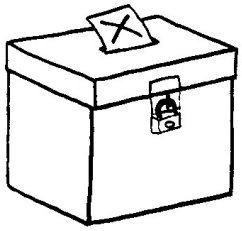
CSE 331, Spring 2011
Homework Assignment #4: Election (50 points)
Due Monday, May 2, 2011, 8:00 AM

This program focuses on class design, design patterns, and JUnit testing. Turn in the Java files listed below from the course's homework web page. This is an optional **pair** assignment; you may choose to work alone, or have a "design buddy" as you did in HW3, or you may have a coding partner with whom you will work on the implementation of this project (see the "Teams and Buddies" section of this document).

You will need to download support files like `ElectionMain.java` and `ElectionTextUI.java` from the course web site into your project to run the GUI. Run `ElectionMain` to launch the program. We have also supplied a bit of helper code to get you started in `Arguments.java` and `CollectionUtils.java`. Note that input text files should be placed in your Eclipse project's root folder, the *parent* of its `src/` or `bin/` subdirectories.

Background Information about Elections and Instant Run-Off:



In this program you will write a set of supporting classes for an **election vote-counting system**. An *election* is a formal process by which a population chooses a person to hold a public office, such as the President of the United States. Normally an election is conducted by having various people register that they are *candidates* for the position. Then on a given election day, the citizens are asked to submit *ballots* to vote for a particular candidate. A voter can choose to vote for a person who has not officially been placed onto the ballot, also known as a *write-in candidate*. In elections for larger offices, such as a governor or President, the votes are taken at various *polling places* and then combined and counted.

Candidates for election are often associated with *political parties*, groups that have well-defined sets of principles and rules about their preferred government policies. In America there are two major political parties, the Democratic Party and the Republican Party, but other "third parties" exist as well.

The winner of an election is the candidate who receives the most votes. But most countries insist that any election winner must be elected by a *majority vote*, that is, that the winning candidate must receive over 50% of the total votes. In a case where a large number of candidates have run for election, the candidate with the most votes might not have over 50% support. For example, if Candidates A, B, C, and D run and receive 29%, 35%, 16%, and 20% of the vote respectively, Candidate B has the most votes but has under 50%. This is called a *plurality*. Most elections do not permit a candidate to win by plurality, so measures are taken to decide a majority winner.

A common way to reach a majority from a plurality vote is to perform a *run-off* vote. A run-off is a re-vote with some of the candidates eliminated. For example, in the situation described previously, candidates B and D received the most votes, so there could be a run-off where the only two choices on the ballot are B and D. This re-vote might lead to B receiving 56% and D receiving 44%, causing B to be elected to the office.

Since it can be costly and time-consuming to conduct a physical run-off vote, some regions perform *instant run-off* voting. The idea with an instant run-off is that on the original ballot, rather than casting a vote for a single candidate, voters give a ranking to *every* candidate, from most to least preferred. Voters rank candidates in order of preference, and their votes are initially allocated to their first choice candidate. If after this initial count no candidate has a majority of votes cast, the candidate with the fewest votes is eliminated and votes for that candidate are redistributed according to the voters' second preferences. This process continues until one candidate receives more than 50% of the votes, upon which they are declared the winner. We will refer to each elimination of a candidate as a *round* of voting.

For example, in the previous situation where A, B, C, and D received 29%, 33%, 16%, and 20% of the vote respectively, Candidate C got the fewest votes (16%). In an instant run-off, we would then eliminate C from the vote and convert each of the C ballots into votes for those folks' second-favorite candidate. Suppose that of the ballots that listed C first, half listed B second and half listed A second. The second round of voting would list A, B, and D with 37%, 43%, and 20% respectively. We still do not have a majority; now D is the candidate with the fewest votes. Suppose that of the voters

who had D as their current first choice, 80% listed A as the next choice and 20% listed B next. For our third round we redistribute these votes and end up with A and B having 53% and 47%. Therefore A would be chosen as the winner.

