Individual Assignment 1

During an election, various polling companies contact voters to ask them how they intent to vote in the election. This polling is on-going and is used to predict the outcome of the election.

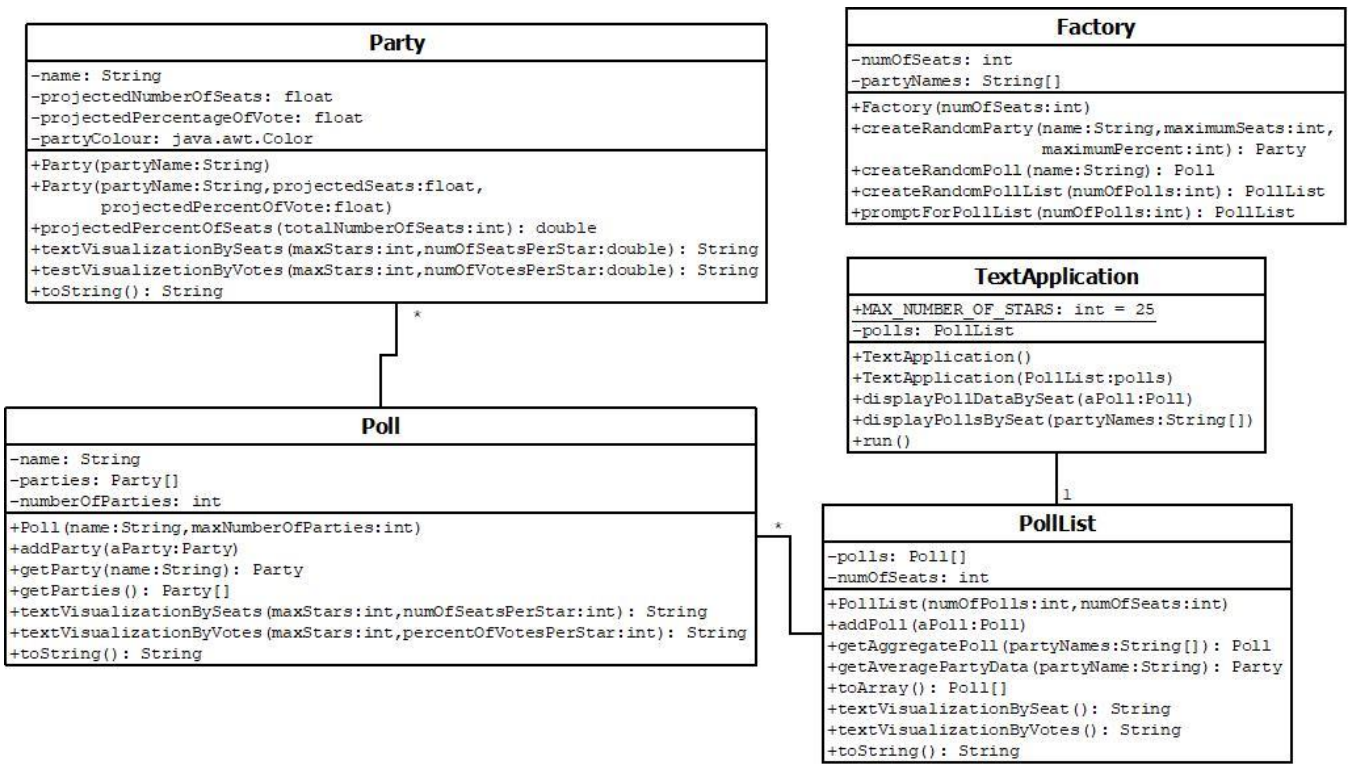
Through the assignments, your application will allow the user to enter multiple polls, update the data for the poll and to show the user the aggregate predictions over all polls. The polls are focussed on representative parliamentary democracy, like Canada's democratic government. (For example, see Representative democracy - Wikipedia for details.)

In this first iteration, you will create a text-based application that will randomly generate poll data and displays the result. For the first individual assignment, you will be assigned one class to complete, either Party, Poll or PollList. Once you have completed your class, you will be placed in a team such that everyone in the team completed a different class. You will use the three classes to create an application by completing the TextApplication class.

