Assignment #6—NameSurfer
Due: 10AM PST on Tuesday, August 13

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

Created by Nick Parlante; revised by Chris Piech, Nick Troccoli, M Stepp, P Young, E Roberts, M Sahami, K Schwarz and B Capoor.

Sandcastle: Election Results

It's election season at Stanford! You've been given a **`ArrayList`** of **`String`**s representing all the votes by the Stanford community for student president. For example, this **`ArrayList`** might look like this:

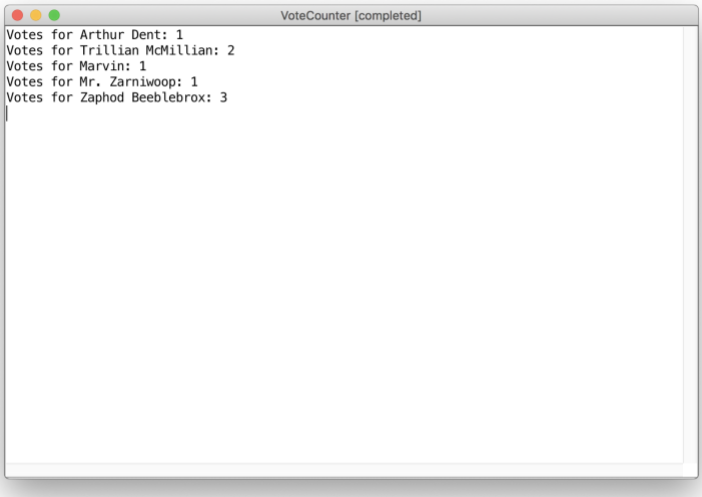
```
{"Zaphod Beeblebrox", "Arthur Dent", "Trillian McMillian", "Zaphod Beeblebrox", "Marvin", "Mr. Zarniwoop", "Trillian McMillian", "Zaphod Beeblebrox"}
```

In this list, each element represents a single person's choice of candidate for president. Note that the list can be of any positive length and that there can be an arbitrary number of candidates, although each candidate will have a unique name.

Your task is to write the following method:

```
private void printVoteCounts(ArrayList<String> votes)
```

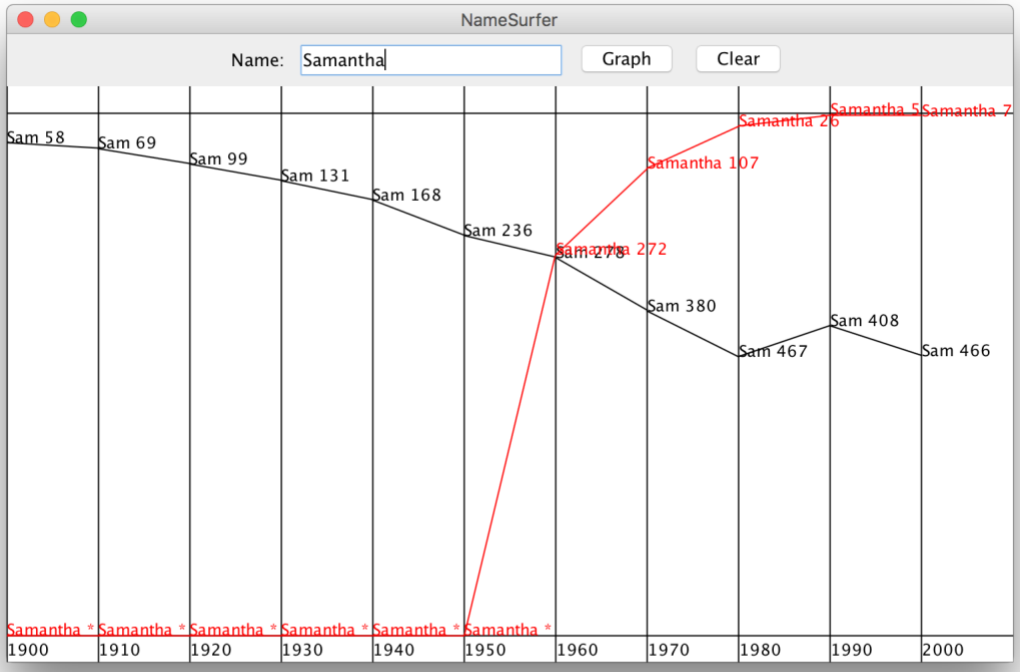
that takes in a list such as the one above and outputs the number of votes that each candidate received. For example, given the list above, the program would output the results displayed on the right. You may output the votes per candidate in any order.