It might be surprising that A won our example and not B, the candidate who originally had the most votes. The idea of an instant run-off is that while more people preferred B as their first choice, the overall satisfaction with B was lower than the overall satisfaction with A; people who liked candidates other than A/B preferred A as a fallback choice over B. Supporters of the instant run-off voting system claim that it encourages voting for third-party candidates because even if that candidate does not win (and is eliminated during a round of the run-off process), one's votes for that candidate will be redistributed to another candidate that is also desirable to the voter. This helps to avoid the concern that voting for a third-party candidate will actually hurt the election, such as voting for Ralph Nader or Ron Paul in 2008 may have taken votes away from Barack Obama or John McCain respectively. Learn more about instant run-off voting at:

- http://en.wikipedia.org/wiki/Instant-runoff_voting
- http://instantrunoff.com/
- http://archive.fairvote.org/irv/faq.htm

Program Description:

In this assignment you will design and implement an election vote-counting system that uses instant run-off. You will be given input files listing the major candidates, as well as input files containing the votes cast on each ballot. Your program will read the votes from each polling place, combine them, and display information about each candidate's votes. If no candidate has a majority, your code will advance to a new round by eliminating the candidate with the fewest votes and re-displaying the current vote counts. Once a candidate reaches a majority, the election ends.

You may assume that the classes you design will always be used to handle a single election, not multiple elections. You may also assume that all candidates represent one of the following fixed political parties:

name	abbreviation
Republican	REP
Democrat	DEM
Libertarian	LIB
Green Party	GRN
Constitution Party	CNS
Tea Party	TEA
Pirate Party	ARR
Unknown	???

An election can be in an "open" or "closed" state. In the "open" state, votes are still coming in from various polling places. In this state you are allowed to add polling place data to the election, and allowed to change the election to the "closed" state, but that's all. In the "closed" state, you aren't allowed to add any more data from any polling places. But you are allowed to view the voting results, either from the overall election or from any individual polling place. You are also allowed to eliminate the candidate with the fewest votes and then view the new election result data.

Whenever election vote results are shown, the listing of candidates is **sorted from most to fewest votes**. If a given pair of candidates tie with the same number of votes, the **tie is broken by the alphabetical order of their names**, with the name that comes earlier in ABC order coming later/lower in the listing. (*Hint:* Consider using comparators for this.)

There are many real-world aspects of elections that are not represented in this system. For example, the system does not anything about voters' names or identifications, since the ballots are anonymous. It also does not know about relevant political issues or policies supported by each candidate. It also has no notion of dates and times; the code is assumed to be running after the real-life election has concluded and the votes have somehow been tallied into a digital file format.

Example Input/Output Console Log:

The following log demonstrates the behavior of your program in a particular usage case. The log is not exhaustive; see the course web site for additional logs and details. Your program is expected to match this output, most importantly in terms of the data and computations made, but also in terms of the wording, spacing, alignment, precision, casing, and exact text of the output. User input is shown in a bold underlined font.

```
CSE 331 Election Vote Counter

Main System Menu
----------------
A)dd polling place
C)lose the polls
R)esults
P)er-polling-place results
E)liminate lowest candidate
?) display this menu of choices again
Q)uit

Main menu, enter your choice: a
Name of polling place: belleVUE
Added belleVUE.

Main menu, enter your choice: c
Closing the election.

Main menu, enter your choice: r
Current election results for all polling places.
NAME                          PARTY   VOTES     %
John McCain                   REP         6     46.2
Ron Paul                      CNS         4     30.8
Barack Obama                  DEM         2     15.4
Ralph Nader                   GRN         1      7.7

Main menu, enter your choice: e
Eliminating the lowest-ranked candidate.
Eliminated Ralph Nader.

Main menu, enter your choice: r
Current election results for all polling places.
NAME                          PARTY   VOTES     %
John McCain                   REP         6     46.2
Ron Paul                      CNS         5     38.5
Barack Obama                  DEM         2     15.4

Main menu, enter your choice: e
Eliminating the lowest-ranked candidate.
Eliminated Barack Obama.

Main menu, enter your choice: r
Current election results for all polling places.
NAME                          PARTY   VOTES     %
John McCain                   REP         7     53.9
Ron Paul                      CNS         6     46.2

Main menu, enter your choice: e
A candidate already has a majority of the votes.
You cannot remove any more candidates.

Main menu, enter your choice: q
Goodbye.
```

Classes to Implement:

This document does not specify exactly what classes you should write, nor what behavior, fields, methods, etc. each class should have. Part of this assignment is for you to choose appropriate classes and contents for each class based on the specification of desired functionality and based on our class discussions of proper class design. You may use any classes you like, but please don't use packages on your classes; place all of them into the **default (unnamed) package**.

