P05 Ballot Box

Overview

As the United States 2020 general election is approaching, we're going to create the object-oriented back-end of a simple voting application. The design of this application will consist of:

- a **Candidate**, who has a name and an office they're running for;
- a **Party**, which contains an array of **Candidates**;
- a **Ballot**, which collects votes for **Candidates** for each office; and
- a **BallotBox**, which collects **Ballots** and reports the winner for each office

In this specification, we will walk you through the creation of these objects and get you started with your test methods.

Grading Rubric

5 points	**Pre-assignment Quiz**: accessible through Canvas until 11:59PM on **10/18**.
25 points	**Immediate Automated Tests**: accessible by submission to Gradescope. You will receive feedback from these tests *before* the submission deadline and may make changes to your code in order to pass these tests. Passing all immediate automated tests does **not** guarantee full credit for the assignment.
20 points	**Additional Automated Tests**: these will also run on submission to Gradescope, but you will not receive feedback from these tests until after the submission deadline.

Learning Objectives

The goals of this assignment are:

- Familiarize yourself with the typical setup of Object-Oriented Programming
- Create and use a series of interrelated objects
- Create constructors, accessors, and mutators to set, examine, and modify private data fields
- Familiarize yourself with the difference between (static) class methods and data, and (non-static) instance methods and data

Additional Assignment Requirements and Notes

Keep in mind:

- You may NOT have any import statements besides java.util.ArrayList and any relevant exceptions.
- You may NOT define any additional class or instance fields beyond the ones detailed in this specification.
- You are allowed to define any **local** variables you may need to implement the methods in this specification.
- You are allowed to define additional **private** helper methods to help implement the methods in this specification.
- All methods, public or private, must have their own Javadoc-style method header comments in accordance with the CS 300 Course Style Guide.
- Any source code provided in this specification may be included verbatim in your program without attribution.

CS 300 Assignment Requirements

You are responsible for following the requirements listed on both of these pages on all CS 300 assignments, whether you've read them recently or not. Take a moment to review them if it's been a while:

- Academic Conduct Expectations and Advice, which addresses such questions as:
    - How much can you talk to your classmates?
    - How much can you look up on the internet?
    - What do I do about hardware problems?
    - and more!
- Course Style Guide, which addresses such questions as:
    - What should my source code look like?
    - How much should I comment?
    - and more!

Getting Started

1. Create a new project in Eclipse, called something like **P05 Ballot Box**.
    a. Ensure this project uses Java 11. Select "JavaSE-11" under "Use an execution environment JRE" in the New Java Project dialog box.
    b. Do **not** create a project-specific package; use the default package.

2. Create 5 Java source files within that project's src folder:
    a. **Candidate**.java (does NOT include a main method)
    b. **Party**.java (does NOT include a main method)
    c. **Ballot**.java (does NOT include a main method)
    d. **BallotBox**.java (does NOT include a main method)
    e. **BallotBoxTester**.java (includes a main method)

The methods in the **Candidate**, **Party**, and **BallotBox** classes will NOT be static methods; all methods in **BallotBoxTester** should be static; and **Ballot** will contain a mix of static and non-static methods.

Implementation Requirements Overview

Your **Candidate** class must contain the following **class constant**, visible at the package level:

- `protected static final String[] OFFICE`
    - Initialize this to contain **at least three (3)** different office titles of your choice
    - You may use President, Vice President, Secretary if you like, but you may also be more (or less) creative

Your **Candidate** class must contain the following private instance variables:

- `name`, a String representing the Candidate's name
- `office`, a String representing the Candidate's office

Your **Candidate** class must contain the following instance methods:

- `public Candidate (String name, String office)`
    - A two-argument constructor, which initializes the instance variables for the object
    - Checks that the office is valid and throws an IllegalArgumentException if not
- Accessor methods for both of the object's instance variables:
    - `public String getName()` to get the name.
    - `public String getOffice()` to get the office.
- `public String toString()`
    - Returns a String representation of this Candidate as NAME (OFFICE)
    - For example, "Mouna Kacem (President)" or "Hobbes LeGault (Court Jester)"

Your **Party** class must contain the following private instance variables:

- `name`, a String representing the Party's name
- `candidates`, an ArrayList of Candidates affiliated with that Party

Your **Party** class must contain the following instance methods:

- `public Party (String name)`
    - A one-argument constructor, which initializes the instance variables for the object
- Accessor methods:
    - `public String getName()` to get the name.
    - `public int getSize()` to get the number of Candidates belonging to the Party.
    - `public Candidate getCandidate(String office)` to find the candidate from the Party running for the given office. Throws a NoSuchElementException if no Candidate is found.
- `public void addCandidate(Candidate c)`
    - Adds a Candidate to the Party
    - Throws an IllegalArgumentException if the provided Candidate is running for the same office as another member of the Party

Your **Ballot** class will contain a mix of class (static) and instance (non-static) values and methods. Be careful!