NameSurfer

NameSurfer is a program that graphs the popularity of US baby names from 1900 through 2000. It lets the user analyze interesting trends in baby names over time, and gives you practice with classes, data structures and interactors to create a large-scale application.

Figure 1. Sample run of the NameSurfer program (with names "Sam" and "Samantha")



You will work in the following files: `NameSurfer.java`, `NameSurferEntry.java`, and `NameSurferDatabase.java`. We provide a completed demo program on the course website. Additionally, there is a built-in output comparison tool to check your output. If you click `File` -> `Compare Output`..., you can compare the image displayed on your graph with provided sample output images in the `output/` folder. You can also save output from the demo to compare to by going to `File->Save`.

This assignment may be done in **pairs**, or may be done individually. **You may only pair up with someone in the same section time and location**. If you work as a pair, **comment both members' names** on top of every .java file. Make **only one** assignment submission; do not turn in two copies.

Overview

Every year, the Social Security Administration releases data about the 1000 most popular names for babies born in the US at **http://www.ssa.gov/OACT/babynames/**. The data can be boiled down to a single text file that looks something like this:

names-data.txt

```
...
Sam 58 69 99 131 168 236 278 380 467 408 466
Samantha 0 0 0 0 0 0 272 107 26 5 7
Samara 0 0 0 0 0 0 0 0 0 0 886
Samir 0 0 0 0 0 0 0 0 920 0 798
Sammie 537 545 351 325 333 396 565 772 930 0 0
Sammy 0 887 544 299 202 262 321 395 575 639 755
Samson 0 0 0 0 0 0 0 0 0 0 915
Samuel 31 41 46 60 61 71 83 61 52 35 28
Sandi 0 0 0 0 704 864 621 695 0 0 0
Sandra 0 942 606 50 6 12 11 39 94 168 257
...
```

Each line of the file begins with the name, followed by the rank of that name in each of the 11 decades since 1900: 1900, 1910, 1920, and so on, up to 2000. A rank of 1 indicates the most popular name that year, while a rank of 997 indicates a name that is not very popular. A 0 entry means the name did not appear in the top 1000 names for that year. The elements on each line are separated from each other by a single space. The lines happen to be in alphabetical order, but nothing in the assignment depends on that fact.

As you can see from the small file excerpt above, the name Sam was #58 in the first decade of the 1900s and is slowly moving down. Samantha popped on the scene in the 1960s (possibly because the show *Bewitched*, which had a main character named Samantha, ran on television during those years) and is moving up strong to #7. Samir barely appears in the 1980s (at rank #920), but by 2000 is up to #798. The database counts children born in the United States, so trends tend to reflect the evolution of ethnic communities over time.

The goal of this assignment is to create a program that graphs these names over time, as shown in the sample run in Figure 1. In this diagram, the user has just typed **Samantha** into the box marked "Name" and then clicked on the "Graph" button, having earlier done exactly the same thing for the name **Sam**. Whenever the user enters a name, the program creates a new plot line showing that name's popularity over the decades. Clicking on the "Clear" button removes all the plot lines from the graph so that the user can enter more names without all the old names cluttering up the display.

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

- **NameSurfer**—This is the main program class that ties together the application. It has the responsibility for creating the other objects, for drawing the display and for responding to the interactors at the top of the window.
- **NameSurferEntry**—This class ties together all the information for a particular name. Given a **NameSurferEntry** object, you can find out what name it corresponds to and what its popularity rank was in each decade.

- **`NameSurferDatabase`**—This class keeps track of all the information stored in the data files, but is completely separate from the user interface. It is responsible for reading in the data and for locating the data associated with a particular name.
- **`NameSurferConstants`** *(provided)*—This interface is provided for you and defines a set of constants that you can use in the rest of the program simply by having your classes implement the **`NameSurferConstants`** interface, as they do in the starter files. You should use these constants when appropriate and your program should respond accordingly if their values are changed.

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 can go through the code and replace stubs with real code as you need it.

You may add extra **private** methods if you would like, 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**. Even though the class structure sounds 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: 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 1, you will see that the region along the **`NORTH`** edge of the window contains the following interactors, from left to right:

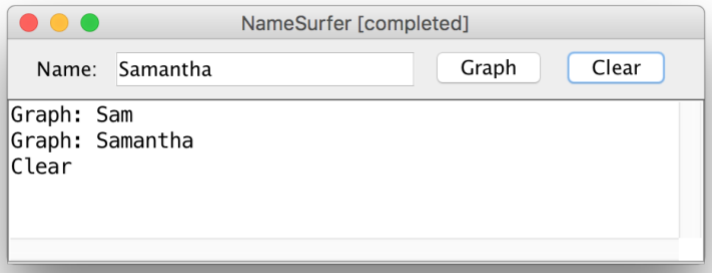
- A **`JLabel`** with the text "Name: ".
- A **`TEXT_FIELD_WIDTH`** wide **`JTextField`**, initially blank, for typing in names.
- A **`JButton`** labeled "Graph". Clicking this button (or pressing ENTER on the text field) should cause the program to graph the ranking data for the currently typed name.
    - The program is **case-insensitive**; "lisa", "LISA" or "LiSa" should all correctly graph "Lisa".
    - If there is no ranking data about that name, the program should not add a new plot line, but existing plot lines should remain displayed.
- A **`JButton`** labeled "Clear". Clicking this button should cause the program to clear all graphed name data.