A major part of your grade will come from how effectively you design the contents of each class to solve this problem elegantly. You should also incorporate **design patterns** taught in class as appropriate to help you solve this problem. (You don't need to use every pattern taught in class; only use the patterns when they improve the program's design.)

In general, if an entity is important in this system, you should represent that entity with a Java class. It would be inappropriate to try to solve this program using only one or two classes; there are more important entities in this system. If you opt for a design that has too few classes and omits major entities from the system in favor of simple variables or collections, you may not receive full credit.

Input File Formats

An election has a set of known **candidates**. They are listed in an input file; the default file name is `candidates.txt`. The file lists one candidate per line, with the name and an abbreviation of the political party separated by commas:

```
Barack Obama,DEM
John McCain,REP
Ralph Nader,GRN
Ron Paul,CNS
```

Your program should read this input file to know the registered candidates running for election. Reading input from a file is a heavy-duty process, so you should provide a "producer" method that reads a file and uses its data to populate an appropriate object. You may assume that no candidate's name is more than 29 characters long. The provided input files have a small number of candidates, but your code should work for any number of candidates that might appear in the file.

The text user interface allows the user to add data from a given **polling place**. The program prompts the user for the polling place's name, then reads all vote data from that polling place and incorporates that data into the election results. Each polling place corresponds to a file with a similar name, which is the name of the polling place in lowercase with all spaces replaced with dashes. For example, the polling place "Redmond Town Center" stores its votes in a file named `redmond-town-center.txt` . Vote files consist of one ballot per line, with the voter's candidates listed in order of preference from highest to lowest, separated by commas. For example:

```
Ron Paul,Barack Obama,John McCain,Ralph Nader
John McCain,Ron Paul,Ralph Nader,Barack Obama
Barack Obama,Ralph Nader,John McCain,Ron Paul
Ralph Nader,John McCain,Ron Paul,Barack Obama
```

You may assume that every polling place's name consists entirely of letters (in upper or lower case) and spaces. You may assume that the candidate input file contains no duplicate names or names that differ by ≤ 3 characters. A candidate listed in the file might have an invalid political party that is not in the list earlier in this document; if so, it is an error.

Your classes should throw an exception when a given input file is not in the proper format (see "Exceptions" section).

Write-In Candidates:

It is possible for a ballot to contain a vote for a candidate who is not among the list of registered candidates. This is known as a **write-in candidate**. If you come across a vote for such a candidate, you should consider the write-in candidate to be part of the election and count the vote for that candidate. If a write-in candidate gets enough votes, it is possible for that person to win the overall election. For example, the following ballot casts a first-place vote for Stuart Reges (who is not a registered candidate) but falls back to other registered candidates if Stuart is not elected:

```
Stuart Reges,Ron Paul,Barack Obama,Ralph Nader,John McCain
```

Since write-in candidates are not listed in the official registry of candidates, their political party is Unknown.

You may assume that every ballot in every file contains a listing for every candidate from the registered candidates list. If there are write-in candidates, the line will contain more than that many candidates. For example, if the list of registered candidates contains 4 candidates and the voter has 2 write-ins, that ballot line in the file will contain a total of 6 names.

Another special case of input that your program must handle is that of **misspelled candidate names**. Some users are not great spellers, and they write the name of their candidate improperly. Here is an example of such a ballot:

```
Ralph Nader,Ronn Paul,Jimmy McClane,Barak Obomma
```

Your program should attempt to detect misspelled candidate names and count them as proper votes for the intended candidate. You should do this by computing the **edit distance** (also called **Levenshtein distance**) between the voter's spelling of the candidate name and the expected spelling. The edit distance between two strings is defined as the minimum number of single-character (insert, delete, or change) operations that must be performed to change one string into the other. For example, to change "mitten" to "sitting" you must perform a minimum of 3 operations: change "m" to "s", change "e" to "i", and add "g" at the end. "mitten" → "sitten" → "sittin" → "sitting".

