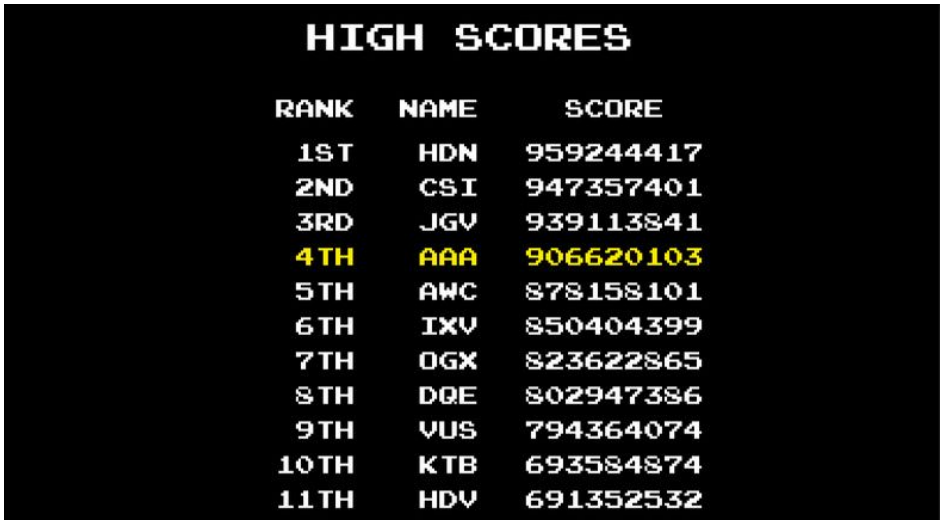
Computer Laboratory 10

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development



1 Essential information

- This assignment is due Tuesday November 22nd at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.

2 Introduction

This lab begins our final topic of the semester: Data structures. We will do this by designing a data structure specifically for classic video-game "top-10" lists. These are data structures which maintain, in sorted order, the top-10 (or any number N) Game Scores that they've seen. We will do this in two parts. First we will build a simple GameScore object – this will

help us practice a few essential java methods, in particular, the compareTo method. Then we will build a LeaderBoard class. Despite it's name, we'll be designing the LeaderBoard class in a broad and re-usable way which will let us use it with any number of data types and list-sizes.

From an educational standpoint, in this lab you will:

- Learn more about the `Comparable<T>` interface
- Practice implementing a comparable data types' `compareTo` method – including the somewhat tricky thinking involved!
- Learn how we can take a specific idea (video game leaderboard) and implement it as a broader, more generally re-usable class
- Learn more about generics
- Program a class using generics
- Apply parts of algorithm's you've learned in the past into new problems
- Practice writing code using the `compareTo` method instead of `< and >` for comparisons

3 Files

This lab will involve the following files.

- `GameScore.java` – (YOU MAKE) a class to represent a single "game score"
- `GameScoreTester.java` – (PROVIDED) tester for previous
- `LeaderBoard.java` – (PROVIDED, NEEDS COMPLETING) a class to represent a top-N list over **ANY comparable datatype**
- `LeaderBoardTester.java` – (PROVIDED) tester for previous

4 GameScore

The game score class will represent a single item in a classic video-game score board like the one seen on page 1 of this document.

4.1 Basic Variables and behavior

The first thing you'll want to do is make a GameScore class. This should have 3 private variables:

- the score (a double)
- The name for this record (a String)
- a boolean indicating if this score was earned on hard mode (true) or easy mode (false)

Based on these variables implement the following functions:

- `public GameScore(String name, double score, boolean hardMode)` – a constructor for this class.