The simplest strategy to check whether your program is working is to change the definition of the **`NameSurfer`** class so that it extends **`ConsoleProgram`** instead of **`GraphicsProgram`**, 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's happening in terms of the interactors

to make sure that you've got them right. For example, Figure 2 shows a possible transcript of the commands used to generate the output from Figure 1, in which the user has just completed the following actions:

1. Entered the name **Sam** in the text field and clicked the **Graph** button.
2. Entered the name **Samantha** in the text field and then typed the ENTER key.
3. Clicked the **Clear** button.

Figure 2. Illustration of Milestone 1



Milestone 2: Implement the `NameSurferEntry` class

The next step is to define a new type of object called a **`NameSurferEntry`** that will help you manage the baby name data. Specifically, each **`NameSurferEntry`** object represents the information pertaining to one name in the database. That information is:

1. The name itself, such as **`"Sam"`** or **`"Samantha"`**
2. A list of 11 values indicating the rank of that name in each of the decades from 1900 to 2000, inclusive

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

```
public NameSurferEntry(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 **`names-data.txt`** file shown previously, such as:

`Sam 58 69 99 131 168 236 278 380 467 408 466`

The constructor should divide up the line of data and store the information appropriately in the new object such that it is easy for the **`getName`** and **`getRank`** methods (see below) to return the appropriate values.

```
public String getName()
```

In this getter method, you should return the entry's name 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, **`getName`** would return **`"Sam"`**.

```
public int getRank(int decadesSince1900)
```

In this method, you should return the entry's ranking for the given number of decades after 1900. For example, given the example line in the constructor description above, **`getRank(0)`** would return 58 because the rank for 1900 is 58, and **`getRank(9)`** would return 408 because the rank for 1990 is 408. If the number passed in is outside the valid number of decades, **you should return -1.**

```
public String toString()
```

In this method, you should return a human-readable string representation of that entry's data. The format must list the person's name followed by a space and a list of their rankings, separated by commas. For example, the **`NameSurferEntry`** for Sam would return the following string:

`Sam [58, 69, 99, 131, 168, 236, 278, 380, 467, 408, 466]`

To help you implement this milestone, we encourage you to write a very simple test program (either in **`NameSurferEntry.java`** or in a new **`ConsoleProgram`** you create) that creates entries from specific strings and verifies that all methods work as intended.

Milestone 3: Implement the `NameSurferDatabase` class

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

```
public NameSurferDatabase(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).

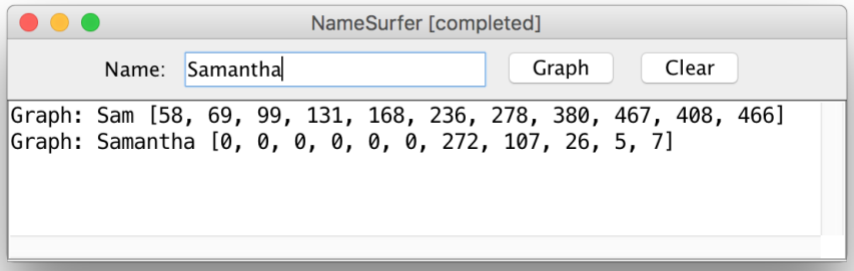
```
public NameSurferEntry findEntry(String name)
```

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 **`NameSurferEntry`** for that name, or **`null`** if that name 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 **NameSurfer** program so that it creates a **NameSurferDatabase**, 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 name 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



Milestone 4: Draw the background of the graph

The next step in the process is to continue the implementation of the **NameSurfer** class, which is responsible for both handling interactions *and* displaying the graph in the window. In this milestone, our goal is to draw the decade lines and labels that make up the background of the graph.

The starter file includes a tiny bit of code that monitors the size of the window and calls the **redraw** 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. Writing a page of description so that you could add a couple of lines seemed like overkill, particularly given that the strategy is easiest to learn by example.

