Project 0 Voting Machine

Introduction

Like many aspects of our lives that we view transparently, when an election comes and we decide to vote, we have the luxury of going to our local polling place — or even just submitting an absentee ballot — casting our vote, and not thinking too much about the work that goes into managing the election. But elections pose a significant logistical challenge. Ballots must be designed and printed. Voter registration lists must be maintained. Only registered voters can be allowed to vote, and each must be allowed to vote only once. It must not be possible to figure out which voters placed which votes. Ultimately, votes must be counted and results disseminated. As with many logistical challenges, the burden of handling elections is increasingly being placed on computers; these kinds of logistics are among the things that computers handle best, though automated solutions present their own problems when not designed and built carefully.

In recent years, various on-paper voting mechanisms have been replaced with electronic voting machines. This project will let you explore the design and implementation of software for a highly simplified voting machine that allows users to cast votes for one race (say, Mayor of Simpleton) and counts the number of votes for each candidate. I should be careful to point out that the design of a viable real-world voting machine requires a fair amount of care; it is important to note that our voting machine lacks a number of important features — not the least of which is some form of security mechanism — that are essential in a realistic one.

Assignment

Your program is the software for a simplified voting machine with a graphical user interface (GUI). When started up, it reads information about a single race from an input file, then displays a ballot. The ballot can be used to cast votes indiscriminately — there's nothing in our program that forces users to "log in" or otherwise identify themselves before casting a vote — with the votes being counted underneath the covers, but not displayed to the user while the voting is in progress. After closing the ballot window, the program saves the results of the election into an output file, formatted in a particular way.

Class structure

You'll be writing code in four classes:

- **Candidate**, each object of which represents one candidate on a ballot, consisting of a name, a party affiliation.
- **Ballot**, each object of which represents a single ballot, consisting of an ArrayList of Candidates, as well as the name of the office that is up for a vote.
- **BallotReader**, which consists of a single static method that reads a ballot from an input file.

- **ResultWriter**, which consists of a single static method that writes the election results into an output file in a format specified in detail later in this write-up.

These classes might need to have additional member data and methods as needed to accomplish the task of the project.

Each of the methods in these four classes are commented but not implemented; your job will be to fill in the implementation of all of these methods. You do not need to read, understand, or modify any of the other code. (You're welcome to read through it if you'd like, but you are not permitted to modify it.)
I suggest working on these classes in the order listed above. Once you've implemented Candidate and Ballot, the program should compile and run, though the ballot will always consist of the same candidates — because the provided BallotReader just constructs a hard-coded ballot instead of reading one from an input file — and the program will not write any output — because the provided ResultWriter does nothing.

Input file format

The program reads ballot information from an input file, an example of which follows:

```
James Beard Award
3
Nina Compton;Compere Lapin
Alon Shaya;Saba
Emeril Lagasse;Emeril’s
```

The first line of the input file specifies the name of the office that is being voted upon. The next line consists of a number that specifies how many candidates are running for the office. If that number is *n*, the next *n* lines each specify a candidate, with the candidate's name appearing on the line first, followed by a semicolon, and followed by the candidate's party affiliation.
You may assume that the input file will always be properly formatted according to this specification. If it's not, it's fine for your program to misbehave or even crash; we will only test your program with valid input files. Scanners are capable of reading one line of input at a time from any input source, including a file. I suggest always reading the file one line at a time, then processing the line.

Output file format

In this program, you won't just be printing unformatted text to the console using System.out.println; you'll instead be writing a nicely formatted output file that indicates the results of the election. (Unrealistic as it may be in any but the most local of elections, we'll assume that there is only one voting machine being used by all voters.) An example of the output format follows:

```
RESULTS - James Beard Award
---------------------------
Nina Compton - Compere Lapin        102
Alon Shaya - Saba                   105
Emeril Lagasse - Emeril’s            97

WINNER: Alon Shaya - Saba
```

The details of the output format are:

