Homework 6 - Building a Voting Machine

Due Dec 10, 2019 by 6pm **Points** 0

Before you begin, please make sure you are familiar with the guidelines discussed on the Expectations on Homework page and the "Homework Assignments" section of the CS 2102 FAQ Page.

Assignment Goals

- To be able to write clean, encapsulated code
- To implement and test code that uses exceptions
- To gain experience with Java's `HashMap` class

Reminders

- **Please use the default package in your Java project.** There will be a penalty for using a named package.
- Please include a Javadoc statemnt above each method. There will be a penalty for forgetting your Javadoc statements.
- In your test cases, please use descriptive names for your test case methods and/or comments above each test case explaining what the method tests. This will help the TAs and SAs in grading. There will be a penalty for forgetting this step.

Overview

You are building software to conduct an election. In this election, voters will state their top three choices (rather than a single choice). There are many different ways to determine winners in such preference-based elections. Unfortunately, the people conducting the election haven't yet decided how they will count the votes. Therefore, they want you to build a voting system that will record votes but will also let them experiment with different ways to determine the winner.

This starter file **(https://canvas.wpi.edu/courses/16466/files/2217211/download?download_frd=1)** gives you a very rudimentary voting program: it presents a screen that asks a voter to enter the name to vote for, stores that choice, and counts votes for a given candidate. Your job is to

- extend the system to store three votes per voter and
- provide two different methods for determining who wins the election (described below),
- while using exceptions to detect malformed votes and duplicate names on the ballot
- and producing clean, encapsulated code.

Your solution to this homework should NOT include the `countVotes()` method from the starter file or the current `votes` variable of type `LinkedList<String>`. You will still have a votes variable (or something like it), but with a different type that can handle three ranked choices per voter.

If you want to test your Java skills, spend a little time thinking about how you might do this before reading how we break down this assignment into steps below.

Problems

Warning: Do this assignment in stages, making sure that your program runs properly after each stage. If you try to make all of the changes at once, you'll spend far longer than needed on this assignment. Build up to your solution step by step. You can get more credit for fully working early steps than for having most of the steps only partly working.

Once you have all of the methods and exceptions in place, you can use this **CompileCheck (https://canvas.wpi.edu/courses/16466/files/2217210/download?download_frd=1)** file to check for naming errors, etc. ***Please make sure you use this Compile Checker--not the one described above--to check your finished code.***

1. Record three choices per voter

Edit the starter code to use hash tables to store the number of first, second, and third choice votes for each candidate on a ballot.

Once you have your hash tables, add the following two methods:

- a `processVote()` method that takes three strings (for the first, second, and third choices, respectively) and returns void. This method stores a single voter's choices in your data structure. (Calling this method corresponds to someone voting in the election.)
- an `addCandidate()` method that takes one string (for the name of a candidate) and adds the candidate to the ballot, returning void. If the named candidate was already on the ballot, throw a `CandidateExistsException` whose constructor takes the name as its only argument. If necessary, you can also do other work in this method to set up your data structures to know about the new candidate. (Just make sure that you don't change the signature of addCandidate(). It should consume a String and produce void.)

Use either Java's built-in `equals()` method (on Strings) or `contains()` method (on LinkedLists) to check whether names are the same. This will treat names with different capitalization as different (such as "gompei" and "Gompei"). Our tests against your code will not try to trick you with this; we will use distinct names for all candidates and attempted votes to be on the safe side.

You do not need any special handling around empty strings passed as names. Our tests will not use empty strings as names, and our solution will treat the empty string as a name like any other. Similarly, you may assume that null will not be given as a name.

2. Use exceptions to report malformed votes

A vote is only valid if the voter enters three different names, each of which is a candidate on the ballot. Modify your `processVote()` method to check these criteria, using exceptions (named below) to report problems and require the voter to vote again if their vote is not valid.

Create Exception subclasses named `DuplicateVotesException` and `UnknownCandidateException`. The constructor for each method should take a String for the name of the candidate that was voted for multiple times or not on the ballot, respectively. In each Exception, make the name field private and write a public getter to get the name value. If the user specifies multiple names not on the ballot, report the first one (in order of the arguments to `processVote()`).

For purposes of autograding, we need to agree on which of these two exceptions will be thrown if both problems arise on the same call to `processVote()`. Check/throw the `UnknownCandidateException` before the `DuplicateVotesException` if both apply; our reference solution and test cases will do/assume the same.

3. Support Methods to Compute the Winner

Next, add the following two methods for counting votes. Each returns the name of the winning candidate except where noted. You may assume that these methods will only be called after at least one valid vote has been processed:

In `findWinnerMostFirstVotes()` the winner is the candidate with more than 50% of first place votes. If no candidate receives more than 50% of the votes, return the string "Runoff required".