To start the process of adding the graphing code, go back to the **NameSurfer** class and change its definition so that it extends **GraphicsProgram** rather than the temporary expedient of extending **ConsoleProgram**. At the same time, you should remove any test code from the earlier milestones.

If you run the program with only these changes, it won't actually display anything on the screen. To create a visual, start by implementing the **redraw** method, which will almost certainly involve defining private helper methods as well. As a first step, write the code to create the background grid for the graph, as follows:

- There should be a **GRAPH_MARGIN_SIZE** sized margin area on the top and bottom of the graph (so that the year labels are always visible)

- For each decade, draw a vertical line from the top of the window to the bottom. The lines should start at the left edge of the window and be evenly-spaced to fill the entire width of the graph.
- For each decade, add a decade label that displays that year as a string in the bottom margin area. The labels should be positioned such that their x coordinates are the same as their corresponding decade line, and their y coordinates should be such that the label baseline is `DECADE_LABEL_MARGIN_SIZE` from the bottom of the screen.

Milestone 5: Complete the Name Surfer program

In addition to creating the background grid, the `redraw` method in `NameSurfer` also has to plot the actual data values.

You will need to link both buttons (Graph and Clear) to change what data values get displayed. Improtantly, both button clicks should call the `redraw` method, which deletes any existing `GObject`s from the canvas and then assembles everything back up again.

At first glance, this strategy might seem unnecessary. It would, of course, be possible to have the Graph action listener just add all of the `GLine`s and `GLabel`s necessary to draw the graph. The problem with that approach is that it would no longer be possible to reconstruct the entire graph, since `redraw` is the only method called when the screen size changes. By storing all of the *displayed* entries somehow within `NameSurfer`, the `NameSurfer` class can redraw everything when `redraw` is invoked from the `componentResized` method.

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

- To make the data easier to read, the lines on the graph are shown in different colors. The first data entry is plotted in **black**, the second in **red**, the third in **blue**, and the fourth in **magenta**. After that, the colors **cycle around again** through the same sequence.
- The **x-coordinates** of each line's segments start and end at the vertical grid lines for those decades.
- The **y-coordinates** should be calculated such that rank 1 is at the top of the graph area, and the max rank at the bottom of the graph area. All other ranks should be evenly-spaced in between.
- On each year for which there is data, also draw a **ranking label** next to the endpoint of the line that displays that entry's name and ranking for that year. The label's color should match the line's color. The x-coordinate of the ranking label is the same as the x-coordinate of that decade's vertical line, and the y-coordinate is the same as the y-coordinate of that corresponding year's plot line segment.
- One special case is a rank of 0, which should be drawn at the **bottom** of the graph area, like the maximum rank. Additionally, in its ranking label, you should list **an asterisk** instead of the numeric rank. You can see several examples of this in the data for "Samantha" in Figure 1, and in the provided demo JAR.

If you are having trouble getting the right coordinates for your lines 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.

Optional Extra Features

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:

- *Add features to the display.* The current display contains lines and labels, but could be extended to make it more readable. You could, for example, put a dot at each of the data points on the graph. You could also choose different symbols for each line so that the data would be easily distinguishable even in a black-and-white copy. For example, you could use little circles for the first entry, squares for the second, triangles for the third, diamonds for the fourth, and so on. You might also figure out what the top rank for a name is over the years and set the label for that data point in boldface.
- *Allow deletion as well as addition.* Because the screen quickly becomes cluttered as you graph lots of names, it would be convenient if there were some way to delete entries individually, as opposed to clearing the entire display and then adding back the ones you wanted. The obvious strategy would be to add a "Delete" button that eliminated the entry corresponding to the value in the "Name" box. If you do this, however, it comes with a challenge: you need to make sure that, if you added a bunch of entries to the graph and then deleted the early ones, the colors of the later entries *remain the same*.
- *Try to minimize the overprinting problem.* If the popularity of a name is improving slowly, the graph for that name will cross the label for that point, making it harder to read. You could reduce this problem by positioning the label more intelligently. If a name were increasing in popularity, you could display the label below the point; conversely, for names that are falling in popularity, you could place the label above the point. An even more challenging task is to try to reduce the problem of having labels for different names collide, as they do for `Sam` and `Samantha` in Figure 1.
- *Adjust the font size as the application size changes.* One of the wonderful features of this program is that it redraws itself to fill the available space if you change the size of the window. If you make it too small, however, the labels run together and become unreadable. You could eliminate this problem by choosing a font size that allows each label to fit in the space available.
- *Plot the data differently*. Right now, your program visualizes the data by showing its popularity over time. What other information about the names could you display? Consider plotting the rate of change over time, the correlation of various names, or other interesting trends that aren't apparent purely through their popularity.