Stanford University, CS 106A, Homework Assignment 7
NameSurfer (PAIR ASSIGNMENT)

Thanks to Nick Parlante for creating the assignment; revised by Marty Stepp, Patrick Young, Eric Roberts, Mehran Sahami, Keith Schwarz.

This assignment focuses on graphical user interfaces (GUI) and on collections such as `ArrayList` and `HashMap`. You will **turn in** a project including the following files:

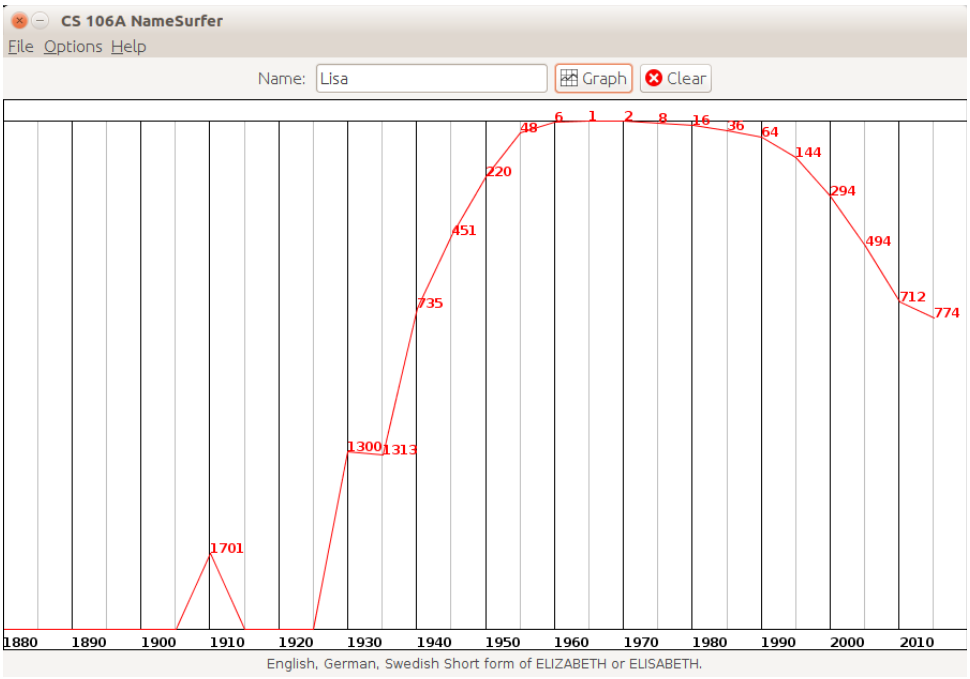
- **NameSurferGui.java**, **NameGraph.java, Person.java**, and **PeopleDatabase.java**

Note: This assignment is sometimes given by other instructors, but our version has significant changes. If you turn in another quarter's version of this assignment that does not conform to this quarter's spec, you will receive heavy deductions in your score.

Program Description:

Every year, the Social Security Administration releases **data** about the 1000 most popular boy and girl names for children born in the US. This data is provided at http://www.ssa.gov/oact/babynames/ . The data shows some interesting trends and implications. For example, the name "Samantha" became popular in the 1960s, possibly because the popular TV show "Bewitched" aired during that time. The name "Zelda" became much more popular in the 1980s due to the video game. The male first name "Adolf" dropped sharply in popularity in the mid-century, and so on. In this assignment you will write a graphical program that displays this data as colored line graphs.

In the screenshot below, the user typed Lisa (Female) into the Name box and then clicked "Graph". When the user enters a name, the program creates a red **plot line** showing that name's popularity for each year from 1880-2015. Clicking "Clear" removes the plot line and meaning from the graph so that the user can enter another name.



The most popular ranking for a name is #1. The graph shows rankings from 1-2000, with 2000 being the least popular ranking shown. Anything above 2000 appears at the bottom of the graph. Some names didn't have any rankings in certain years, in which case they are represented as a 0, which is treated the same as a ranking above 2000 and shown at the bottom of the graph. The program also shows the meaning of many names.

Demo: We provide a **runnable demo JAR** with a full solution to this assignment that you can run to test your program's expected behavior. We encourage you to run this JAR and match its behavior as closely as possible. We also provide several expected output images on the class web site to show the expected behavior and appearance.