- `public String getName()` – getter for the name property
- `public double getScore()` – getter for the score property
- `public boolean isHardMode()` – getter for the hardMode flag – returns true if the game was hardmode.
- `public String toString()`
  - Like other String based situations in this class, we do expect letter-to-letter matches with the output seen in the testers.
  - In this case, the format should be the name, followed by a single space, then the score (no rounding needs to be applied). If the score was made in hardmode then an asterisk should be appended to the end of the toString.
  - Like other toString methods – make sure to review the examples in the tester files for full details.
- `public boolean equals(Object o)`
  - The parameter (Object) is not a typo here – make sure you get that correct.
  - If you can't remember how to do this, remember that we did a few examples in lecture – check your notes for how we handled it then!
  - A GameScore should not be equal to any other data type
  - Two GameScore objects should only be equal if and only if they have equal names, the same score, and the same difficulty (both are hard mode or both are easy mode)
    - To be clear – even if two GameScores have the same name and score, if one is from hard mode and the other is not from hard mode – they should not be equal.
  - **Warning – the provided tests are not very good for this function** You should not rely only on the provided tests to know if your code is correct. I recommend adding a few extra cases to make sure your code returns false when provided different names, different scores, and different values for the hardmode boolean.

4.2 Make it Comparable

Your final task with this class is to make it Comparable. To begin, update the class declaration as follows:

```
public class GameScore implements Comparable<GameScore>
```

This is required – and skipping it will not allow the class to work with other java code based on sorting. Normally we wouldn't give you this exact code, but seeing as how this is your first time with both interfaces in general, and the Comparable interface specifically, we've given you this code directly.

Secondly, implement the `public int compareTo(GameScore other)` function. Remember, this function's job is to compare the current GameScore object (`this`) to the parameter object (`other`)

- if *this* > *other* return any positive integer
- if *this* < *other* return any negative integer
- if *this* == *other* return 0 (NOTE – it's possible for two items to be equal for sorting purposes, but not equal in general.)

For Game Scores we sort based on the hardmode value (all hardmode scores are bigger than all easy mode scores) and then the score itself (a bigger score is bigger than a smaller score). Note – we ignore the name when comparing GameScore objects. So for a few examples:

- `DAK 100 < DJK 100.01` (bigger score same difficulty)
- `JAK 1* > DAK 10000` (hard mode is always bigger than easy mode)

When coding – make sure to think about both bigger-than and smaller-than cases, a common mistake is to code up (for example) the situation where `this` is a hardmode score and `other` is not, but forget to code the reverse situation.

5 LeaderBoard

To help you get started with the LeaderBoard class, and to help resolve a few hard-parts (which we likely wouldn't discuss in lecture before lab) we've given you a template LeaderBoard.java file. You should begin by downloading this class.

The design of the LeaderBoard class is based originally on the idea of a videogame leaderboard: store a top-10 list of elements in order. However, following good computer-science practice, we've made the class able to be re-used in new situations by making it more generic. We've done this in two ways:

- The constructor to this class takes a "size" parameter allowing it to be used for any size of top-N list (top-5, top-10, top-100, etc.)
- Using a particularly tricky form of java's "generics" feature we will make the class work for any data type implementing the Comparable interface. This still allows us to use it for GameScore, but we can also use the class for strings, numbers, and future comparable classes we haven't seen yet!

It will be useful, and important, for you to keep this more generic (any size, any type of data) vision in-mind as you work on this class.

The provided template code has the following already-programmed:

- A type parameter T. This type parameter describes the actual type of data being stored. Anywhere in the code you can (and should) use `T` as if it was the name of the class being stored. We've provided you *most* of the places where you will need this already – but you might need to make local variables of type T in some functions.
- A *type bound* for type T that says it implements comparable with itself. This is particularly tricky java code, but what it means in practice is **you can assume all data of type T has the compareTo method**.

- A T array which holds the scores.
  - This is the only instance variable this class should need. Do not add more instance variables.
  - The scores array should not need to be recreated – it's size does not change, and reassigning it to a new array is over-complicated for this problem.
  - The scores array should always be kept sorted LARGE to SMALL (so top-scores would be in earlier positions) **NOTE** You will need to do work to make this property stay true.
- A constructor. The constructor takes a size and a default value, it then creates the data array and fills it with the default value. Do not modify this.
- A toString method. Do not modify it.
- Several empty method definitions for functions you will need to write.