This document contains the requirements for Party, Poll and PollList. Make sure to read the requirements for your class in detail. The other two are included for reference if you need to understand more about those classes to be able to complete your own.

Overview of Requirements

The class diagram below shows the five classes that will be included once your team has completed the first team assignment.



For this individual assignment, you will be assigned either Party, Poll or PollList. Only add code to the class that is assigned to you. The team will complete the TextApplication class. The Factory class will be provided.

Grading

There are three grading categories: functionality, code quality, and code documentation. Once you have submitted your solution for the first assignment, we'll first run the JUnit test for your class. If all the tests pass you will be assigned to a team. The actual grading of the assignment will take more time since we'll also look at the quality of the code and documentation.

Code that does not compile, can't be run, or doesn't pass the JUnit tests is not worth any marks and will not be graded. Once the code compiles and all the tests pass, we will read your code to grade it. This means that you should have close to full marks for the functionality portion of the code. (You may lose some marks if error messages are not printed as required.) The class that you created will be marked as follows:

- Functionality:
  - (1 mark) Setter methods do all required validation and print error message when an error is encountered.
  - (1 mark) All other public methods meet requirements.
  - (1 mark) All other public methods deal with unexpected value of arguments by printing an error message while allowing the program to continue. In other words, null pointers, negative values, etc do not cause to program to crash or end in some other way and values displayed are reasonable.
- Code quality:
  - (1 mark) Variable names are self-documenting and code is easy to read and understand.
  - (1 mark) Good use of white space and all methods fit on a single (small) screen (both height and width).
  - (1 mark) Code duplication is avoided and minimized; code is well organized over methods; no additional instance variables added beyond required ones.
  - (1 mark) Doesn't duplicate code from other classes but calls methods in these other classes instead.
  - (1 mark) Code is well organized and easy to maintain: nesting deeper than three levels is avoided; multiple return statements are avoided; break and continue are never used.
- Documentation (see *D2L -> Content -> Setting up a learning environment -> Code documentation in Java* for additional details on expectations for documentation):
  - (1 mark) Class and all public methods are fully javadoc'd with appropriate tags and content.
  - (1 mark) Good use of in-line documentation to explain blocks of code.

Collaboration and Academic Misconduct

You are expected to complete this first assignment without support from classmates or individuals outside the course. If you are struggling when trying to complete your class contact your TA or the instructor directly. You can make an appointment using one of the links in *D2L -> Content -> Teaching Team*. You can direct message (DM) the teaching team in Discord using the handle CPSC219 Teaching Team. Or you can e-mail the course instructor, Nathaly, at nmverwaa@ucalgary.ca.

You may copy code that already exists elsewhere in the public domain and use it for your project. If you do so cite your source! Not citing your source is considered **plagiarism** and will be brought to the attention of the Dean of Science. The best place to cite the source of code is by using method or in-line documentation. If your code is mostly put together by copying code from elsewhere, your mark may be reduced to reflect this.

Detailed Requirements for each Class

Party

This class represents a single political party. The following **instance variables** should all be declared private and should each have a public getter and setter method.

- *name* of type String which can be any string.
- *projectedNumberOfSeats* of type float which must be a non-negative value.
- *projectedPercentageOfVotes* of type float which must be a value between 0 and 1 (both inclusive).
- *partyColour* of type Color imported from java.awt.

The class Color represents colours as RGB (Red, Green, Blue) values (as an int). This site https://teaching.csse.uwa.edu.au/units/CITS1001/colorinfo.html gives a nice overview of RGB values. You can get the intensity of each colour through methods provided in the class. For example, to get the blue intensity value, you would use *partyColour.getBlue().* Take a look at Color (Java Platform SE 7) (oracle.com) for more detail on this class.

If an error is encountered, print an error message. Do NOT end the program. Instead allow the program to continue running as if no error had occurred. A later iteration will manage errors better.