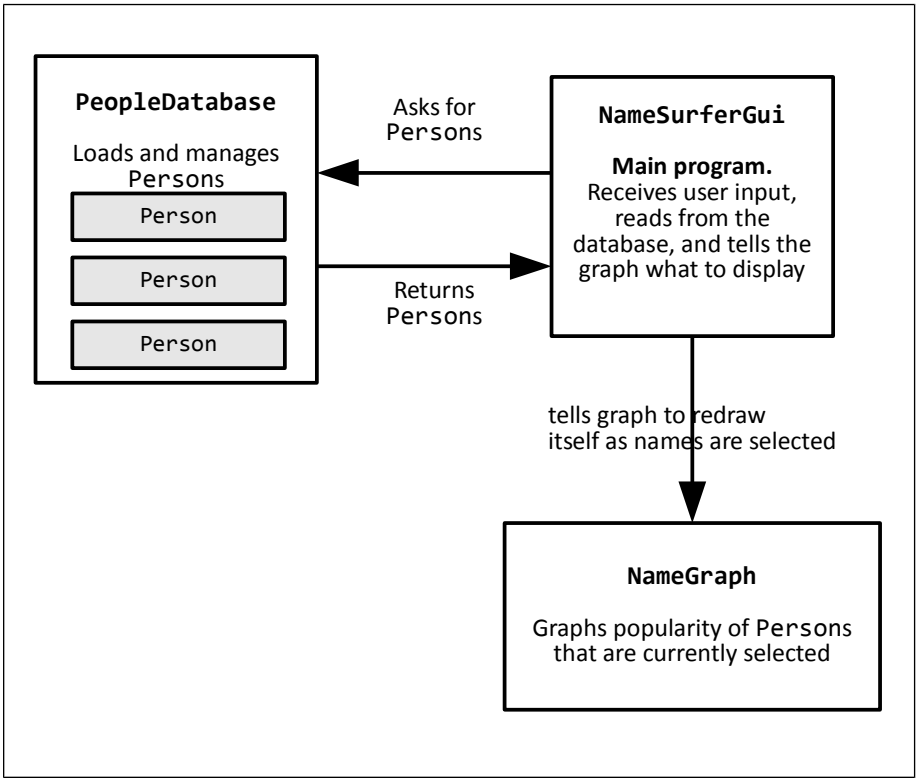
(The demo initially pops up a dialog box asking you which rank .txt input file to use. Your own program shouldn't do this. This is done in the demo to help you test what the output would be like if you changed the input file names.)

Classes to Implement:

To give you more experience working with multiple classes, the NameSurfer application is broken down into several **classes**, as follows. It's your job to write all of them except `NameSurferConstants`.

- **`NameSurferGui`**: This is the main program class that ties together the application. It has the responsibility for creating the other objects and for responding to the buttons at the bottom of the window, but only to the point of redirecting those events to the objects represented by the other classes.
- **`NameGraph`**: This is a graphical canvas (a subclass of `GCanvas`) that displays the graph of the various names by arranging the appropriate `GLine` and `GLabel` objects on the screen.
- **`Person`**: Each `Person` object contains the information for a particular name. Given a `Person` object, you can find out its name and what its popularity rank was in each decade.
- **`PeopleDatabase`**: 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 each 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 by having your classes implement the `NameSurferConstants` interface.

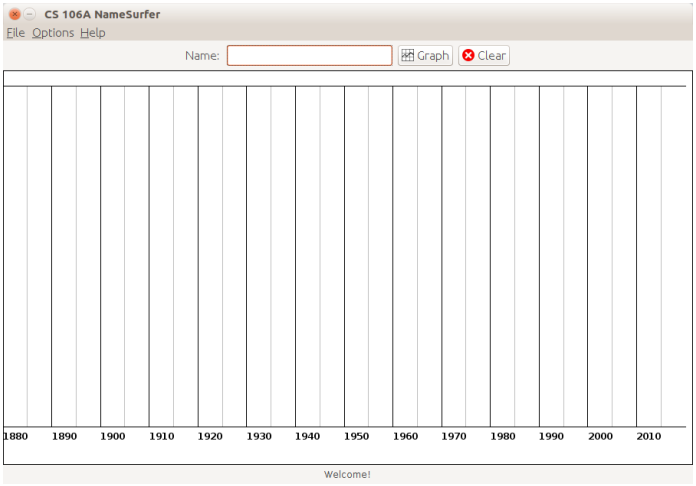
The following diagram roughly summarizes the relationship between the various classes:



Part of your style grade will come from properly **separating responsibilities** between these classes utilizing the concepts of **model** and **view** taught in lecture. For example, the `NameSurferGui` and `NameGraph` classes shouldn't directly contain the code that reads input files or breaks those lines apart into tokens; those lines should be in `PeopleDatabase` and/or `Person` as appropriate. Similarly, the `NameSurferGui`, `PeopleDatabase`, and `Person` classes should not contain any code to draw graphical shapes like `GLine`s or `GLabel`s on the screen; that code should be in `NameGraph`. `NameGraph` should not contain any code to set up GUI components such as buttons or text fields; that code should be in `NameSurferGui`. And so on.

