Colin Kincaid
CS 106A

Assignment #6
August 6, 2018

Assignment #6 — BiasBars

Due: 11:00AM PDT on Tuesday, August 14th

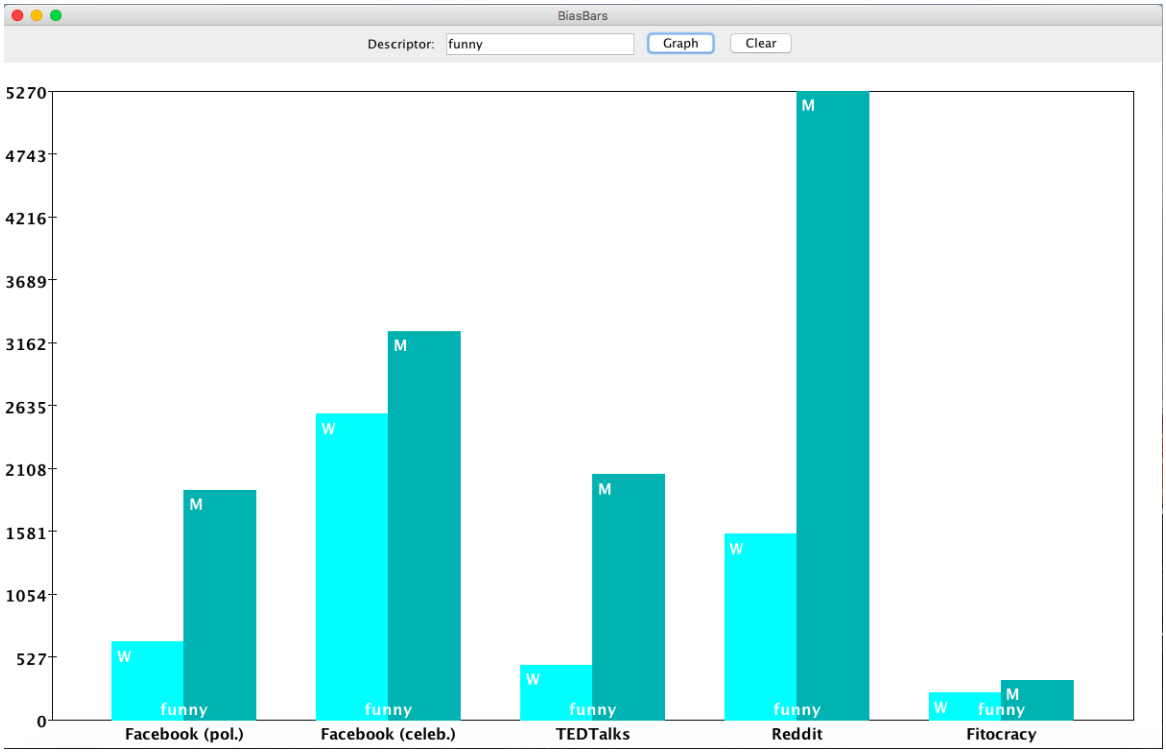
This assignment may be done in pairs (which is optional, not required)
Note: NO late days (free or otherwise) may be used on this assignment

---

The BiasBars assignment was devised by Jennie Yang and Monica Anuforo and inspired by NameSurfer, which was created by Nick Parlante and tweaked by other CS106A instructors over the years.

This assignment will give you a chance to build a program using Java interactors including buttons, text fields, and a resizable graphical display—something that resembles an application you would find on your own computer. The result will be an application that, unlike Breakout or Snowman, is not a game but rather a useful program that presents data on important social questions.

Figure 1. Sample run of the BiasBars program using the word "funny"



Much of today's work in artificial intelligence involves natural language processing, a field which studies the way language is used today and has been used in the past. The datasets we use to train artificially intelligent systems are collections of text that humans have written. If there are imbalances in how different groups of people tend to be described, then our machines will pick up on and potentially amplify those imbalances. Extreme manifestations of these biases like Tay, Microsoft's 2016 chatbot infamous for

tweeting racist and anti-Semitic statements after just a day of learning from anonymous posts on the Internet, magnify the importance of understanding the ways we use language.