In `findWinnerMostPoints()` the winner is the candidate with the most points under the following formula: three points for each first-place vote they received, two points for each second-place vote they received, and one point for each third-place vote they received. If there is a tie between two or more candidates, please return the name of any one of the winners (it doesn't matter which).

You should delete the `countVotes()` method from the starter file once you have these methods in place. We put `countVotes()` in the starter file only to give you an initial understanding of the voting system.

4. Include appropriate access modifiers

Put an appropriate access modifier on each of your fields and methods. Please be very mindful about writing getters and setters. You will likely not need them for most fields.

5. Separate the User Interface from the Voting Data

So far, you have the entire voting system (ballot information, votes, and input-output methods) in a single class. Separate the system into two classes: `VotingMachine` for the input/output portion and `ElectionData` for the ballot and votes information. The `VotingMachine` class should have a variable that holds an object of the `ElectionData` class.

The `VotingMachine` class should be where you catch your exceptions, much like we did with `BakingConsole` in the in-class example. Catch your exceptions as follows:

- If you catch an `UnknownCandidateException`, prompt the user if he or she would like to add the candidate's name to the ballot. If the user types in "Y" or "y", call an `addWriteIn()` method (also in `VotingMachine`) that takes the candidate's name as a single parameter. Then restart the voting process.
- The `addWriteIn()` method calls the `addCandidate()` method in `ElectionData`. If `addCandidate()` throws a `CandidateExistsException`, tell the user that the candidate already exists. Otherwise, inform the user that the candidate was added successfully.
- If you catch a `DuplicateVotesException`, inform the user that he or she cannot vote for the same candidate twice, and restart the voting process.

6. Testing

Think of your test cases as mock elections: each will set up a ballot, cast some votes, and determine the winner by one of the two voting methods. Here is an example of one way to set up a test case for this assignment: we write a method to populate a ballot and cast votes, then write a test method to test the outcomes.

```
// method to set up a ballot and cast votes

ElectionData Setup1 () {

  ElectionData ED = new ElectionData();

  // put candidates on the ballot
  try {

     ED.addCandidate("gompei");
     ED.addCandidate("husky");
     ED.addCandidate("ziggy");

  } catch (Exception e) {}

  // cast votes

  try {

    ED.processVote("gompei", "husky", "ziggy");
    ED.processVote("gompei", "ziggy", "husky");
    ED.processVote("husky", "gompei", "ziggy");

  } catch (Exception e) {}

  return(ED);

}

// now run a test on a specific election
@Test
public void testMostFirstWinner () {
  assertEquals ("gompei", Setup1().findWinnerMostFirstVotes());
}
```

Note that tests should run directly through the `ElectionData` methods, not through the screen-based interface. You do not have to unit test the interface (including the `addWriteIn()` method).

You may run multiple tests on the same ballot and/or election, or you may create different elections.

Your test cases must include unit tests that check for user-defined exceptions (that includes both `DuplicateVotesException` and `UnknownCandidateException`). Such test cases look like the following:

```
@Test(expected=SampleException.class)
public void testSampleMethod() throws SampleException
{
        myObject.sampleMethod(sampleArgument);
}
```

where sampleException is the exception that the method should throw in this example.

NOTE: The above syntax is for JUnit 4. Please make sure you are running JUnit 4--NOT JUnit 5--when conducting your tests.

7. JavaDoc Comments

If a method throws an exception it will have `throw [ExceptionName]` as part of the method signature. This introduces a new JavaDoc tag: `@throws`. Please be sure to use this tag as part of your JavaDoc above your methods. For instance:

```
/**
 * This method does something.
 * @throws SomeException when something exceptional happens.
 **/
public void someMethod() throws SomeException
{
        ...
}
```

Program design expectations

In grading this assignment for program design, we will be looking for various good OO coding practices as we have discussed them this term. This includes

- Creating helper functions where appropriate
- Making sure methods are in the appropriate classes
- Avoiding unjustified uses of null
- Encapsulation and grouping related data into classes as appropriate
- Testing in the presence of multiple correct answers

Grading

Homework 6 Rubric (https://canvas.wpi.edu/courses/16466/files/2355129/download?download_frd=1)

Programs must compile in order to receive credit. If you submit a program that doesn't compile, the grader will notify you and give you one chance to resubmit within 24 hours; a penalty (25% of the total points for the assignment) will be applied as a resubmission penalty. Code that is commented out will not be graded.

What to Turn In

Submit (via **InstructAssist** **(https://ia.wpi.edu/cs2102/)**) a single zip file (not tar, rar, 7zip, etc) containing all of your .java files. Do not submit the .class files. The name of the project in InstructAssist is **Homework 6**.