NameSurferGui:

Your `NameSurferGui` class should set up the overall GUI for your program. The following are the major elements that should be present in the GUI when it initially appears, along with how each should behave.



- The window title should be "CS 106A NameSurfer".
- The overall window size should be 900x600 pixels. The window should not be **resizable**.
- The following components should appear in the **northern region** of the window, in this order left-to-right:
  - A should contain a **label** with the text "Name:".
  - A 16-character-wide **text field**, initially blank, for typing in names.
  - Two **buttons** labeled "Graph" and "Clear".
    - They must use icons from **graph.gif** and **clear.gif** from the **res/icons/** folder of your project.
  - Clicking the Graph button (or pressing Enter on the text field) should cause the program to graph the ranking data about the currently typed baby name.
    - The program is **case-insensitive**; "lisa", "LISA", or "LiSa" should all correctly find "Lisa". (Your `PeopleDatabase` should enforce this, not the GUI.)
    - If there is no ranking data about that baby name, the program should not add a new plot line. Instead, it should display text in the southern text label that says, "*Name* was not found." Any existing graph data lines should be cleared from the screen.
    - If there is ranking data about the name but no meaning data, the graph should still draw the lines, and "unknown" should appear in the Meaning text area when the user presses the Graph button.
- The **central region** of the window is occupied by a `NameGraph`, which will be described later.
- The southern region of the window should contain a text label (`JLabel`) for displaying name meanings. The label should use a 12-point "SansSerif" font. Initially this label should display the text, "Welcome!" If the user types a person's name that is found in the data set, that name's meaning appears here. Some name meanings are very lengthy strings that are too wide to fit in the window; do not worry about this and just let them run off the edge of the screen.

Loading large files: Loading data from large data files such as **ranks.txt** can take a long time and can cause a delay when your program is starting. Your window should appear immediately when the program runs rather than waiting for the file to be done loading. To do so, put the code to initiate the file-reading in the **`run` method** (or a helper that is called by `run`). This will cause the GUI to appear before the file has loaded. But since we don't want the user to try to search for name data before the data has been read, you should initially set your Name text field to be disabled and set its text to be "Loading data ...". Once the data is done loading, enable the field and clear out its text so that it is able to be interacted with by the user. (Code to create and add GUI components should be in `init`, not `run`.)

Your `NameSurferGui` should not directly contain the code that reads data from text input files. Such code should be in your `PeopleDatabase`. But `NameSurferGui` can call methods of the database to initiate such operations.

Input Data Files:

The input data about names' popularity rankings and meanings comes from a pair of input files provided on our web site. You will write code to read those files using `Scanner`s, so we will discuss the file formats here. You may assume that both files are readable and that all of the data in each file is in the proper format described below.

1) **ranks.txt**: This file contains information about how popular each baby name was in every 5th year. Each line of the file contains a single-word first name, followed by a series of numbers representing that name's rank every 5 years. The first ranking is for the year 1880, the second is 1885, the third is 1890, ..., 2005, 2010, and the last is 2015, the last year for which there is available data. Here are a few example lines from that file, abbreviated with .... *(Your program should not assume any ordering to the lines nor assume that every name has a particular casing.)*

```
Ethel 38 15 7 8 8 12 13 23 30 43 65 82 106 134 176 ... 0 0 0 0 0 0 0
Lisa 0 0 0 0 0 0 1701 0 0 0 1300 1313 735 451 220 48 6 1 2 8 16 ... 712 774
Morgan 288 337 369 372 405 389 ... 476 290 298 278 236 332 334 517 663
David 18 22 26 32 32 32 30 31 25 ... 11 7 6 5 2 1 3 3 5 4 6 6 16 16 13 15 18
```

When it comes to rankings, rank #1 was the most popular that year, #2 was second-most popular, and so on. Rank 0 means the name was not popular enough to appear in the data set for that year. For example, the name "Lisa" unranked (0) in 1880 - 1905 before appearing at #1701 in 1910. "Lisa" dropped to unranked status until 1930 when it resurfaced at #1300. The numbers continue in this fashion until 2010 - 2015, where "Lisa" is sitting in the 700s. The name "Ethel" for women was very popular in 1880 (#38) but has dropped steadily and is now unranked (0).