Even when people do not mean to be malicious, their language can still exhibit biases that influence how our machines learn. For example, when history of science professor Londa Schiebinger attempted to Google Translate a Spanish article written about her, all of the pronouns became "he" and "him" rather than "she" and "her" simply because masculine pronouns were more common than feminine pronouns in the available data. In a later study, Schiebinger found more insidious translation errors that assumed genders for people of certain professions, based on the frequency of word usage in gendered languages such as German[1]. The software engineers who made Google Translate probably did not mean for this to occur; they probably did not even account for that possibility as they were designing their translation algorithm. The moral of the story? To prevent these kinds of slip-ups, computer scientists need to consider the social and ethical impacts of their work *beforehand*.

We hope that by introducing these sorts of topics early in computer science education, we can help the next generation of software developers and computer science researchers—which might include you!—be more mindful of the potential social implications of their work.

A Brief History of BiasBars

BiasBars is a spinoff of NameSurfer, a past CS106A assignment that asked students to graph data about the popularity of baby names over time. For an ethics-themed hackathon in April 2018, Jennie and Monica decided to create a parody of NameSurfer that instead graphed information about gender and RateMyProfessor reviews, based on an applet[2] created by history professor Ben Schmidt of Northeastern University. (The data you will be using for this assignment will be similar, but not from the same source.) When Colin heard about the project, he decided to help bring it to life!

Overview of RtGender

The advent of social media has given us unprecedented access to celebrities, politicians, and others who would otherwise be complete strangers. The indirectness and anonymity of being behind a computer or phone screen also gives people a sense of security to say whatever they want, which can range from the supportive or constructive to the downright offensive or harmful. But can the addressee's gender influence how we choose to speak about them?

In 2018, a team of researchers from Stanford, Michigan, and Carnegie Mellon compiled a dataset called Responses to Gender (RtGender)[3], comprising comments from popular Facebook pages, Reddit, TED Talks, and Fitocracy in order to study just that. From this corpus, we can extract data about the kinds of language typically used when describing women and men on the Internet.

---

[1] https://genderedinnovations.stanford.edu/case-studies/nlp.html

[2] http://benschmidt.org/profGender/

[3] For further analysis of the RtGender dataset, see the research team's paper (presented at the 2018 International Conference on Language Resources and Evaluation) here: https://nlp.stanford.edu/robvoigt/rtgender/rtgender.pdf

There are a few details to note about this dataset:

1. Comments on Facebook posts and TED Talks are "broadcast," in that they are addressed to people in positions of authority for a wider audience to see. In contrast, comments from Reddit and Fitocracy typically come from person-to-person interactions. This fundamental difference can affect how people think or speak about the recipient of their comments.
2. The gender of the commenter can also have an impact on what language they use, so analysis based purely on the gender of the comment recipient does not tell us everything we might want to know. Nonetheless, it can still give us some understanding of the biases that might exist when people use social media.
3. In this dataset, gender is the only piece of information we have about these people's social identities; it does not include data on other salient identities such as race and ability. Furthermore, gender is only classified into the categories of woman and man, which means non-binary people are unfortunately not represented. In the future, it would be great to have data about all sorts of people (maybe you can help make that happen!). For now, we at least have a place to start.
4. The data visualized in this applet should not be seen as indicative of any trends. The number of responses to women's posts is not the same as the number of responses to men's posts, and we ask you to consider the data as a count of the number of occurrences of words rather than a true "frequency" (a frequency is the number of occurrences per some number of words; we use that term throughout the assignment for simplicity, but we really mean a count). Moreover, this dataset categorizes responses based on the gender of the original poster rather than the subject of each descriptor, so it does not exactly work for our intended purpose of studying how people describe women and men differently. The dataset also does not take into account the fact that many of these descriptors are preceded by the word "not". To visualize trends more accurately, see the original applet upon which this assignment is based (link in Footnote 2).

