Project 5: Wearable Devices

1 Task

1.1 Scenario

Binary Search Trees and their kin can be used in many different contexts. We will use them to create a TreeMap that can quickly add data and do in-order data retrievals.

You are given a starter data set of wearable devices, falling into various categories like health, fitness, lifestyle, etc. The sample file (Wearables.txt) contains over five hundred such devices.

Wearables.txt, a simple text file, contains:

- An integer on the first line, a count of how many items are in the file.
- A header line, describing the contents of each attribute given per item. This is only for your information; you do not need your code to utilize it in any way.
- Wearable items, one per line, with attribute data delimited by "@".

1.2 Indexes

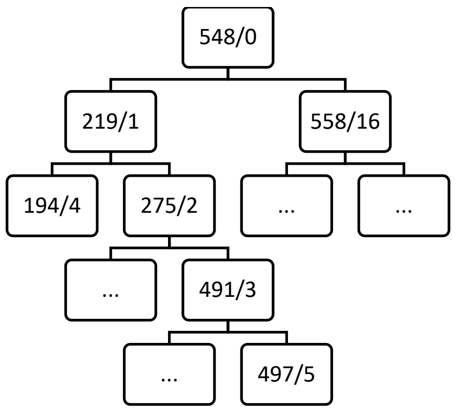
1.2.1 Data Used

We want index-based data retrieval using three different indexes:

- Ranking—the devices are ranked by consumers, from 1 to *n*. These rankings *will always be unique* (you don't have to check for that; just assume it).
- Price—device prices vary greatly, and *aren't unique*. If the price isn't known, a price of -99.99 is provided in the file.
- Company Name—some companies produce many items; some, only one or two (i.e., these *are not unique*). We want an easy way to find and display those.

1.2.2 Building the Indexes

You will implement TreeMap-based indexes. Each node must carry both the *tree* data (rank, for example) and *position* data (the position at which that Wearable exists in the master array). For example, the first item in the file is ranked 548, and resides at index 0 in the array. Example of the top of the ranking index:



But because we're handling *duplicates*, we'll need to get a bit more creative with our indexes/nodes.

1.2.3 Services

Indexes will provide one basic service: they can retrieve an array of positions at which that data is found. For example, using the ranking index above, the data retrieved would be…

4	1	2	3	5	0	16

2 WearableManager

The WearableManager class serves as the client interface; client code is not allowed to deal directly with the indexes and doesn't even need to know they exist. This class should offer services that allow clients to generate in-order data (returned as an array of positions), and to generate a CSV file, when asked, using a specified list of positions. Here are some of the public methods you should have:

- `Wearable getWearableAtIndex(int index)`
- `int[] getRankingPositionData()`
- `int[] getPricePositionData()`
- `int[] getCoNamePositionData()`
- `boolean generateCsv(int[] positions, String filename)`

3 Design Documentation

Don't underestimate the design requirements of this project; they are substantial. Before you code, create appropriate design documentation and obtain feedback. Update designs per feedback, then use them during the rest of the development process, and submit them as part of your project. For this project, this you'll need a UML Class Diagram and UML Object Diagram.

4 Code Implementation

4.1 Building Blocks You'll Need

- TreeMaps (BST relatives)
- Generics[1]
- Recursion

4.2 Requirements

Here are requirements for your coding project:

- Each Wearable should become its own object. Provide accessors for everything. You may skip mutators, assuming one full-bodied constructor is sufficient for now.
- WearableManager constructor (with no parameters) manages a single array of Wearable objects (created while reading the file) and calls for the creation of the various indexes described below, by adding to the indexes during file read. Don't proliferate FileNotFoundException throws from the reading method; if it fails, just leave the array empty.
- In a Main class, after instantiating the WearableManager object, write some code that demonstrates the data retrieval and CSV capabilities. Exercise each index. Also show off extra credit, if done.

[1] As always, ensure you have no raw types or unchecked warnings; and fix things rather than suppressing, in all but unavoidable cases where these have expressly been permitted by your instructor.