If the edit distance between the voter's spelling of their candidate's name is **within 3 operations or fewer** from the name of a registered candidate, case-insensitively, you should cast the vote for that registered candidate. For example, in the misspelled ballot above, "Ronn Paul" becomes "Ron Paul" with 1 edit, removing the second N; and "Barak Obomma" becomes "Barack Obama" with three edits: adding a "c" before "k", changing "o" to "a", and removing the second "m". If the name the voter has written has an edit distance of more than 3 from every registered candidate's name, you should treat this name as a write-in candidate. "Jimmy McClane" above is a bit like "John McCain" but its edit distance exceeds 3, so you should treat this as a write-in candidate. You may assume that no misspelled candidate name is within ≤ 3 operations from two or more distinct registered candidate names at the same time.

The **algorithm** for computing the Levenshtein distance between two strings involves creating a 2-dimensional array of counters and looping over both strings to fill the array. The algorithm description and pseudo-code can be found at the following address:

- http://en.wikipedia.org/wiki/Levenshtein_distance

Text User Interface:

Your program uses a **text user interface**. You are given a small amount of provided code for the user interface to get you started, but you must make some modifications to the provided UI to link it to your other classes and code. The parts of the UI code that need editing have been marked with `// TODO` comments to make them easier to find.

The classes you add to the program should not perform any **user interaction**. That is, there should not be any `println` statements or `Scanner` input from the console performed directly by them. Instead, the provided `ElectionTextUI` class should perform all user input/output. Your other classes are allowed to read data from files as appropriate.

Do not place core system functionality into the text UI class. Its primary purpose should be to read input from the user and then hand off this input data to your other core objects as described previously, to let those core objects do the bulk of the actual work and computations. For example, it is not appropriate for the text UI class to contain the code that counts votes and determines the winner of the election. You may add fields and methods to the text UI class as necessary or desired, but the text UI should not directly store any of the core data of the system.

Your classes are to exactly reproduce the format and overall behavior found in our sample solution and sample outputs. In several places the text UI displays **percentages** of votes. A percentage should always be displayed by the UI with exactly 1 digit after the decimal point, such as 54.2% .

You will have to run the text UI and test individual inputs on your own to verify that your classes are working correctly. A **sample solution** will be posted to the course web site that you can run to verify the behavior your program should have. The sample solution is our best effort to try to meet this spec, but it might contain minor bugs or errors. If the sample solution's behavior differs from this specification, follow this spec, not the sample solution. Please ask if you are unsure.

Exception Checking and Error Handling:

Your classes should forbid invalid parameters. If any method or constructor is passed a parameter whose value is invalid as per this spec, you should throw an exception. In particular, you must enforce the following:

- No parameter passed to any method (except `equals`) should ever be **`null`**.
- Strings that represent names should not be **empty** or consist entirely of **whitespace**.
- Integer fields that store counts of real-world things, such as a number of votes, should always be **non-negative**.
- If you parse data from files as strings and turn the strings into objects, you should check that the strings are in a **valid format** and throw the exception if they are not.

In past assignments you were instructed explicitly to always throw an `IllegalArgumentException` on any error. In this assignment we will let you decide what type of exception is appropriate to throw. In many cases it should still be `IllegalArgumentException`, but in some cases it may make more sense to throw other types such as `NoSuchElementException` when a client mistakenly asks a collection for an element it does not have, `IllegalStateException` when a client tries to perform an operation on an object at an inappropriate time, or `InputMismatchException` when the format of input data (such as from a file) does not match what is expected.

The main client program should not throw any **exceptions** to the console. If any errors occur, it is considered a bug in your code. It is okay (encouraged) for you to incorporate exception throwing into the design of your core classes, but you should make sure to catch those exceptions in the text UI client or somewhere else in the active call stack.

The text UI performs its own checking for the validity of various user input values, such as making sure that numbers are non-negative. But this is completely unrelated to your own tests and exception checks. Remember that valid arguments are preconditions for your methods, and preconditions are *supposed* to be things that the client could check for and avoid. Just because this particular client does so does not absolve you from performing checks in your core classes' code.

Teams and Buddies:

For this assignment you are allowed to optionally have one partner to work on the program. You aren't required to have a partner; you may complete the assignment by yourself if you prefer. If you choose to have a partner, the two of you may work together fully, sharing all aspects of your code and design with each other. If you have a partner, you and your partner should turn in a single copy of your assignment (you don't both need to turn it in). If you have a partner, make certain that you **document this in the comments** of your files so that the grader will know.