We provide input files such as **ranks-tiny.txt**, **ranks-small.txt**, etc. in the **res/** folder of your project for testing.

2) **meanings.txt**: This file contains data about the origin and meaning of names. The data in this file is organized into pairs of lines with one pair about each name. The first line of each pair is the name itself in uppercase. The second line is the meaning of the name. Here are some example lines from this file:

```
ETHEL
English Old short form of beginning with the Old English element el meaning "noble".
LISA
English, German, Swedish Short form of ELIZABETH or ELISABETH.
MORGAN
Welsh, English From the Old Welsh masculine name Morcant, possibly derived from ...
```

The contents of **ranks.txt** and **meanings.txt** come from different source web sites and are not in sync with each other. In other words, not every name from **ranks.txt** has a meaning provided in **meanings.txt**; and vice versa, not every name from **meanings.txt** is necessarily present in **ranks.txt**. (For example, "Alfonzo" and "Rakeem" have ranks but no meanings in the data, and "Stefano" and "Vivek" have meanings but no ranks.) Your code should not crash or behave improperly when given a name that is found in one data file but not the other.

Some names' meanings contain unusual characters, misspellings, or strange text, especially for international names. You don't have to worry about this. Whatever text is on the meaning line of text is the text you should display in your text box in the GUI on the screen. *(Issues like this are common when grabbing messy data from various online sources.)*

Storing the Data in HashMaps:

In your `PeopleDatabase` class described later in this document, you will write the code to read each of the above data files. You must store the data from each file into a **`HashMap`** as a private field so that any person's data can be looked up efficiently using their name as a key. For example, the key of "Morgan" could map to the value of the `Person` object related to Morgan. *(Implementations that do not use a `HashMap` internal representation or use their `HashMap` in inappropriate ways will receive a substantial deduction.)*

You should read each file only once during the lifetime of your program, when the program first loads up (in the `run` method, or a helper called by `run`). Perform all file reading, tokenizing, `Scanner` work, etc. in code that is called from your overall GUI's `run` method so that the program simply needs to look up the data later when the user types names.

Person class:

In your `Person` class, you're defining a new type of objects that will help you manage the baby name data. Each `Person` object represents the data about a single baby first name, including the name itself, the meaning of that name, and the rankings for that name for each year in the data set.

`public Person(String dataLine)`

In this constructor you should initialize the state of a new person based on the given line of data. You should assume that the line of data is in the format from **ranks.txt** described previously, with a first name and a series of integer ranks all separated by spaces, such as:

```
Morgan 288 343 365 404 302 337 303 410 408 363 369 402 ... 517 536 582 663
```

In your constructor you should break apart this data and store it appropriately in your new `Person` object. You should not store the entire line itself as a field, but rather, break it apart and store each piece separately. You must store the Person's ranking data in an **`ArrayList`**. Do not use an array; use an `ArrayList`.

`public String getName()`

In this accessor method you should return the person's name as was read from the input data.
For example, given the above line, `getName` would return `"Morgan"`.

`public String getMeaning()`
`public void setMeaning(String meaning)`

These two methods allow you to access and modify the meaning of this person's name as read from the meanings data file. If no meaning is known for this person, return the string `"unknown"` . Most of the private data fields of the Person don't have a set method such as `setMeaning`, but you'll want one here because you read the person's ranking data before you read its meaning data.

`public int getRank(int year)`

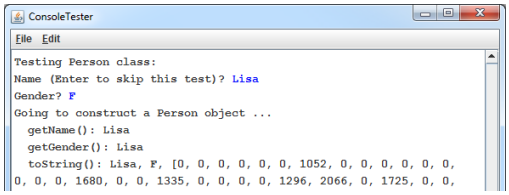
In this accessor method you should return the person's name's ranking for the given year. For example, given the above line, `getRank(1880)` would return `288` and `getRank(2015)` would return `663`. If the year passed is outside the valid range of the data, you should return `-1`.