The BiasBars Project

Your task is to take pre-processed data from the RtGender dataset and create a program that allows you to visualize how different descriptors are used for women and men on the Internet. This visualization will take the form of a *bar graph*; for the particular descriptor you are observing, different bars will show the relative frequency with which women and men are described with that word. We have laid out some milestones to guide you as you tackle this project:

Milestone 0: Understand the Starter Files

All of the data you will need for this assignment can be boiled down to a text file that looks like this:

gender-data.txt

```
fair W 1018 1240 209 1680 95 M 3155 380 821 6195 138
inevitable W 59 29 123 77 10 M 170 14 467 287 15
different W 1318 1660 3012 5019 410 M 3789 1012 8768 16638 471
...
```

Each line of the file begins with a word (typically an adjective). This descriptor is followed by the letter "W[4]" and the number of occurrences of that word in comments describing women from each social media platform. The line ends with the letter "M" and the word frequency data associated with men.

The different social media sites that act as the sources for our comments can be found in a constant array of Strings called `COMMENT_SOURCES`, which is defined in the **`BiasBarsGraph`** class (more about this later). The comment sources are

1. Facebook pages of politicians
2. Facebook pages of other celebrities
3. The TED website
4. Reddit
5. Fitocracy

In each line of the file, the five frequencies that appear after "W" and the five frequencies that appear after "M" correspond, in order, to these five sources of comments. So, the first frequency listed will be from politicians' Facebook pages, the second frequency will be from celebrities' Facebook pages, and so on.

Once you have completely finished the BiasBars assignment, you should be able to visualize the data for particular words as shown in Figure 1.

To give you more experience working with classes that interact with one another, the BiasBars application as a whole is broken down into several class files, as follows:

- **`BiasBars`** — The main program class that ties together the application. It is responsible for creating the other objects and for responding to the buttons at the top of the window, but only to the point of redirecting those inputs to the objects represented by the other classes.
- **`BiasBarsEntry`** — Encapsulates all the information for a particular descriptor. Given a **`BiasBarsEntry`** object, you can find out what descriptor it corresponds to, as well as the number of times it occurs for women and men across each of the internet platforms.
- **`BiasBarsDatabase`** — Stores all of the data from the source file in a structure better suited for the needs of the program. This class is completely separate from the user interface. It is responsible for reading in the file and for locating the data associated with a particular descriptor.
- **`BiasBarsGraph`** — A subclass of **`GCanvas`** that displays the graph for a descriptor by arranging the appropriate **`GObject`**s on the screen, just as with the various graphical programs you've written this quarter. Includes the `COMMENT_SOURCES` array as well as the values and dimensions needed to plot your graphs correctly.

In each of these classes, you must implement certain **public** methods as outlined in the milestones below. In the starter code, all of these methods are already included as *stubs*. Stubs are methods that will eventually become part of the program structure but that are temporarily unimplemented. They play a very important role in program development because they allow you to set out the structure of a program even before you write most of the code. As you implement the program, you will go through the code and replace stubs with real code as you need it.

[4] We choose to describe the two genders included in this dataset as "woman" and "man" rather than "female" and "male," as the former terms refer to gender and social role whereas the latter typically refer to sex assigned at birth.

You may add extra **private** methods where you find them helpful, but you may not add any **public** methods other than the ones specified in the milestones below.

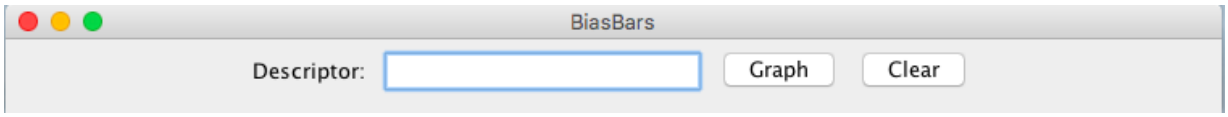
Part of writing code with good style on this assignment is properly **separating responsibilities** between these classes as outlined below, and choosing **the appropriate data structure(s) to use**. Although the class structure may sound complicated, the scale of the project is comparable to previous assignments. That being said, we encourage you to get started early and use the following milestones.