Create **two constructors** for the class: One takes the name of the party as an argument; The other takes the name of the party, the projected number of seats as a float and the projected percentage (as a value between 0 and 1) of the votes as a float.

In addition to the instance variables and their getter and setter methods, also add the following **public methods** to the Party class. (These are also noted in the class diagram above.)

- *toString* which does not take any arguments and returns something of type String. The string returned should have the format: '<name> ([<colour value>], <projected % of votes>% , <projected seats> seats)'.
    - Anything between the angle brackets (<>) describes information that should be placed there.
    - Everything else is the literal character that should be placed there.
    - The projected percentage of seats should be a number between 0 and 100.
    - Colour value should be the values for red, green, and blue as a comma separated list. For example, if the *partyColour* has value Color.RED (with red value 255, green value 0, and blue value 0) then the string representation should be '255,0,0'. If no *partyColour* is set (and the value of *partyColour* is null), then omit the colour value from the string returned.
- *projectedPercentOfSeats* which takes an int as an argument and returns a double. The argument is the total number of seats available in parliament, which should be a positive number. The returned value should be the percentage of seats the party is expected to win as a value between 0 and 1.
- *textVisualizationBySeats* returns a String that gives a visual representation of the data relevant to this party by displaying a row of stars that represents the expected number of seats and a bar to indicate the number of seats needed for a majority in parliament. (Assume that you would need half the total seats, rounded up, to have a majority.) The total available seats can be more than a hundred, so one star per seat is not a reasonable visualization. Instead each star will represent a certain number of seats. The method has two arguments to help you generate such a string:

- *maxStars* which is an int representing the maximum number of stars that should be displayed on a single line.
- *numOfSeatsPerStar* which is a double and indicates how many seats are represented by a single star.

For example, if we call textVisualization with 20, 18.0, on a party that is expected to win 208 seats, the visualization would include 11 stars (rounded down from 11.56) with a bar between star 10 and 11 and would look as follows:
**********|*

If the party has fewer seats than needed for a majority, there should be spaces (one for each 'missing' star) between the last star and the bar. For example, if we call textVisualization with 15, 4.6, on a party that is expected to win 25 seats, the visualization would include 5 stars with a bar three spaces after the last star and would look as follows:
***** |

A final piece of information to add to this visualization is the party information that is returned by the toString method. This information should be added at index (maxStars + 1). (This ensures that is at least one space between the last star and the string representation of the party.)

In the first example above, the visualization should look something like, assuming no colour was set:
*********|* PartyName (53% of votes, 208 seats)

In the second example above, the visualization should look something like, assuming Cyan is the colour:
***** | AnotherPartyName([0,255,255], 36% of votes, 25 seats)

An example where a party is expected to win all seats, would like something like:
*****|***** My Party (95% of votes, 200 seats)

- *textVisualizationByVotes* is like the previous method but it creates a visual representation to represent the percentage of votes the party is expected to win. The total possible is of course 100% of the votes. You can find the expected arguments for this method in the class diagram at the start of this document. Make sure you avoid code duplication. Any code that this method and the previous method have in common, should be placed in a separate method that is called by both.

You may add any private methods you wish to support the public methods you are required to create. If a method gets very long and no longer fits on a screen (including a laptop screen using a reasonable font) you should create such helper methods that are declared private.

Poll

This class represents a single poll. The following **instance variables** should all be declared private.

- *name* of type String. The class should have a getter method for this instance variable but no setter method. (The name must be set using the constructor and can't be changed after construction.) Any string is considered a valid string for a name.
- *parties* whose type is an array of Party objects (Party[]). The type must be an array, do not use an ArrayList or other object from Java Collections framework. There should be no (direct) setter method but do provide a getter method. (It is not required to prevent privacy leaks for this instance variables.
- *partiesInPoll* of type int represents the number of unique parties that have been added to the poll to date. (It should **not** represent the size of the parties array.) The class should have a getter method for this instance variable but no setter method.