- The first line begins with the word "RESULTS", followed by a space, a dash, another space, and then name of the office that users voted for.
- A line of dashes, where there exactly as many dashes as there are characters on the first line.
- For each candidate on the ballot (you can list them in the same order they appear on the ballot; it's not necessary to sort them by the number of votes), with each candidate appearing on a separate line:
    - The name of the candidate, a space, a dash, a space, and the candidate's party affiliation. Collectively, we'll call this the candidate's *tag*.
    - The number of votes received by that candidate.
    - The candidate names should be left-justified in the first column of the output.
    - The numbers of votes should be right-justified in the last column of the output.
    - To calculate how many columns your output should have total, first figure out the number of characters in the longest tag for any candidate. (In the example above, "Nina Compton - Compere Lapin" is the longest tag.) Then, add 12 to the length of the longest tag.
- A blank line appears after all of the candidates are listed with their vote totals.
- Finally, a line that indicates the winner, with the word "WINNER", a colon, a space, and the winning candidate's tag.
    - In the event of a tie, this line should read "NO WINNER" instead.

When you want to write formatted output, with left- and/or right-justification within certain numbers of characters, the **String.format()** method helps; it knows how to take data of various types, format it according to your specifications, and return it to you as a String. The first parameter you pass to it is called a *format string*, which is used to tell it how you'd like the data to be formatted, with placeholders for the data. Subsequent parameters specify the data that will replace the placeholders.

Additional documentation

You will find some or all of the following classes and methods in the Java library useful. Sometimes, you'll find that you need to know details about how they work that we haven't yet talked about in class. When you need more information about them, see the Java API documentation, which describes all of the classes and methods in the Java library. (There is obviously much more documentation that you'll ever have the time to read; the trick with documentation, when there's as much of it as there is in the Java API, is to know what you're looking for and focus on that, rather than trying to read everything.)

- ArrayList
- FileReader
- FileWriter
- IOException
- PrintWriter
- Scanner
- String.format()
- String.length()

Note that you may well be able to finish this project without using everything on the list above. Java has a large, industrial-strength library, which means that there are often many ways to accomplish the same goal. The ones listed above are the ones that present the simplest path to a solution, but your prior experience may have turned you on to different choices; for the most part, that's fine.

Starting point

To give you a more realistic context for your project, the project has a complete GUI provided for you; you will not need to read, understand, or modify that code in any way. (Still, you can check it out if you're interested. Even a cursory glance will demonstrate that coding up even simple user interfaces like this one can be a complicated process!)

You also have a skeleton implementation of the "model," the classes that implement the underlying engine for the program, which handles tasks like reading the input file, writing the output file, and counting the votes. You are not permitted to modify the signatures of the methods provided in these classes. This restriction is motivated by one practical need: the GUI code expects these methods to have the same names, take the same kinds of parameters, and behave (outwardly) exactly as specified. If you alter the signatures of these methods, the GUI code will no longer compile or work. (Note that you can feel free to add new methods; you just can't change the signatures of the methods provided.)

The starting point is available in a zip archive on the class website.

A word of warning about contracts

As you work on this project, you make an agreement with the developer of the starter code, a *contract*, of sorts. You're provided the GUI in its entirety; in return, you're required to write your code according to the provided specification. If your code deviates from the specification — say, a method returns null when it's not supposed to — it's entirely possible that the GUI will behave unpredictably or even crash.

(This illustrates an important point about building large-scale software: When many people work together on a large project, it's important that they agree on how their parts will communicate with one another. In this case, you are collaborating with GUI (front-end) developer — theGUI depends on your engine — and the agreement is that your engine conforms to the specification provided. Naturally, the larger the software and the more people and pieces involved, the more important these kinds of agreements become.)

Code quality

Code should follow OO approach. Each class should correctly use public and private specifiers, getter and setter methods. As always, code should be clean and well-commented. Portion of the grade will be assigned based on code quality.

Version control

As you work on the project, remember to frequently save your work, and keep several copies of it under different names. Remember to keep a copy of your most recent work on the cloud storage (in your email, Google Drive, or Dropbox). This way even if you lose your computer, you will be able to recover the project code. There will be no adjustments or extensions for projects that were accidentally lost.