`public String toString()`

In this accessor method you should return a string representation of your person's data. The format must list the person's name followed by a space and a list of their rankings. For example, the `Person` object for Morgan would return the following string *(abbreviated)*:

```
"Morgan [288, 343, 365, ..., 582, 663]"
```

Console tester: To help you develop this program incrementally, we have provided a class named `TestPerson` that you can run to test the functionality of the `Person` class. We strongly suggest that you develop class `Person` in isolation and then thoroughly test it in the `TestPerson` before moving on to other parts of the assignment.



PeopleDatabase class:

In your database class, you will write code to read files of data about baby names, rankings and meanings, as well as code to keep track of all the data about the persons in the data set. This is the class where you'll declare your collections to keep track of the massive amount of information from the input files.

You should represent each person's data as a `Person` object inside the database.

You will implement the database **using a `HashMap`** internally so that each `Person` object can be looked up efficiently using their name as a key. *(`PeopleDatabase` implementations that do not use a `HashMap` internal representation or use their `HashMap` in inappropriate ways will receive a substantial deduction.)*

`public PeopleDatabase()`

In this constructor you should initialize the state of an empty database, including any data fields you declared.

`public void loadData(String ranksFile, String meaningsFile)`

When this method is called, your database should reading the files of data about people in the formats described previously. Our intention is that your `NameSurferGui` should create your `PeopleDatabase`, then call the database's `loadData` method, passing appropriate file names to it.

You should read the entire contents of these files and use them to create many `Person` objects that are to be stored into your internal `HashMap`. You may assume that both files are readable and that all of the data in the files will be in the proper format described previously.

`public Person findPerson(String name)`

After the data has been read, a call to `findPerson` should retrieve `Person` objects from your database's hash map and return it to the caller. This method should be **case-insensitive**; the name can be passed with any capitalization and it should still be found properly. If no person with that name exists in the data set, return `null`.

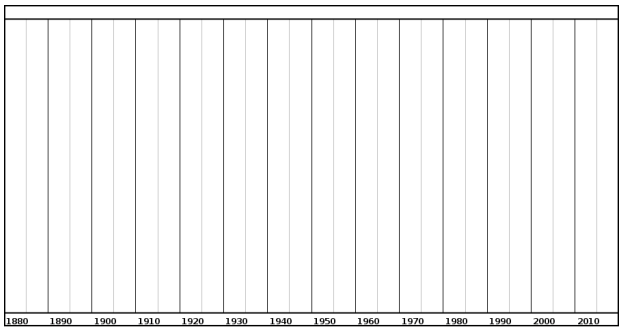
If you find that you would like to add **extra methods** to `Person`, `PeopleDatabase`, or any other class in this program, you may do so, as long as your extra methods are **private**.

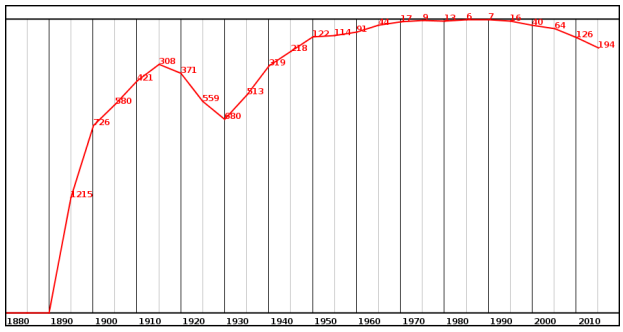
As with the `Person` class, you can test `PeopleDatabase` in isolation using our provided `TestDatabase` console program. We strongly recommend that you do this before moving on to the other parts of the assignment that depend on your database working properly.

NameGraph class:

The name graph is your central graphical canvas where you will draw the line graphs of baby name popularity. This class extends `GCanvas`, which means that a `NameGraph` can be added to the central display area of your overall `NameSurferGui` program (using its `add` method). You can call any of the 2-D graphics methods you've learned, like `add` or `remove` (passing a `GLine`, `GLabel`, etc. as a parameter), from within your `NameGraph`.