Create **one constructor** for the class that takes the name of the poll and the maximum number of Parties this poll will take as an argument. The maximum number of parties should be at least 1. If an invalid number is provided, set the maximum number of parties this poll can contain to 10.

In addition to the instance variables and their getter and setter methods, also add the following **public methods** to the Poll class. (These are also noted in the class diagram above.)

- *toString* which does not take any arguments and returns something of type String. The string returned should have multiple lines.
    - The first line is the name of the poll.
    - This is followed by one line per party in the poll which contains the string representation of the party (retrieved by called toString on each party).

  In other words, the format is '<name> <newline> <string representation of first party> <newline><string representation of second party><newline>...' Note that anything between the angle brackets (<>) describes information that should be placed there. Everything else is the literal character that should be placed there.
- *addParty* which does not return anything and takes an object of type Party as an argument. If the argument is null, print an error message. Parties are uniquely identified by their name. Only one party by a given name (not case sensitive) can be in a poll at a time.
    - If the party to add has the same name as another party already in the parties array, the existing party should be replaced by the party provided as an argument.
    - If no party with the same name exists in the poll, this party should be added at the end of the list and the partiesInPoll instance variable should be incremented by one.
    - If there is no room left in the parties array, print an error message that the poll is full and no further parties can be added. Do NOT increment partiesInPoll in this case.
- *getParty* which takes the name of the party to find and returns the party in the poll with that name. If no party of that name exists in the poll this method should return null.
- *textVisualizationBySeats* returns a String that gives a visual representation of the data relevant to this poll. It is very similar to the toString method, but instead of using the toString method for each party in the parties array, use the textVisualizationBySeats method instead. This method takes two arguments that need to be passed to the party's textVisualizationBySeats method. See the class diagram for the information about the two arguments.

- *textVisualizationByVotes* returns a String that gives a visual representation of the data relevant to this poll. It is very similar to the toString method, but instead of using the toString method for each party in the parties array, use the textVisualizationByVotes method instead. This method takes two arguments that need to be passed to the party's textVisualizationBySeats method. See the class diagram for the information about the two arguments.

Recall that the creation of an array of objects will fill the array with null values. When looping through the array of Party objects, make sure you check if the party is null or not. Make sure to handle null values appropriately. Sometimes a null value means 'do nothing'. Other times it means an error. It can also mean that it is a good place to insert a new Party in the array.

If an error is encountered, print an error message. Do NOT end the program. Instead allow the program to continue running as if no error had occurred. A later iteration will manage errors better.

You may add any private methods you wish to support the public methods you are required to create. If a method gets very long and no longer fits on a screen (including a laptop screen using a reasonable font) you should create such helper methods that are declared private.

This class uses the Party class but the code that tests the Poll class does not depend on Party being correctly implemented. Instead, the test using a Mock Party object. If you are interested to learn more about that, search for information about unit testing and mock objects. For this assignment, it is important that you **do NOT modify any code in the Party class**: it will not help you pass any tests since the test is using a Mock Party object which overrides code from the Party class. (You will learn more about overriding code later in the semester.)

PollList

This class represents a collection of polls that all collect data for the same election. The following **instance variables** should all be declared private.

- *polls* which is an array of type Poll (Poll[]). (The type must be array, do not use ArrayList or other type from the Java Collections framework.) There should be a getter method but no setter method for this instance variable. **The getter method should be called toArray, not getPolls.** This instance variable will change using other mutator methods instead of a setter method. You do not have to prevent privacy leaks for this instance variable.
- *numOfSeats* of type int which represents the number of seats available in the election covered by the polls. There should be a getter method for this instance variable but no setter method.

The class should also contain a constant called MAX_STARS_FOR_VISUALIZATION of type int with value 18.

Create **one constructor** for the class that takes the number of polls this list should be able to contain and the number of seats that are available in the election covered by the polls in the list. The number of polls should be at least 1. If an invalid number is provided, set the number of polls to 5. The number of seats should be at least 1 as well. If an invalid number is provided, set the number of seats to 10.

In addition to the instance variables with required getter and setter methods, also add the following **public methods** to the PollList class. (These are also noted in the class diagram above.)

