]
Phoenix         [Type: Fire/Flying][HP: 366]
Fremen          [Type: Sand/Water][HP: 800]

Begin!
Creature C defeats Creature A.
Creature D finally vanquishes legendary Creature B.
. . . omitted for clarity ...
Fremen defeats Worm.
Creature F automatically advances.
Round Complete!
```

Creature Object

Override the toString method in the Creature class created for this assignment such that it provides meaningful output when printed to a leaderboard. At a minimum, this output should display the creature's name and type.

This object must also implement the `Comparable` interface. Details appear in the requirements section of this document.

The creature object must include some means of leveling up or improving itself after a battle as indicated in the experience section.

Errors and Error Messages

If the user fails to provide sufficient or correct arguments to the application, the program shall notify the user of the correct usage directly to `System.out.` Any other errors or anomalies shall appear in the output file itself in clearly written terms.

Requirements

Each student shall work alone on a unique implementation, but one may discuss, in general terms, the approach with your colleagues. As an improved solution, however, consider office hours for either the TA team or your instructor.

- Generate a proper Maven Project Object Model (POM) file for the project with:

  Group Id: `edu.sdsu.cs`
  Artifact Id: `assign1`

  In order to execute the `jar` Maven produces outside the IDE, the POM must include a manifest. Your instructor will use this file to build and grade your assignment, so errors in this process incur substantial point deductions.

- Construct a class belonging to the package `edu.sdsu.cs` and meeting the operational requirements established in this document. The package requirement directly impacts the project's directory structure for use with Maven. Understanding it remains critical.
- Independently implement an ArrayList by extending the java.util.AbstractList
- Independently create the driver application using the ArrayList.

Array List

Rather than updating the size to precisely match the current contents, the ArrayList (or Vector) continually maintains a block of extra, unused space. One may think of it as padding. When the data structure's load passes an established threshold the ArrayList automatically creates a new, larger array and copies over the current contents. Thus, the ArrayList ensures it maintains an open space for any new items at all times. That is, at the conclusion of every addition and deletion, the ArrayList guarantees it possesses sufficient,

allocated storage to support the next add operation in constant time. Consequently, adding to a ArrayList typically takes constant time (although certain additions will take more time due to the reallocation). Essentially, the ArrayList manages an array behind the scenes.

- Increase size: Capacity at or above 80%
- Decrease size: Capacity at or under 30%
- New array size: Twice current contents

Private Methods

Internal, private methods help keep code clean and organized. Their use, consequently, remains unrestricted. Create as many as necessary to modularize your code.

Interface

The ArrayList implements the List interface which, in turn, extends the Collection interface. Consequently, every ArrayList is both a List and a Collection, and polymorphism allows developers to use Vectors wherever the code demands either interface.

The List interface facilitates getting and setting via index value -- functionality the Collection interface lacks.

Extending Abstract Class

By extending the AbstractList when creating the new ArrayList, students inherit a substantial, powerful code base. In fact, to implement a modifiable list, students need only implement five methods in the child class:

- `get(int)`
- `size()`
- `set(int, E)`
- `add(int, E)`
- `remove(int)`

Required Constructors

Best practices dictate all Collection objects include two constructors: a default, zero-argument constructor and a single argument constructor which takes a Collection object. The Collection interface defines the required behavior, so read the documentation and ensure your solution is compliant. Thus, you must implement two constructors:

```
ArrayList()
ArrayList(Collection<E> c)
```

Comparable

Each creature object possesses its own relative strength representing the sum of its HP, Attack, Defense, SP Atk, SP Def, and Speed. These represent intrinsic characteristics associated with the creature itself, so

using this total to establish the creature's *natural order* seems logical. Thus, the creature class created for this assignment must implement the `Comparable` interface using the sum of these statistics.

For three creatures, if creature A's total is 300, creature B's total is 500, and creature C's total is 50, then then natural order is: C, A, B

Thus, `A.compareTo(B)` shall be *negative*.

Sorting the List

To sort this array based upon the creature's different abilities, use a Comparator and Collections.sort. Specifically:

`sort``(List<T> list, Comparator<? super T> c)`
Sorts the specified list according to the order induced by the specified comparator.

Additional Classes and Interfaces

Students will need multiple classes when satisfying the application requirements of this assignment. When doing so, developers must remember to design *to an interface, not an implementation*. Thus, students will likely need to create at least one interface files capturing necessary public methods.

Data Deliverables

Each student shall submit a single `.zip` file named in the following format:
`LastName_FirstName``_cs310s17_assign1.zip`

Thus, were your instructor turning in the assignment, it would read:
`Healey_Shawn_cs310s17_assign1.zip`

The archive itself shall *only* include the `POM` file created for this project and the `src` directory (and any of its subdirectories). The build process itself generates the contents of the `target` directory, so including it, or any `class` files, in the archive only wastes bandwidth.

Every Independent Java file generated for this assignment must include a header line identifying the student by name and edoras class account number in each file. For the driver, group Java files, students must identify all members of the group and their class accounts in every file.

Grading

Pay close attention to the deliverable file requirements in this exercise – particularly the packaging of your POM file and source directories. An automated script shall extract the contents of the .zip archive you provide, step into the resultant directory, package your program with Maven, and then execute it with a specific text file. Programs submitted which fail to compile, or fail the Maven packaging process, receive *zero* credit.

The `ArrayList` serves a greater purpose. Its generic nature makes it useful in other applications. Specifically, students will use it again in the next homework assignment.

CS 310: Data Structures
Spring 2017

Place `ArrayList` in the package: **`edu.sdsu.cs.datastructures`**

Optionally, students may place `CsvReader` in the package: **`edu.sdsu.cs.util`**

All other application-specific and driver files you create for this project must belong to the broader package: **`edu.sdsu.cs`**

Points Breakdown

Data structure work counts for the majority of the points available in this assignment, seventy-five percent, with the student's successful implementation of an ArrayList. The remaining twenty-five percent of the points on this assignment derive from the game application driver and its required classes. Focus on making the data structure work!

Additional Notes

Maven will automatically generate the correct name for the jar file for this assignment when correctly configured.

The Java package requirement defines the directories wherein your files reside. An incorrect package statement, or misplaced file, will generate Maven packaging errors. Make certain you understand this process.

During development, group members may share output files to verify every team member's data structures produces identical output.

References

Javadoc

https://docs.oracle.com/javase/6/docs/api/java/util/List.html

https://docs.oracle.com/javase/6/docs/api/java/util/Collection.html

https://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html

https://www.kaggle.com/abcsds/pokemon