- Write a *generic* index class for indexes. When implementing data retrieval in the indexes, return an array with the results, as arrays are the most standard data structure for communicating data. Use no ArrayList or other data structure unless you're doing extra credit.
- You'll need two types of nodes. Do not create separate classes of nodes for each type of data indexed, however; instead, create generic nodes that allow any object data type, including those we need now or might need in the future. You will need a second node type that should become apparent as you design, and as we discuss designs together.
- Use no public data except in nodes; we understand that navigation through nodes requires public access. Unless nodes are needed by outside classes (hint: they usually aren't), create them within the tree class in which they are used.
- Implement *no sorting algorithms* anywhere in this project.

4.3 CSV Files

- CSV files are simple text files, with one record per line and commas between fields.
- Strings containing dangerous characters (especially commas and quotation marks) should themselves be surrounded by quotation marks, with interior quotation marks doubled[2].
- Numeric fields should not be quoted.
- The first line of the file should contain a header, so the end user knows what each field represents.
- Check output in Excel, for verification purposes.
- Important: extra spaces (for visual separation) *must not be added after field-separating commas*, or Excel may do a poor job breaking apart fields.

Sample CSV output, when passed an array of positions from the ranking index (line wraps only shown for space consideration):

```
Ranking,Name,Price,Body Location,Category,Company Name,Company URL,Company
      Location,Company City,Company US State,Company Country
1,"Cogito Classic - Green Velvet",179.95,"Wrist","Lifestyle","Cogito",
      "http://cogitowatch.com/","Hong Kong, China","Hong Kong","Non-US","China"
```

4.4 Additional Requirements

Important: there will be additional requirements that we will determine together after you have some time to ponder design approaches the TreeMap and the node types it will require. You'll need to include those designs in your project.

4.5 Style

Follow the Course Style Guide. You'll lose points on every assignment if you fail to do so.

5 Testing

Take a break from the usual JUnit unit tests. If you get the whole thing working properly, we'll assume you're darn close on most of these objects and their methods.

[2] To make coding easier, consider surrounding *all* strings with quotation marks.

6 Submitting Your Work

Follow the course's Submission Guide when submitting your project.

7 Hints

- The indexes shouldn't know anything about any Wearable object or the WearableManager class; they have a narrow, well-defined job, and that's to maintain ordered tree data and return position data.
- Before starting this project, incorporate feedback from your designs; you should start building on a firm foundation.
- To create generic indexes, you will need a hint provided in your textbook regarding Comparable. If you fail to recognize and implement this, you'll run into issues that will prove insurmountable.
- Apart from the issue discussed in the previous bullet point, don't repeat code; be smart and leverage existing code.

8 Extra Credit

8.1 Ranges

In addition to allowing only index-based list generation for the entire array of Wearables, provide an overloaded method which allows the client to specify an inclusive *range* of values. For example, the client might want a list of only the top 20 items by rank (minimum is one, maximum is 20). For prices, they might want items priced from $0.01 to $99.99. For companies, they might want companies "Archos" through "Casio". You may use an ArrayList when building the list of positions to return; turn it into an array before returning it, however. Implement this code in the index, not in WearableManager.

8.2 Balancing

Implement a periodic balancing algorithm so that the rating BST (only) becomes height balanced. You do not need to incorporate this into the index's add method, just write it and make sure it works.

(continues…)

9 Grading Matrix and Points Values

Area	Value	Evaluation
Design docs	10%	Did you submit initial design documents, and were they in reasonable shape to begin coding? Did you submit final design documents, and were they a good representation of the final version of the project?
Wearable class	5%	Was the Wearable class correctly created, with all file columns loaded and available via get methods?
WearableManager class	10%	Does the WearableManager class provide all client-facing methods? Was it well written?
Generic index classes	25%	Were generic indexes created, providing the desired functionality? Were they scoured for unchecked warnings, and issues fixed?
Node classes	10%	Were both node classes built correctly and well coded?
Data retrieval	15%	Was data retrieval well implemented, returning the required indexes via arrays?
CSV file	10%	Were CSV files created properly from their position arrays, and are they readable by Excel?
Demo code	5%	Does Main provide a good demo of all the available functionality, using each of the three indexes?
Style/internal documentation	10%	Was the -Xdoclint run clean? Did you use JavaDoc notation, and use it properly? Were other elements of style (including the Style Guide) followed?
Extra Credit: ranges	2%	Were range filters implemented for all data types?
Extra Credit: rebalance	2%	Was a working periodic balance algorithm provided?
Total	**104%**	

Code that does not compile or run will be given 50% credit and returned for rework and resubmission.