Milestone 1: Assemble the GUI interactors

Your first milestone is to add the interactors to the window to detect button clicks and read what's in the text field. If you look at the top of Figure 2 below, you will see that the region along the **`NORTH`** edge of the window contains the following interactors, from left to right:

- A **`JLabel`** with the text "Descriptor: ".
- A *`TEXT_FIELD_WIDTH`*-wide **`JTextField`**, initially blank, for typing in words.
- A **`JButton`** labeled "Graph". Clicking this button (*or pressing ENTER on the text field*) should cause the program to graph the frequency data for the currently typed descriptor.
    - The program is **case-insensitive**: "nice", "NICE" or "NiCe" should all correctly graph information for the word "nice".
    - If there is no frequency data about that descriptor, the program should not change the currently displayed graph.
- A **`JButton`** labeled "Clear". Clicking this button should cause the program to clear the currently graphed data.

Figure 2. Close-up of interactors used in BiasBars user interface



The simplest strategy to check whether your program is working is to change the definition of the **`BiasBars`** class so that it extends **`ConsoleProgram`** instead of **`Program`**, at least for the moment. You can always change it back later. Once you have made that change, you can then use the console to record what is happening in terms of the interactors to make sure that you've got them right. For example, Figure 3 below shows a possible transcript of the commands used to generate the output from Figure 2, in which the user has just completed the following actions:

1. Entered the word **`nice`** in the text field and clicked the **Graph** button.
2. Clicked the **Clear** button.
3. Entered the word **`smart`** in the text field and then typed the ENTER key.

Figure 3. Illustration of Milestone 1

The hard part about reaching this milestone is understanding how interactors work. Once you do, you will be able to accomplish the milestone in a relatively small amount of code.

Milestone 2: Implement the `BiasBarsEntry` class

The **`BiasBarsEntry`** class encapsulates the information pertaining to one descriptor for both women and men. That information consists of three parts:

1. The descriptor itself, such as "nice" or "smart"
2. The number of times that descriptor appears in comments describing women for each source
3. The number of times that descriptor appears in comments describing men for each source

Within the class, you are required to implement the following constructor and methods:

```
public BiasBarsEntry(String dataLine)
```

In this constructor you should initialize the state of a new entry from the given line of data. You should assume that the line of data is from the **`gender-data.txt`** file shown previously, such as:

```
nice W 2031 10179 1077 3338 2311 M 4606 2926 3274 9603 2682
```

Note that the number of comment sources might not be five; **the `BiasBarsEntry` class has no knowledge of how many frequency entries it will need to store**, just that there will be some number of frequencies for women after the "W" and some number for men after the "M".

The implementation of the constructor has to parse the line and store all of this information as the private state of the object, in such a way that it is easy for **`getDescriptor`**, **`getFrequencies`**, and **`getMaxFrequency`** to return the appropriate values.

```
public String getDescriptor()
```

In this getter method, you should return the entry's descriptor as it was read from the input data passed in when it was created. For example, given the example line in the constructor description above, **`getDescriptor`** would return **`"nice"`**.

```
public ArrayList<Integer> getFrequencies(char gender)
```

This method should return a list of the frequencies stored in a descriptor's entry for a given gender, where the frequencies are in the same order that they were in the original file. For example, given the example line in the constructor description above, **`getFrequencies('W')`** would return **`[2031, 10179, 1077, 3338, 2311]`**, and **`getFrequencies('M')`** would return **`[4606, 2926, 3274, 9603, 2682]`**. If the character passed in is not 'W' or 'M', **you should return null.**

```
public int getMaxFrequency()
```

This method should return the highest frequency among all the frequencies across both genders for this entry. This will help with graphing later on.

```
public String toString()
```

In this method, you should return a human-readable string representation of an entry's data. The format must list the descriptor followed by lists of that word's frequencies for both women and men , separated by commas. For example, the **`BiasBarsEntry`** for "nice" would return the following string:

```
nice: {W=[2031, 10179, 1077, 3338, 2311], M=[4606, 2926, 3274, 9603, 2682]}
```

You must *exactly* match this output in your implementation.

Hint: instead of manually generating the list of frequencies portion of the string yourself, see if there is an already-implemented method we know of that can help create this string for you.

To show that you have implemented **`BiasBarsEntry`** correctly, you might want to write a short test program that creates an entry from a specific string and then verifies that the other methods work as they are supposed to. To do so, you can (temporarily) change **`BiasBars`** from extending a **`Program`** to a **`ConsoleProgram`**, and then use the console to verify that your **`BiasBarsEntry`** is behaving as expected.