Your **Ballot** class must contain the following private class variables, which will be common to all Ballots:

- `parties`, an ArrayList containing the Parties for the election
- `counter`, an int value for generating unique ID values for each Ballot

Your **Ballot** class must contain the following class methods:

- `public static void addParty(Party p)`
  - Adds a Party to the class ArrayList
  - If the class ArrayList already contains this Party, do nothing
- `public static ArrayList<Candidate> getCandidates(String office)`
  - Creates and returns an ArrayList of all Candidates running for the given office
  - Should not crash if a Party has no Candidate running for the given office

Your **Ballot** class must contain the following private instance variables:

- `votes`, an array of Candidates with the same number of elements as the constant String array in the Candidate class. Each element of this array corresponds to the Ballot's vote for that office, where a null entry means no vote has (yet) been cast for that office.
- `ID`, a final int representing the **unique ID** of this Ballot

Your **Ballot** class must contain the following instance methods:

- `public Ballot()`
  - A no-argument constructor, which initializes the votes array to the correct length and creates a unique ID number for the Ballot
- Accessor methods:
  - `public Candidate getVote(String office)` returns the Candidate that this Ballot has voted for, or null if there is no vote for that office.
  - `public boolean equals(Object o)` overrides the default Object method equals, and determines equality by comparing the unique ID values of two Ballots. If the provided Object o is not a Ballot, this should return false.
- `public void vote(Candidate c)`
  - Records the vote for the given Candidate at the appropriate index in the Ballot's array.
  - Can overwrite an existing vote.

Two more classes to go...

Your BallotBox class must contain the following private instance variable:

- `ballots`, an ArrayList of the Ballots which have been cast

Your BallotBox class must contain the following instance methods:

- `public BallotBox()`
    - A no-argument constructor, which initializes the instance variable for the object
- Accessor methods:
    - `public int getNumBallots()` returns the total number of unique Ballots in this BallotBox
    - `public Candidate getWinner(String office)` records all of the existing Ballots' votes for the given office and returns the Candidate with the most votes. Make use of your other classes' accessor methods here!
- `public void submit(Ballot b)`
    - Adds a Ballot to the BallotBox if and only if the Ballot has not already been added.

Your BallotBoxTester class must contain boolean test methods for each of your classes. For example, for the Candidate class you might write:

- `public static boolean testCandidate()` to test the constructor, accessors, and toString method

But for classes with accessor methods that do more than just show you the values from a given instance's constructor, you might want to break it out into more detail:

- `public static boolean testPartyConstructor()`
- `public static boolean testPartyGetCandidate()`
- `public static boolean testPartyGetSize()`

You must write at least one (1) boolean test method per instantiable class. As these methods are all contained within the same test class, they must be named so that it is obvious which class they are testing (see above examples).

Implementation Details and Suggestions

We recommend that you begin your implementation with the **Candidate** class, as it's both simple and doesn't depend on any of the other classes.

Candidate: Implement and Test

Follow the implementation requirements outlined above, and test your class as follows:

- Create two or more instances of the **Candidate** object by calling the constructor with different names and offices.
- Use the accessor methods to verify that each object returns different values (and that they're the ones you expect for that particular instance).
- Verify that the toString() method's return value matches the information returned by the accessor methods (and that it's returning the String rather than printing it).

Party: Implement and Test

Once you have verified that your **Candidate** class works as expected, move on to the **Party** class. The instance variables and getName() and getSize() accessors work very similarly to the **Candidate** class.

Next, we recommend implementing the addCandidate() method, so that you can create and add **Candidates** to this class and test your getSize() method as well. Before you add the **Candidate** to the ArrayList, make sure that no other **Candidate** in this **Party** is running for the same office -- if they are, throw an IllegalArgumentException and *don't* add the **Candidate**.