If you do decide to work alone, you may still choose to have one optional **"design buddy"** for this assignment. Your "design buddy" is one other student in the class with whom you are allowed to speak in great detail about the designs you both have chosen for your classes. In other words, you may share with your buddy the exact classes, methods, fields, etc. that you have chosen to use to implement this specification along with the reasoning behind those decisions, pros/cons you have discovered about your implementation of those decisions, etc. The goal behind this is to allow you a limited amount of greater collaboration, to help make the assignment more manageable, and for you to learn from each other.

If you choose to have a "design buddy," each of you should **document this in your code** by writing your buddy's name in your class header comments. (It is generally good practice to cite sources and collaborators in submitted code.) You should also include the design buddy in the `@author` tag of any Javadoc comment headings atop your classes. If you choose to work with a partner, you may not also have a separate design buddy.

The instructor and TAs will not be providing you with any help or resources to facilitate working as a team, such as shared file storage space, version control repositories, etc. It will be up to you to coordinate this with your partner.

If you do work with a partner, we expect that **both partners should contribute substantially** to the final program. It is not appropriate for one team member to do the vast majority of the work. To help us understand which partners worked on which parts of the program, please indicate in your comments which person was the primary author (the person who did the majority of the work) on each file you submit. If the two of you sat together and co-authored a file, please indicate this; but even in such a case, someone must have been at the keyboard "driving" the input of the code, so indicate this.

Development Strategy:

It can be difficult to come up with the classes in this system without guidance. There may be some objects that are non-obvious from a real-world perspective but still useful in this system. For example, your program needs to display a count of votes for the entire election and also for a given polling place. You don't want to have this code in two places. So perhaps there should be another class that can be given vote data from various sources and tabulate relevant results.

If you try to write all of these classes at once and then compile/run the overall program, you are unlikely to succeed. This is a large system that uses all of your classes in complex ways. A small error in your code will stop it from functioning entirely, giving you poor feedback about what code does and does not work successfully.

Therefore it is important to write and test your code incrementally. We suggest that you initially write any smaller, simpler classes that have less behavior and fewer connections to other classes. Remember that you can insert a "stub" version of a particular method or entire class that simply has a pair of empty `{}` braces for the method's body (or simply returns a dummy value like `null` or `-1`). This may allow you to test unfinished classes together.

JUnit Testing:

Along with your program files you will turn in a **JUnit 4 test class** that contains code to test at least one of your other classes. For full credit, this test class should test the majority of the behavior of the class under test, including edge cases, illegal parameters, exceptions, and so on. Choose a representative set of test cases to try to handle all major behaviors and cases that would be encountered by the code of the class under test. You should follow the guidelines taught in class about how to properly craft testing methods and how to choose test cases effectively.

Teams may, of course, share testing code between partners. And if you have a "design buddy", you may share your testing code with that buddy if you like. Otherwise, please do NOT share testing program code with other students. The reason is that your testing code reveals details about the implementation and design you have used, through the objects it tries to construct and the methods it tries to call on them, and we do not want your design to be revealed to other students.

If you choose to create other election data input files to test your program, you may, however, share those input files or input/output logs with any other students.

Style and Design Guidelines:

Since you will be designing the classes for this assignment, a large part of your grade will come from the quality of your **object-oriented design**. Follow guidelines taught in class such as: striving for the "C" words (cohesion, completeness, clarity, convenience, and consistency); minimizing unnecessary coupling; choosing a good set of methods and fields for each class; designing constructors to accept appropriate parameters to fully initialize the object and minimize their workload; following standard naming conventions; making all classes `final` that are not to be extended by inheritance; following the Expert pattern, the Law of Demeter, and the Open-Closed Principle; and avoiding representation exposure.

Your code will be graded on whether it follows the **style and design guidelines** taught in lecture, particularly those indicated as tips from the *Effective Java* textbook. Follow the **object-oriented design heuristics** discussed in lecture from Arthur Riel's book by the same name. Follow a readable and consistent style for your classes such as the one found in the Sun Java official coding conventions linked from the course web site.