The screenshots below show the initial appearance of the name graph, and its appearance with a plot line added:





Here are the details about the appearance of the graph that you must match exactly:

- The overall name graph has a 1px-thick black solid **border** around it (use `BorderFactory`.)
- The graph's overall **background** is white.
- On the top and bottom of the graph, leave a 20px-tall **margin** area. The rest we'll call the "central area."
- Draw a set of **vertical grid lines** to indicate years. For each year in which there is rank data (1880, 1885, 1890, ..., 2010, 2015), draw a vertical line that spans the entire central area, that is, the canvas other than the top/bottom 20px margin areas. The lines are evenly spaced, dividing the horizontal space of the graph evenly for each 5-year period, including an extra bit of space after the last data point so that it and its associated rank text can be seen. This means that the horizontal positions of the lines are based on the number of years of data and the graph's width. The default range of years is from 1880-2015, which corresponds to 28 five-year ranges to show on the graph, including the extra one at the end: *ceiling(2015-1880)/5 + 1*. For example if the graph is 900px wide by 440px tall, each 5-year gap would occupy 900 / 28 ~= 32.14px of horizontal space, meaning that the vertical grid line for 1880 would run from x,y of (0, 20) to (0, 420); the line for 1885 would run from (32.14, 20) to (32.14, 420); the line for 1881 would be at x=64.29, and so on up to 2015 at x=867.86. (Use `double`s in case numbers don't divide evenly.)
  - For years that mark the start of a decade (1880, 1890, ..., 2010), the line should be colored **black** (`Color.BLACK`). For years that are not the start of a decade (1885, 1895, ..., 2015), the vertical line is **light gray** (`Color.LIGHT_GRAY`) . In the default data set and constants' values, every other ranking falls on a decade starter like 1890, 1900, etc., but you should not assume this.
  - Every alternating year should also contain a **decade label** (`GLabel`) that displays that year as a string, such as "1880" for the year 1880, with its horizontal position the same as the vertical line for that year, and with its vertical position offset by -3px from the bottom of the window. Show a label for the first year for which there is data (1880 by default) and for every 2nd data point after that (1890, 1900, 1910, etc). So in our example with 28 chunks of data and a window size of 900px wide by 440px tall, the decade label for 1880 would be at (x=0, y=437); the decade label for 1890 would be at (x=64.29, y=437); and so on. The decade labels should be drawn in a 12pt bold SansSerif font. You do not need to worry about the case where there is not enough horizontal space to fit the labels.
- Your graph should draw a red (`Color.RED`) **plot line** for the person that is typed and chosen when the user presses the Graph button. The `NameSurfer` should pass the data about that person to the `NameGraph`, including the rank data to be drawn for each year. Your plot line should draw a line segment (`GLine`) between each neighboring pair of years. For example, you'd draw a line from 1880 to 1885, and a line from 1885 to 1890, and so on, up through a line from 2010 to 2015. *(The Stanford libraries have a class called `GLineGroup` that might prove useful for this part of the program. See web site.)* If the user types a new name and presses the Graph button again, clear out the plot line and rank information of the previous name and replace it by drawing the plot of data for the new name.

- The **x-coordinates** of each line segment's start and end are the same as the vertical grid line for the relevant years. For example, in our previous example of a window of size 900x440, the line from 1880-1885 goes from x=0 to x=32.14, and the line from 1885-1890 goes from x=32.14 to x=64.29, and so on, up through the line from 2010-2015 which goes from x=867.86 to x=900.
- The **y-coordinates** are calculated based on the height of the central area of the graph relative to the maximum rank to display (`MAX_RANK_TO_DISPLAY`), which is 2000 by default. Top ranks like 1, 2, 3 are drawn closer to the top of the window (y=20), and worse ranks like 1998, 1999, 2000 are drawn closer to the bottom (y=410, in our example window of height 440), with a linear scale in between. For example, rank 1 would be drawn at y=20. Rank 500 would be drawn at y=120 (one fourth of the way down); rank 1000 would be drawn at y=220 (halfway down); rank 1500 would be drawn at y=320 (3/4 of the way down), up to rank 2000 and above, which would be drawn at y=420 (the bottom).
- There are a few **special-case rankings**. A rank of `MAX_RANK_TO_DISPLAY` or above (2000 by default) is drawn at the bottom of the central area, at the same y-coordinate as a rank of `MAX_RANK_TO_DISPLAY`. Also, a rank of 0 is treated the same way; the rank of 0 is considered to mean "no data available," meaning that it is not a popular name that year, so it also should be drawn at the bottom of the central area.

- On each year for which there is data, also draw a **ranking label** next to the line endpoint that displays the name's ranking for that year. The label's color should match the line's color. The ranking labels should be drawn in a 12pt bold SansSerif font. The x-coordinate of the ranking label is the same as the x-coordinate of the corresponding year's grid line, and the y-coordinate of the ranking label is the same as the y-coordinate of the corresponding year's plot line segment.