- *addPoll* which does not return anything and takes an object of type Poll as an argument.
    - If the argument is null, print an error message and leave the list unchanged.
    - If there is no room left in the polls array, print an error message that the list is full and no further polls can be added.
    - If there is room in the array, put the poll in the first open slot in the array.
- *getAveragePartyData* which returns an instance of Party and takes as argument the name of a Party.
    - The method should create a new Party object. The name of this new party is the name provided as an argument.
    - The method should then calculate the expected number of seats and expected percentage of the vote for this new Party object as follows before returning it.
        - Seats: This should be the average number of seats this party is expected to win over all polls in the list. If the party is not included in one of the polls, ignore this poll in the computation (do not assume that the expected number of seats is 0).
        - Votes: The average expected percentage of votes this party is expected to win over all polls in the list. If the party is not included in one of the polls, ignore this poll in the computation (do not assume that the expected percentage of votes is 0).
    - To get the info about a party from a poll, use the getParty method in the Poll class. Make sure that you call this method at most once for each party in each poll. **Store the result of the method call in a variable rather than calling the method multiple times!** (If you call the method multiple times for the same party in the same poll, the tests will fail.)
- *getAggregatePoll* which returns a poll representing the aggregate of all polls in the list. The method takes as an argument the names of all the parties as an array of Strings that should be included in the aggregate poll. Consider the following when creating this aggregate poll:
    - The name of the poll should be ‘Aggregate’.

    - Each party should be the party created by the method getAveragePartyData.
- *textVisualizationBySeats* which provides a text visualization for each poll in the list. Each poll visualization should be separated by a newline. For each poll in the list, get the text visualization by calling the textVisualizationBySeats method in the Poll class. (Do NOT modify this method (or any other method) in the Poll class. Just use the string that is returned by this method, whether the string is correct or not.) This method in the Poll class needs two arguments:
    - The maximum number of stars to use in the visualization. Use the value stored in the constant MAX_STARS_FOR_VISUALIZATION.
    - The number of seats that are represented by a single star. This should be calculated based on the number of seats in the election (which is stored in instance variable numOfSeats). Note that MAX_STARS_FOR_VISUALIZATION represents all seats. For example, if we had 10 stars and 200 seats, then each star would have to represent 20 seats. Make sure to always round this up.
- textVisualizationByVotes which is very similar to textVisualizationBySeats but it uses the textVisualizationByVotes in the poll class instead of textVisualizationBySeats. This method also takes two arguments:
    - The first remains the same, the maximum number of stars to use in the visualization.
    - The second indicates the percent of votes that are represented by a star. In this case, the total percent is always 100%, so the maximum number of stars always represents 100%.
- *toString* which does not take any arguments and returns something of type String. The string returned should have the format: 'Number of seats: <numOfSeats>' followed by a newline, followed by the string that is returned by textVisualizationBySeats. Note that anything between the angle brackets (<>) describes information that should be placed there. Everything else is the literal character that should be placed there.

Recall that the creation of an array of objects will fill the array with null values. When looping through the array of Poll objects, make sure you check if the poll is null or not. Make sure to handle null values appropriately. Sometimes a null value means 'do nothing'. Other times it means an error. Or it can mean that you've found an empty space to insert a new poll. If an error is encountered, print an error message. Do NOT end the program. Instead allow the program to continue running as if no error had occurred. A later iteration will manage errors better.

You may add any private methods you wish to support the public methods that you are required to create. If a method gets very long and no longer fits on a screen (including a laptop screen using a reasonable font) make sure to break it up over multiple methods.

This class uses the Party and Poll class but the code that tests the PollList class does not depend on Party being correctly implemented. Instead, the test using Mock Party and Poll objects. If you are interested to learn more about that, search for information about unit testing and mock objects. For this assignment, it is important that you **do NOT modify any code in the Party or Poll class**: it will not help you pass any tests since the test is using a Mock Party and Poll objects which overrides code from the Party and Poll classes respectively. (You will learn more about overriding code later in the semester.)