Milestone 3: Implement the `BiasBarsDataBase` class

The next step is to define a new type of object called a **`BiasBarsDatabase`** that will manage the entire database of descriptors. Within the class, you are required to implement the following constructor and method:

```
public BiasBarsDatabase(String filename)
```

In this constructor, you should initialize the state of a new database and read in the data from the given data filename such that all the data is stored within the database object and can be easily returned as needed from the **`findEntry`** method (see below).

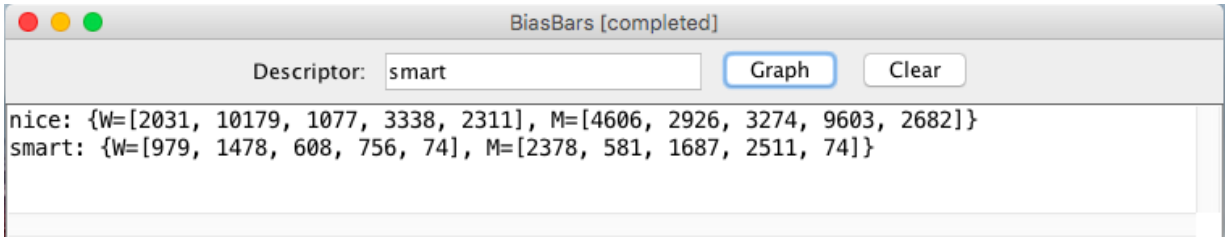
```
public BiasBarsEntry findEntry(String descriptor)
```

This method takes a name, looks it up in the database (note that this method should be case-insensitive; the name can be passed with any capitalization) and returns the **`BiasBarsEntry`** for that descriptor, or **`null`** if that descriptor does not appear.

For this class, think about different ways you can store data within the database using variable types we have discussed, and which might be the most appropriate here given the structure of the data and what the database needs to do with it.

To test this part of the program, you can add a little code to the **`BiasBars`** program so that it creates a **`BiasBarsDatabase`**, and then change the code for the interactors so that clicking the "Graph" button (or pressing ENTER in the text field) looks up the current descriptor in the database and then displays the corresponding entry (using its **`toString`** method), as shown in Figure 4 below.

Figure 4. Illustration of Milestone 3



The code for this part of the assignment should not be very long. The challenge lies in figuring out how to represent the data so that you can implement **`findEntry`** as simply and as efficiently as possible.

Milestone 4: Draw the background of the `BiasBarsGraph` class

The next step in the process is to begin the implementation of the **`BiasBarsGraph`** class, which is responsible for displaying a graph in the window using information from the database. There are a couple of important items in the **`BiasBarsGraph`** starter file that you should be aware of:

1. This class extends **`GCanvas`**, which means that we can call any **`GCanvas`** methods that you've already learned, like **`add`** or **`remove`**, from within **`BiasBarsGraph`**.
2. The starter file includes a tiny bit of code that monitors the size of the window and calls the **`update`** method whenever the size changes. This code requires only a couple of lines to implement, but would be hard to explain well enough for you to implement on your own. So, we have taken care of it for you; you're welcome!

To start the process of adding the graphing code, go back to the **`BiasBars`** class and change its definition so that it extends **`Program`** rather than the temporary expedient of extending **`ConsoleProgram`** (if you were using that for debugging). At the same time, you should remove any test code from the earlier milestones. Then, create a new **`BiasBarsGraph`** and add it to the screen.

If you run the program with only these changes, it will show a blank graph and not actually display anything on the screen. To create the graph, you need to implement the **update** method, as well as any other private helper methods you find appropriate.