Follow the **Expert pattern** throughout your code, making sure that the object that knows the most about how to perform a given task (or is most closely related to the idea of that task) performs that task. For example, if a given type is read and/or written from a data source such as a file, that type should be the expert in regards to how to read itself properly.

Utilize **design patterns** in your class design as appropriate, such as Singleton, Flyweight, Prototype, Factory, and Iterator. Document wherever you use patterns by commenting them.

The majority of the methods you write should run in **constant time** (O(1)) regardless of any parameter value(s) passed. Methods that must process an entire data structure iteratively are by nature not O(1); this is expected and allowed for some parts of the functionality of the program, such as displaying all candidates and their vote counts.

This document does not specify what **collections** to use inside of your various objects. But you should always choose data structures and algorithms that minimize the complexity class necessary for solving a given task. Choose the

collection that is most appropriate and that will provide the fastest overall runtime for the operations implemented, taking into account any issues related to desired ordering, duplicates, mapping, and so on.

You should intelligently choose which behavior should and should not be declared **`static`**. The majority of your classes' behavior should be non-static, but methods that bear no relation to the data of any particular object should be `static`.

You should intelligently choose which behavior should and should not be declared **`public`**. If a given method is a natural part of how clients would expect to use that object, make it `public` in the interest of providing a clear, complete interface. If a method is used primarily by the class upon itself to help it implement other functionality, it should be made `private`.

Minimize mutability; if a class can be cleanly implemented as an **immutable** class, you should do so.

As appropriate, your classes should provide standard Java methods such as `equals`, `compareTo`, and `clone`.

A major focus of this assignment is ensuring that all of your objects are always kept in a valid state. This is accomplished through rigorous **argument checking** and throwing exceptions on invalid arguments. Part of your grade will be based on whether you handle all of the exceptions and all possible combinations of invalid arguments that could be passed.

Document all of your files by commenting them descriptively *in your own words* at the top of each class, each method/ constructor, and on complex sections of code. Use **Javadoc comments** for all external documentation (atop class headers or public method/constructor headers). Include descriptive initial summaries along with proper tags such as `@param`.

Your **class header comments** should include this course (the assignment / course / section), author information (your name and other contact info), a description of the Java class itself (the overall purpose of the Java class written in the file), along with any class **invariants**. Think carefully about what invariants clients may assume about the objects you have written. Come up with a reasonable set that is useful to clients and consistent with the expected functionality of each class and the system; document those invariants, then enforce them rigorously throughout the class.

Your **method header comments** should describe the method's behavior, parameters, return values, exceptions thrown (the type of exception and what would cause it to occur), any modifications to the state of the current object ("this"), and any pre/postconditions. Private helper methods are graded more loosely on commenting than public ones; simply give a private method a brief header explaining its purpose.

You should also include **internal documentation** in the form of `//` comments written inside bodies of classes and methods as appropriate to describe complex sections of code. These comments are different than external comments; they are for a fellow developer or maintainer of your own classes, not for clients. Therefore these internal doc comments should describe what the code is doing and how/why it is doing it, as appropriate. Trivial code does not need such comments, but complex operations should receive at least a modest amount of internal documentation.

Redundancy is a major grading focus of every assignment for this course. If you find yourself repeating code, make the code into a method or pull it out into its own class to reduce the redundancy.

Follow good **general style guidelines** such as: making fields `private` and avoiding unnecessary fields (you will lose points if you declare variables as fields that could instead be declared as local variables); declaring collection variables using interface types (e.g. `List` rather than `ArrayList`); initializing fields in constructors, not at declaration; appropriately using control structures like loops and `if`/`else`; following "Boolean Zen" (proper use of boolean logic); properly using indentation, good variable names and types; and not having any lines of code wider than 100 characters.

As a rough sanity check, our solution contains roughly **325 "substantive lines"** according to the class Indenter Tool.

Submitting Your Files:

Since we don't know what classes you will choose, you should **submit a .zip file** named hw4.zip containing all .java source files necessary to build and execute your program. We should be able to compile and run your code as follows:

```
javac *.java
java ElectionMain
```

The files in your ZIP archive should be in the root level directory of the ZIP archive; in other words, when we unzip your file, your .java files should appear in the current directory. Do not include .class, .txt, or other supporting files, only .java.