It is up to you to decide what constructor(s), methods, and fields your name graph should have. Part of your style grade will come from choosing a good set of methods to provide the above functionality and to decompose it well into manageable sized chunks of code. Some general principles you should adhere to in your design are:

- The `NameGraph` should not store the `HashMap`s of ranks and meanings; those are in the `PeopleDatabase` .
- The `NameSurferGui` will need some way to tell the `NameGraph` what to draw then the user presses the Graph button. You'll need to provide a method where the `NameSurferGui` can pass to the graph the relevant data about the person to be drawn.
- When the user tries to graph a name that is not found, the graph should clear its red plot lines and rank labels. Provide some kind of method so that the `NameSurferGui` can tell the `NameGraph` to do this.
- There is a lot of drawing code to be found in the `NameGraph` class. If you put literally all of it into one gigantic method, you aren't going to get full style credit. You have been warned. Decompose your code.

Your name graph's code should use the appropriate constants defined in **NameSurferConstants.java** *(described on the next page)* and should adjust accordingly if they are changed. For example, if the number of years is changed, the width between the grid lines changes. If the max rank to display is changed, the y-coordinates of the grid lines change. And so on. Your program should not make any assumptions about the values of these constants. For example, the default range of years is 1880-2015, but you should not assume this, nor should you assume that the data starts/ends on a year that is a multiple of 10, etc. But you may assume that the values of the constants are reasonable/valid, such as that the end year will be greater than the start year and so on.

NOTE ABOUT VERY COMMON "WIDTH AND HEIGHT ARE 0" BUG: A lot of students encounter a bug where the width and height of their `NameGraph` are 0. This will happen if your code tries to ask for the size of your graph in its constructor, and/or if you ask for the size of the graph in code that was called by the `NameSurferGui`'s `init` method. Instead, ask for the graph's size in a method invoked by the `NameSurferGui`'s `run` method, after the graph has already been added and appeared on the screen. Then the width and height will be the correct value. Do not work around this problem by "hard-coding" an expected size, nor by adding constants to the GUI/graph, nor by passing a size from the GUI to the `NameGraph`.

NameSurferConstants interface *(provided)*:

Our starter ZIP contains a file **NameSurferConstants.java** full of constant values that are useful to your program. Your code should use these constants when appropriate and should respond accordingly if their values are changed.

```
public static final String RANKS_FILENAME
public static final String MEANINGS_FILENAME
```

These constants represent the input file names your program will use to read its data about baby names. By default they are set to `"res/ranks.txt"` and `"res/meanings.txt"`, but it can be useful to set them to other file names for debugging. For example, we have files named **res/ranks-tiny.txt** and **res/ranks-small.txt** with fewer names; try testing with those files before using the big one.

```
public static final int MIN_YEAR
public static final int MAX_YEAR
public static final int YEAR_GAP
public static final int INTERVALS_OF_DATA
```

These constants represent the minimum and maximum years of the data in the **ranks.txt** input file. By default the minimum year is 1880 and the maximum year is 2015. The constant `YEAR_GAP` is the gap between each integer in the ranking data, default 5. The constant `INTERVALS_OF_DATA` is the number of 5-year gaps between these two values, which is: *ceiling*((2015-1880)/5)+1 = 28. This can be useful as a boundary or array size.

```
public static final int MAX_RANK_TO_DISPLAY
```

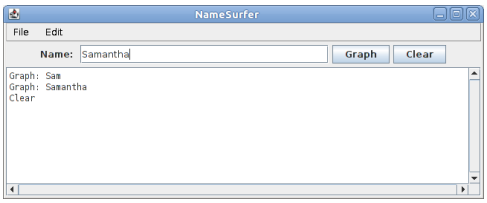
The `MAX_RANK_TO_DISPLAY` constant represents the maximum rank your program should display on the screen. Its default value is 2000. A rank higher than this is considered to be "zero" or unranked for that year.

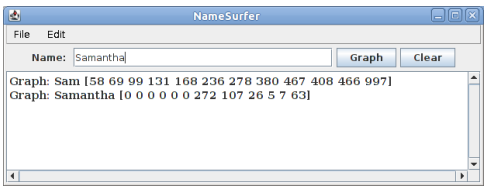
Your code must work properly if constants are changed in sensible ways. For example, if we changed the meanings file name to a different file, we would expect your code to read from that file. Or if we changed the `MAX_YEAR` constant to 2020 and also added another integer of data to the end of each line in the ranks input file, we would expect your graph to draw all of the data.