To begin the implementation of your **update** method, write the code that draws the blank graph. For this, you need to create the axes, y-axis tick marks, labels for the frequencies along the y-axis, and labels for the comment sources along the x-axis, as follows:

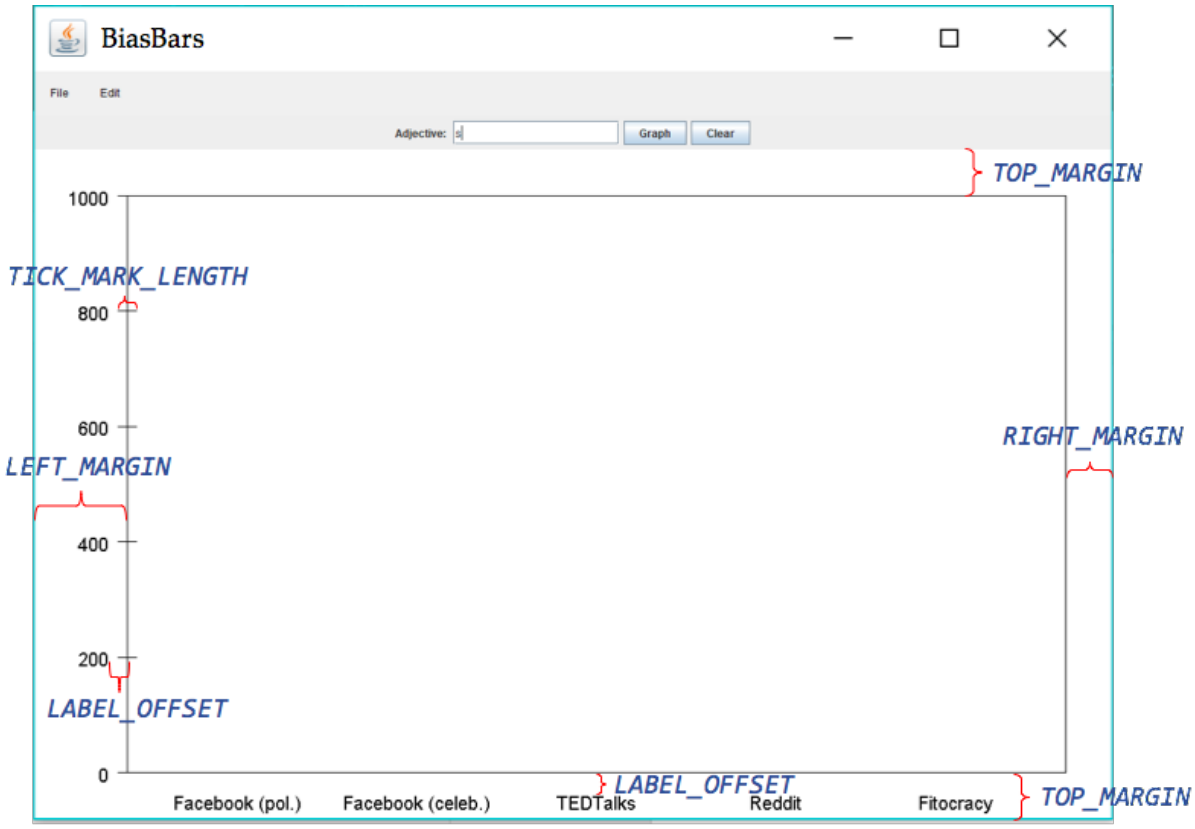
- The boundaries of the graph should be drawn using a **GRect,** which is offset by *LEFT_MARGIN* and *RIGHT_MARGIN* from the left and right of the window, respectively, and offset by *TOP_MARGIN* from both the top and bottom of the window. The left edge of the graph corresponds to the y-axis and the bottom edge to the x-axis.
- There should be *NLABELS* tick marks evenly spaced along the y-axis. Each tick mark should be represented by a **GLine** with length *TICK_MARK_LENGTH* and horizontally centered on the y-axis.
- For each tick mark on the y-axis, you should add a label representing frequency, which should be vertically centered on the corresponding tick mark. The label for the lowest tick mark should have value 0 and the label for the highest tick mark should have a value of *DEFAULT_MAX_FREQ* for now, although eventually you will scale it according to the data being shown in the graph. The remaining labels should be evenly spaced between 0 and the value of the highest label.
- For each source in *COMMENT_SOURCES*, you should add a label that displays the name of the source. The centers of these labels should be evenly spaced across the x-axis.
- Labels on the y-axis must be positioned so their right edge is *LABEL_OFFSET* distance from the y-axis. Labels on the x-axis must be positioned so the top of the label is *LABEL_OFFSET* distance below the x-axis. You should set the font of all labels to be bold and size 16, which you can do with the following call to **setFont**:

  ```
  label.setFont("*-BOLD-16");
  ```

Figure 5 is a diagram of the blank graph with all of the necessary margins, offsets, and other lengths marked. The dimensions you will need to know for drawing the axes can be found in **BiasBarsGraph**. (Note that the diagram is NOT to scale; the dimensions have been enlarged for illustration purposes, and there are fewer tick marks and y-axis labels than you will have.)

