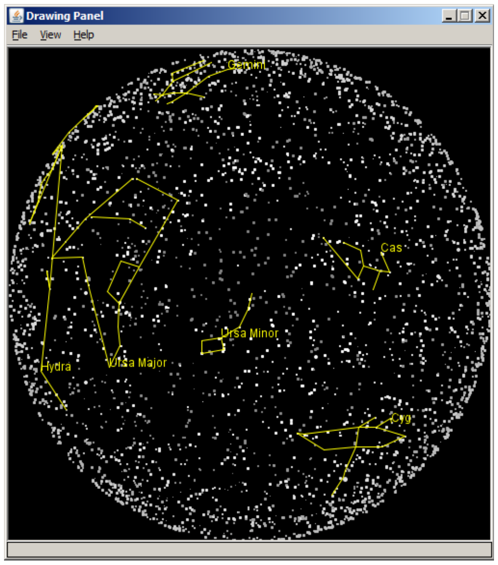
CSE 373, Winter 2013
Homework Assignment #2: Star Chart (25 points)
Due Friday, January 25, 2013, 11:30 PM

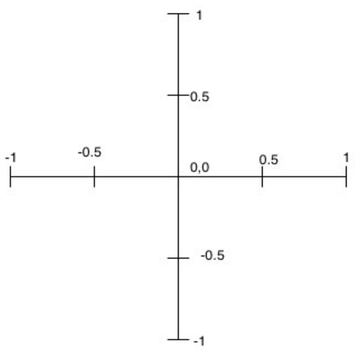
Thanks to Karen Reid for the inspiration and data for this assignment!

This program continues our focus on the Java collections framework, as well as the Google Guava collection library. It also involves considering the Big-Oh complexity of code to process various collections. Turn in two files named `StarChart.java` and `Star.java` from the Homework section of the course web site.

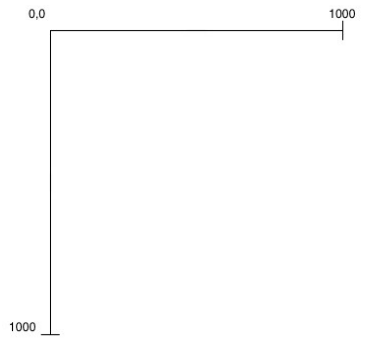
You will also need to download the support files `Main.java` and `DrawingPanel.java` and various input text files from the Homework section of the course web site; place it in your Java project and add to it. If you are using Eclipse, place .java files in your project's `src/` folder and .txt files in the project root folder (the parent of `src/`). We also provide a 'stub' version of `StarChart.java` and a partially completed `Star.java` on the web site as a starting point if you like.

A set of provided data files to use as input for the algorithm is also posted on the Homework section of the web site. We will not provide testing code for you other than the `Main.java` program, but we do not guarantee that the main program is an exhaustive test. You should perform additional testing of your own before you submit your program.





Star Catalog Coordinate System



Picture Coordinate System
(with an example width and height of 1000 pixels)

Program Description:

Astronomers collect lots of data about stars and there are many catalogs that identify the locations of stars. In this assignment, you will use real data from a star catalog to draw a "**star chart**" figure that plots the locations of the stars. Though the data is 3-dimensional, our figure is 2-D and uses the third dimension ("*z*") to indicate star color brightness. Each star in the input file has *x*/*y*/*z*-coordinates between -1.0 and 1.0, with 0.0 being the center; we will map these to screen coordinates when we draw stars in our star chart. This conversion is done for you in provided code.

A **constellation** is an internationally recognized pattern formed by connecting lines between prominent stars within apparent proximity to one another on Earth's night sky. Your program will also read input data for constellations and draw lines between stars to depict those constellations on top of your star chart. The provided `Main` program has a text menu interface that allows you to add constellations to your star chart, show/hide star names, and so on.

Since a real data set often has some incorrect or missing data, a cleaned up catalog has been prepared for your use in this assignment. The provided input file `stars.txt` contains one line for each star that is represented in the catalog. The provided `Main` file handles all file processing and creates `Star` objects that are added to a `StarChart` object. The `StarChart` object can then draw itself onscreen using a `Graphics` object and perform other manipulations.

Just for fun, we are also going to simulate stars going "supernova" and exploding, possibly destroying other nearby stars in the process. This is not very astronomically accurate, but it is fun to destroy a few stars and see what happens. When stars explode, they turn red on the screen in our star chart.

Star Implementation Details:

The fundamental object of data in this program is a star, so you will write a class named `Star` where each object represents one star in the sky. A partial skeleton of this class is provided to you. The following are **already provided**:

```
public Star(double x, double y, double z, double magnitude)
```

Initializes a new star object at the given 3D position and having the given magnitude (size). The x/y/z positions passed are from -1.0 (bottom or left) through 1.0 (top or right), with 0.0 being the center.

```
public double getX()
public double getY()
public double getZ()
public double getMagnitude()
```

Returns the *x*/*y*/*z* position and magnitude of the star as was passed to its constructor originally.

```
public int pixelX(int width)
public int pixelY(int height)
```

Returns integer *x*/*y* coordinates at which this star should be drawn on a 2D window of the given size. They translate the star's own *x*/*y* coordinates, which are in the range [-1.0 .. 1.0] as described in the constructor, into screen pixel coordinates which begin at x=0, y=0 for the top-left corner of the screen and increase +x rightward and +y downward. (Note that the screen's y-axis is flipped relative to the star y-coordinates.)

```
public int pixelSize()
```

Returns an integer width/height with which this star should be drawn on a 2D window. The size is half of the star's magnitude, rounded to the nearest integer, with a minimum pixel size of 1.

```
public Color pixelColor()
```

Returns a color with which this star should be drawn on a 2D window.