STOP: write tests for your **Party** methods so far. Create at least two different **Parties**, and add **Candidates** to one of them. Check the size of each **Party** (one should still be 0). Try adding two **Candidates** for the same office to one **Party**. Check the size of the **Party** (it shouldn't change if you threw an exception).

The getCandidate() method should look through the ArrayList of **Candidates** for the **Party** and check whether each **Candidate's** office matches the query String. (Use the **Candidate's** accessor method to check this value.) If nothing matches, throw a NoSuchElementException.

Finish up your **Party** test methods!

Ballot: Implement and Test

Now things get interesting: move on to **Ballot**. Begin by implementing the class (static) variables and methods. Initialize the `counter` class variable to an int value of your choice. These should feel pretty familiar, as you've been doing this all semester.

Note that at this point, if you have tested your **Candidate** and **Party** classes thoroughly, you can effectively use them as though they are provided classes. Call their accessor and mutator methods. Catch their exceptions.

Add in the **Ballot** instance variables (`votes` and `ID`) but don't initialize the constant just yet -- we get one chance to do so, in the constructor, and that's what we're going to do next.

Begin the constructor by initializing the `votes` array to a new array of the same length as the **Candidate** class' array of possible offices (if you haven't already done so). In order to guarantee that each **Ballot** object is given a unique ID number, we'll utilize the static int variable -- assign this **Ballot's** `ID` the current value of `counter`, and then increment the `counter`. This way, your first **Ballot** might have `ID=0`, the second has `ID=1`, and so on.

Next we recommend implementing the overridden method equals(). This method's stub will look like this:

```
/**
 * Returns true if the argument is equal to this Ballot, false otherwise
 * @param o the object to compare to this Ballot
 * @return true if the Ballots have the same ID, false otherwise
 */
@Override
public boolean equals(Object o) {
  return false;
}
```

The @Override tag tells the compiler (and other programmers) that this implementation is *overriding* the default Object implementation for equals(). In this case, you will first want to check whether the provided Object o is an `instanceof` the **Ballot** class, and return false if it's not; otherwise, cast o to a **Ballot** and check whether its ID is the same as this **Ballot's** ID. If it's the same, they're equal!

STOP: write some test methods. Create at least two **Ballots**, check whether they're equal. Check whether a **Ballot** object is equal to itself.

Now write the vote() and getVote() methods. Note that a **Candidate's** position in the votes array should correspond to their office's position in the constant array of offices in the **Candidate** class. When testing these methods, you may want to use the static methods from the **Ballot** class to retrieve the list of possible **Candidates** for a given office!

BallotBox: Implement and Test

You're almost there. Begin by adding the instance variable, constructor, submit() and getNumBallots() methods to **BallotBox**, and accompanying tests. Since we have a tested implementation for **Ballot's** equals() method, you can make use of ArrayList's contains() method to detect duplicate **Ballots**!

Finally, implement the most critical method for a voting application: getWinner().

This method should:

1. Retrieve the list of potential **Candidates** for the given office
2. Create some way of tracking how many votes each of these **Candidates** receives
3. Go through each of the **Ballots** in the **BallotBox** and get the vote for that office
4. Add that vote to the corresponding **Candidate's** count
5. Find the **Candidate** who has received the most votes
6. Return that **Candidate**

For this implementation, in the case of a tie between **Candidates**, return the **Candidate** who appears first in the list of potential **Candidates**. This is NOT what you should do in the case of an actual tie, but (hopefully) this voting application will not be used in an actual election…

Finish up your **BallotBoxTester** class and submit!

Assignment Submission

Hooray, you've finished this CS 300 programming assignment!

Once you're satisfied with your work, both in terms of adherence to this specification and the academic conduct and style guide requirements, submit your source code through Gradescope.

For full credit, please submit ONLY the following files (source code, *not* .class files):

- **Candidate**.java
- **Party**.java
- **Ballot**.java
- **BallotBox**.java
- **BallotBoxTester**.java

Your score for this assignment will be based on the submission marked "**active**" prior to the deadline. You may select which submission to mark active at any time, but by default this will be your most recent submission.

Copyright Notice

Additionally, students are not permitted to share source code for their CS 300 projects on any public site.