Development Strategy and Debugging Tips:

This is a challenging program, so we suggest that you develop it in stages. Here are some suggested milestones:

- **Implement Person:** The main challenge here is taking a string of data as a parameter and breaking it apart into its individual pieces. Use our provided `TestPerson` to make sure it works.
- **Implement PeopleDatabase:** Get the file reading and name lookup working next. Use our provided `TestDatabase` to verify its functionality.
- **NameSurferGui that reads/prints name only:** Write an initial version of your NameSurfer GUI that just sets up the various controls and, when the user types a name and clicks "Graph", displays that name or `Person` on the screen. You could display it by temporarily modifying your `NameSurfer` class heading to say `extends ConsoleProgram` , so that you can `println` and see the results.





- **`NameGraph` that just draws the grid:** Write an initial version of `NameGraph` that draws the top/bottom margins and the grid lines for each year, without trying to draw any of the graph plot lines. Make sure it works even when the constants from **NameSurferConstants.java** are changed. Only once the grid lines and decade labels are perfect should you go on to the red ranking plot lines.

Debugging Tips:

Tip 1: If you are having trouble getting the right coordinates for your lines or labels in your name graph, try printing the x/y coordinates to verify them on the console. You can't generally use `println` statements in classes other than `NameSurferGui`, such as `NameGraph`, because they are not your main program class. But if you write `System.out.println` instead of just `println`, you will see the printed messages in the Eclipse console.

Tip 2: Here is a quick reference for how to convert between some common Java types:

- `int` to `String`: `String s = "" + myInt;`
- `String` to `int`: `int n = Integer.parseDouble(myString);`
- `double` to `String`: `String s = "" + myDouble;`
- `String` to `double`: `double d = Double.parseDouble(myString);`

Extra Features: *(Extra features are optional and will earn you a small amount of extra credit if you complete them.)*

If you are going to do extra features, **submit two versions** of the assignment: the basic Java files that meet all the normal assignment requirements, and an "extra" set of files such as **NameSurferExtra.java** or **NameGraphExtra.java** containing your extended version. (If your extra features don't break or change the base functionality in any way, you can just put the extra features in the existing java files.) At the top of your .java file in its comment header, you must **comment** what extra features you completed. Here are a few ideas for extra features:

- *Allow multiple names:* The current default version graphs only one name's data at a time. But you could extend it to show multiple names at the same time in different colors.
- *Add features to the display:* The current display contains only lines and labels, but could easily be extended to make it more readable. You could, for example, put a dot at each of the data points on the graph. Even better, however, would be to 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.
- *Plot the data differently:* Right now, your program visualizes the data by just showing its popularity over time. What other information about the names could you display? Consider plotting the rate of change over time, or (if you're statistically-minded) the correlation of various names. Can you find any interesting trends in baby names that aren't apparent purely through their popularity?
- *Resizable graph:* The default version of the program is supposed to set its window to be non-resizable. Make it so that the program gracefully resizes and the name graph resizes to fit its new window size. You may need to learn about adding a component listener and the `componentResized` method to do this.
- *Change out the data set:* The type of data you're exploring in this assignment is called time-series data because it encodes values over time. There are many other time-series data sets out there, though: you could look at stock prices, population totals, literacy rates, etc. See if you can find any other data sets that you could add.
- *Other:* Use your imagination. What other features could you imagine in a program like this?

Grading:

Review the "Grading" section of prior specs for reminders of our general grading expectations.

Style: As always, follow the style guidelines taught in class and listed in the course **Style Guide**. Show us a good procedural **decomposition** of code into methods to indicate structure and avoid redundancy. Minimize the use of **data fields** and prefer local variables as much as possible. Encapsulate fields by making them **private**. If there are important fixed values used in your code, declare them as final **constants**. Use descriptive **names**. **Format** your code cleanly. Avoid **redundancy**. Use descriptive **comments**, including the top of each .java file, atop each method, inline on complex sections of code, next to each field, and a **citation of all sources**. If you complete any **extra features**, list them in your comments to make sure the grader knows what you completed.

Honor Code: Follow the **Honor Code** when working on this assignment. Submit your own (pair's) work; do not look at others' solutions. Do not give out your solution. 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.

Copyright © Stanford University and Marty Stepp, licensed under Creative Commons Attribution 2.5 License. All rights reserved.