You should begin by writing the three easy functions:

- `public int getSize()` get the size of the leaderboard.
- `public T highScore()` get the largest value in the leaderboard
- `public T lowScore()` get the smallest value in the leaderboard

All three of these functions should be easy 1-line functions which run in time $O(1)$. If you're struggling to see how to code these you should think carefully about the scores array, and how that array is organized.

5.1 The Hard function: add

The final method on this class is also the hard-one: the add method. This method is used to attempt to add a new score to the leaderboard. The two key challenges: 1) not every new element should be added to the leaderboard – we only update the leaderboard if given a new item bigger than the current smallest item. 2) If we do modify the leaderboard, we need to do so in a way that keeps the scores array sorted big-to-small.

A few examples may help: Let's say we had a Leaderboard of Integers:

```
1. 100
2. 58
3. 42
4. 15
5. 8
```

If we were to call add(7) or add(8) we would expect no change (as these numbers are too small to make it on the top-5 list) However, if we were to call add(30) we would expect the following:

```
1. 100
2. 58
```

```
3. 42
4. 30
5. 15
```

8 (the old smallest) would be removed and 30 would be inserted in the correct place in sorted order (bumping 15 down to position 5) If we were to then do add(58) we would expect the following:

```
1. 100
2. 58
3. 58
4. 42
5. 30
```

The new 58 is added at position 3 (under the old 58), with 42 and 30 shifted down and 15 (the old smallest) removed.

This last behavior is worth one more example, this time using GameScores: If we began with:

```
1. DAK 103
2. DJK 86
3. JAX 73
4. JSK 72
5. TNA 56
```

and then added "JAX 86" the new result would be:

```
1. DAK 103
2. DJK 86
3. JAX 86
4. JAX 73
5. JSK 72
```

As shown here – if we add something that is equal (according to the compareTo method anyways) an existing element it should show up UNDER the element it's equal to. (I.E. in this case the new element "JAX 86" shows up after the previously-set score "DJK 86")

This task can be surprisingly hard algorithmically, and coming up with an algorithm for it isn't the focus of this lab, therefore we've provided you with the following pseudocode to get started. You will still be in charge of translating this pseudocode into working java code, including correctly using the compareTo method to compare elements.

1. First, check if the element is larger than the current smallest element. If it is not, then the function does not need to do anything and can simply return.
2. If the new item is larger than the current smallest element in scores – replace the smallest element with this one. The smallest element is the one that would have been removed anyways

3. Now all we need to do is "swap" the last element to the left until the sorting is correct. This is essentially the same as we saw in insertion sort – and the same pseudocode can be used:
    - use a variable to track which index our new-element is currently in
    - while the new element is not in position 0, and the element left of the new element is smaller than the new element: swap the new element one to the left.

6 Submission

For this lab you need to turn in:

- `GameScore.java`
- `LeaderBoard.java`

Grading on this assignment will follow the given general rubric:

- 50 points autograder This is split 20-30 between GameScore and LeaderBoard, with GameScore tests focusing on the compareTo method and LeaderBoard tests focusing on the add method across a range of different data types.
- 10 points code style:
    - Your name (and the name of any collaborators) should be in a comment at the top of the file
    - Correctly formatted JavaDoc (this is the `/** ... */` comment blocks placed before each function that describes the function
        - We will consider this optional for constructors and 1-line functions.
        - This requirement stands for any function longer than 1 line.
    - You should also have a JavaDoc before each class that describes the class as a whole. You can look at the provided code from lab06 to see examples of this
    - Good variable names
    - Consistent tabbing (while java does not require this the same way python did – we **do** require it. Reading and grading poorly tabbed code is a painful experience)
    - Outside of these explicit requirements, as normal, we will generally be looking for (and giving feedback on) anything that makes the code harder/more painful to read, and will take away points if it's particularly bad.
- 20 points for manual review of GameScore
- 20 points for manual review of LeaderBoard

REMEMBER manual review points serve as both partial credit for code that fails tests, and lost-points for code that passes tests but is, nonetheless, incorrect.