If you are having trouble getting the right coordinates for the tick marks or labels in your graph, try printing the x/y coordinates to verify them. You can't generally use **println** statements in classes other than your main program; however, if you use **System.out.println** instead, you will see the printed messages in the bottom Eclipse console. Getting the positioning perfect should also not be your top priority; first focus on making a functioning graph that looks roughly correct, and then work on the precise math for spacing and centering.

Figure 5. Diagram of a blank BiasBarsGraph



Milestone 5: Complete the `BiasBarsGraph` class

In addition to creating the axes for the graph, the **`update`** method in **`BiasBarsGraph`** also has to plot the actual data values. In **`BiasBars`**, data is shown using bar graphs. The x-axis separates the data by comment source. Along the y-axis is the number of occurrences of the selected descriptor.

There are a couple of points that you should keep in mind while implementing this milestone:

- Each comment source will have its own section on the x-axis, with a corresponding label, and each section will have the same **total** width for its bars (the constant *`BARS_WIDTH`*). These sections should be evenly spaced.
- For each comment source, the graph will have one bar to represent the frequency for women and one bar for men, side by side. The bars for women will be one color, and the bars for men will be another. You can choose any colors for your bars as long as the bar for women is on the left and the bar for men is on the right. To expand your range of colors, consider using the `darker` method:

  ```
  Color darkRed = Color.RED.darker();
  ```

- When you display a new descriptor, you should set the scale of your y-axis such that the top of the graph will represent the maximum frequency in the data for that descriptor (this is where the **`getMaxFrequency`** method from **`BiasBarsEntry`** comes in), rounded up to the nearest multiple

of 10 (represented by the *MAGNITUDE* constant in the **BiasBarsGraph** file). To round a frequency up to the nearest *MAGNITUDE*, you can do the following calculation, which takes advantage of integer division:

```
int highestFreqLabel = entry.getMaxFreq() / MAGNITUDE * MAGNITUDE;
if (highestFreqLabel != entry.getMaxFreq()) {
      highestFreqLabel += MAGNITUDE;
}
```

(Take a minute to think about what this snippet of code is doing!)

- The height of each bar should be calculated such that frequency 0 is at the bottom of the graph, the maximum frequency is near the top of the graph, and all other frequencies are evenly-spaced in between.
- For each bar, also display a label at the top of the bar that shows "W" if the bar represents frequency for women and "M" if for men. This gender label should have a buffer of *LABEL_OFFSET* horizontally and vertically from the top-left corner of each bar.
- Finally, for each pair of bars, display a label at the bottom of the bar that shows the descriptor being graphed. This label should be centered across the two bars, *LABEL_OFFSET* above the bottom of the bar.

The **BiasBarsGraph** class includes a few methods for specifying which entry is displayed on the screen and modifying the display. You've already started working on **update**, but you'll need to complete its implementation at this point. The **addEntry** method supplies the graph with a **BiasBarsEntry** whose information it should plot when the graph is updated. The **clear** method removes that entry so that nothing is displayed when the graph is updated. You will need to implement all three of these methods in order to make the graph work as expected.

It is important to note that neither **addEntry** nor **clear** actually changes the display. To make changes in the display, you need to call **update**, which deletes any existing **GObjects** from the canvas and then reassembles everything onto the display. At first glance, this strategy might seem unnecessary. It would, of course, be possible to have **addEntry** just add the objects necessary to draw the graph.

The problem with that approach is that it would no longer be possible to reconstruct the entire graph. For example, you need to recreate the graph whenever you change the size of the display, which wouldn't be possible without knowing the magnitude of the frequencies that need to be shown on the graph. By storing the current entry internally, the **BiasBarsGraph** class can redraw everything when **update** is invoked from the **componentResized** method.