Javadoc

This technical specification will show the complete design work for your software. It is only intended to be read by programmers and managers (not the user).

Write a technical specification for the software including:

- A problem statement describing the assignment in your own words; put this problem statement at the top of your PerfectCandidate class in a Javadoc comment. What does the software do from a user's point of view?
- A Javadoc comment explaining the purpose of each other class you write; put this comment at the top of the class.
- A Javadoc comment on each method that you write. This comment will stating its purpose, inputs, outputs, any methods it calls, and any (high level) algorithms it uses to accomplish the task.
- A Javadoc comment on all instance fields in your classes. The comment should state what the field is used for.

User manual

Your user manual describes how the program should be used and what are the valid inputs from a user. They tell the user what operating system it runs on, how to launch and quit the software, and what buttons to hit to run it. Assume your user can be told to run Eclipse but does not know Java so they are just clicking on the buttons you tell them to hit.

Unless you find a nicer way to accomplish this, it can be a text file. (Yes, you can use doc, docx, odf, rtf, or pdf instead of txt if you want to use a fancier editor).

Testing

Test your classes and methods well as (and before) you write them. Some tasks can be tested with automated testing, and you should write JUnit tests for them. Some tasks can be hard to automate with JUnit (e.g. testing GUI) and you should describe your tests as test scenarios. Your JUnit tests and test scenarios are one of the project deliverables.

Normal, error, and boundary cases

There are three kinds of test cases that I'd like you to focus your attention on:

- *Normal cases.* A normal case is a test case that exercises some ordinary function of the program using valid input. For example, depositing $20 into an ATM constitutes a normal case, since it is legal. We would expect to see the account's balance increase by $20.
- *Error cases.* An error case is a test case that verifies that the program correctly handles invalid input or other error conditions. For example, withdrawing $30 from an account with only $20 in it constitutes an error case, since the program will not perform normally (i.e., withdraw the money) in this case.

- *Boundary cases.* Sometimes, programs work correctly in normal cases as well as most error cases, but misbehave on the *boundary* between the normal and error cases. For example, it is useful in an ATM to test what happens when you attempt to withdraw the exact balance of an account (leaving $0 remaining in it). Test cases such as these often highlight a program's most serious flaws, when you've made a mistake such as using "<" where you should have used "<=".

Extra credit extensions

Extra credit is available for extensions, creativity and innovation brought into the project:

- Keep track and display additional information about each Candidate.
- Have a help button to display user manual, add fancy graphics or effects to the UI.
- Add another type of issues on the ballot: in real life, ballots usually contain candidates and issues that citizens vote yes/no on ("Do you support replacing Panera Bread with Dat Dog in LBC?").
- Go wild and implement any crazy feature that you'd like your voting machine to have.

To receive extra credit, your added feature should be fully working and explained in an auxiliary readme document supplied with the project files. *(Extra credit will only be given if the required main functionality is implemented correctly. Don't attempt extra credit if the basic Voting Machine is not working well, focus on fixing it first).*

Procedures

Group work and academic integrity policies apply to this project. You can work solo or in groups of two. No larger groups are allowed. No copying from external source is allowed, except for the code from our Zybook, and from our class notes. If you need to use external libraries or external code snippets for extra credit parts, get in touch with me and I will help you to adhere to the honor code.

Deliverables

Minimum functionality: Complete UML diagram describing the project. Prototype application code containing at minimum the implementation of Candidate and Ballot classes. JUnit tests for implemented methods Candidate and Ballot methods. Javadoc for all implemented functionality. User manual for all implemented functionality.

Complete functionality: Fully working application code, test plan and JUnit tests, Javadoc, user manual, revised UML.

Submission

Create a project folder with 5 subfolders named exactly as specified: UML, JavaDoc, Source Code, Executable JAR, Documents. Place your files in the subfolders. Submit a .zip archive of the main project folder to Canvas.

Acknowledgement

This project is an adapted from assignment by Alex Thorton submitted on EngageCSEdu.