Note: COMMON BUG – Some students encounter a bug where the width and height of their **BiasBarsGraph** are 0. This will happen if your code tries to ask for the size of your graph before it's been added to the screen. This may happen in two places; in **BiasBars**'s **init** method (since the program has not yet launched), or in the **BiasBarsGraph** constructor (since, if you're still creating the graph, there's no way it's been added to the screen yet!). Instead, make sure to only ask for the dimensions of the graph once it has been added to the screen.

The final step in completing your program is to actually call the graph's public methods in the **`BiasBars`** class so that the user's requests to graph and clear entries are executed. Once you have that working, congratulations! You just built a Java applet that takes user input through a GUI and works with a large amount of real-world data.

Optional Extensions

There are many possibilities for optional extra features that you can add if you like, potentially for a small amount of extra credit. If you are going to do this, please *submit two versions of your program*: one that meets all the assignment requirements, and a second extended version (see the FAQ on the Eclipse page for how to create new files in your project). At the top of your extended files, in your comment header, you must **comment** what extra features you completed. Here are a few ideas:

- *Support for showing multiple descriptors at once*
  - You might want to allow your **`BiasBarsGraph`** to support displaying the data for multiple descriptors at once. Because each comment source has a set width, you'll need to divide the allotted width by the number of bars needed in order to determine how wide the individual bars should be. The y-axis should continue to scale to the maximum frequency represented on the graph rounded up to the nearest `MAGNITUDE`. An extension upon this extension would be to allow deletion as well as addition.
- *Toggling between bar graph and other sort of graph*
  - You might consider allowing your graph to show the data with more than just bars. Try other forms of data visualization and see what you like best.
- *Changing colors*
  - Try modifying your graph such that every time you graph a new descriptor, the graph changes its colors according to a particular sequence. First, add a constant array of colors to **`BiasBarsGraph`**. Then, you'll want some way of figuring out how many different graphs have been shown in order to cycle through the color array.
- *Plot the data differently*
  - What other information about the descriptor frequencies could you display? Consider adding the ability to view the most / least frequently used descriptors, correlation between descriptors, or other interesting insights that aren't immediately apparent.

Grading

Functionality: Your code should compile without any errors or warnings.

Style: Follow style guidelines taught in class and listed in the course Style Guide. For example, use good procedural decomposition into methods to indicate structure and avoid redundancy. Use descriptive names for variables and methods. Format your code using indentation and whitespace. Avoid redundancy using methods, loops, and factoring. Minimize the use of private instance variables, preferring local variables as much as possible. Use descriptive comments, including at the top of each .java file, atop each method, inline on complex sections of code, and a citation of all sources you used to help write your program. If you complete any extra features, list them in your extra files' comments to make sure the grader knows what you completed.

Honor Code: Follow the Honor Code when working on this assignment. Submit your own work and do not look at others' solutions (outside of your pair, if you are part of a pair). Do not give out your solution. Do not search online for solutions. Do not place a solution to this assignment on a public web site or forum. Solutions from this quarter, past quarters, and any solutions found online, will be electronically compared.

Final Remarks

We hope you enjoy this assignment! :-) If you're interested in learning more about natural language processing and computer science, check out the resources linked in this handout, talk to Colin, Annie, Jennie, or Monica, or take a look at CS124 at Stanford!

Acknowledgements

- Rob Voigt, Vinodkumar Prabhakaran, Dan Jurafsky of Stanford, David Jurgens of University of Michigan, Yulia Tsvetkov of CMU for creating the RtGender dataset and graciously allowing us to use it for this assignment
- Ben Schmidt of Northeastern for creating the "Gendered Language in Teaching Reviews" applet, which inspired this assignment
- Londa Schiebinger of Stanford for inspiring us to explore the relationship between gender, language, and software in the first place
- Cynthia Lee and Keith Schwarz for their enthusiasm about our project
- Colin Kincaid for finding the RtGender dataset for us to use and rigorously editing the handout
- Annie Hu for transforming the RtGender dataset into a format suitable for this assignment
- Kashif Nazir for suggesting the assignment's final name, as well as the runner-up: GenderRender