You must add the following methods to the class with exactly the headers shown, so they can be called by the other classes. In addition, your class should implement the `Comparable` interface so that stars can be compared to other stars for ordering. You may define additional methods if you like, but **they must be `private`**.

```
public int compareTo(Star other)
```

In this method you will return an integer indicating the relative ordering of this star with respect to the given other star. Stars are compared by z-position, breaking ties by y-position, then by x-position, and lastly by magnitude. A larger position or magnitude is considered "later" in the ordering than a smaller one and therefore should cause your method to return a positive integer. If all values are the same, your method should return 0.

```
public double distance(Star other)
```

In this method you should compute and return the 3-dimensional direct distance between this star and the given other star. The direct distance between two three-dimensional points (x_1, y_1, z_1) and (x_2, y_2, z_2) is found by the formula:

$$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2}$$

```
public boolean equals(Object o)
```

In this method you should return true if `o` refers to a star with equivalent state to this star. Equivalent state means having exactly the same x, y, z coordinates and magnitude.

```
public String toString()
```

In this method you should return a string representing the state of this star. The string contains the *x*/*y*/*z* coordinates of the star in parentheses and separated by commas, followed by a colon and the star's magnitude. For example, a star located at x=-0.54, y=0.216, z=-0.87, with magnitude 6.3, would return `"(-0.54,0.216,-0.87):6.3"`. Note that there are no spaces in the string. Your string's format must match exactly.

StarChart Implementation Details:

The second class you will write named `StarChart` represents a collection of stars. You must write the following methods with exactly the headers shown, so they can be called by the provided `Main` class. You **may define additional methods** if you like, but they must be `private`. Do not add any data fields, nor methods that mutate a star's state.

`public StarChart(int width, int height)`

In this method you should initialize a new star chart object of the given screen pixel size. Initially there are no stars.

`public void addStar(Star star, String name)`

In this method you will add a single star to your star chart. This method is called by the `Main` class, once for each star in the input file. Some stars do not have a name, in which case a `null` string will be passed. You may assume that the given star and the given name are unique, and no duplicates of either are ever passed again to this method.

`public void addConstellation(String[] starNames, String constellationName)`

In this method you will add a constellation to your star chart. The given array is arranged such that each pair of elements represents a line segment between the two stars with those two names. For example, the array `{"PHECDA", "MERAK", "MERAK", "DUBHE", "MEGREZ", "DUBHE"}` represents three line segments: one between Phecda and Merak; one between Merak and Dubhe; and one between Megrez and Dubhe. You may assume that the array's size is even and that all names in the array correspond to stars that were added previously via `addStar`.

`public String getName(Star star)`

In this method you should return the name associated with the given star in the star chart. If the given star is not part of the chart, or if it has no name, you should return `null`.

`public Star getStar(String name)`

In this method you should return the star associated with the given name in the star chart. If the given name is not associated with any star in the chart, you should return `null`.

`public Star getStar(double x, double y)`

In this method you should return the star located at the given *x*/*y* position in the star chart. These are star coordinates, not screen coordinates. Though stars also have *z*-coordinates, those are ignored here; in the star chart we will assume that no two stars have identical *x* and *y*. If the given position is not associated with any star in the chart, you should return `null`.

`public int supernova(Star star)`

In this method you should record that the given star has exploded in a supernova. The explosion also destroys any star whose 3D distance is ≤ **0.25** from the given star. These "dead" stars are not removed from your star chart nor purged from all of its internal data structures, but they are drawn differently on the screen (see below). Your method should return the number of stars destroyed, including the given star itself if applicable. For example, if the star explodes and destroys 3 other stars, return 4. If the given star is already "dead", calling this method on it has no effect and returns 0.

`public void draw(Graphics g, boolean showStarNames)`

In this method you should use the given graphical pen to draw all of the stars and constellations in this star chart. Use the various provided "pixel" methods of each star to draw it. Draw each star as a solid rectangle (`fillRect`) at the star's pixel size and pixel x/y coordinates, using its pixel color. Any "dead" stars that have been destroyed in a supernova should be drawn in red (`Color.RED`) rather than their normal pixel color.

Draw each constellation by a yellow (`Color.YELLOW`) line segment (`g.drawLine`) between each pair of stars. Also display the constellation's name (`drawString`) at the pixel *x*/*y* position of the constellation's *last* star. Your code should draw the stars first and the constellations second so that the constellation lines appear "on top" of the stars.

If the parameter `showStarNames` is `true`, write the name of each star that is part of a visible constellation, at the same pixel *x*/*y* coordinates of that star. Write the star's name in the same color that was used to draw that star.

For all methods in both classes, **assume that the parameters passed are valid**. No parameter will be `null`; the width/height will be non-negative; duplicate/missing stars, star names, and constellations will not be passed; and so on.

Development Strategy and Hints:

- Write the `Star` class first, because `StarChart` does not work without it. We will provide resources so that you can test `Star` in isolation even if your `StarChart` is not yet completed.
- Make it so that you can add stars to your star chart. Don't worry about constellations or supernovas yet. Write the `draw` method so that the stars show up as little squares on the drawing panel.
- Write the various `get` methods on the star chart.
- Implement supernovas.

Graphics Class:

The `Graphics g` object passed to your `draw` method has the following useful methods:

```
public void drawLine(int x1, int y1, int x2, int y2)
public void drawString(String s, int x, int y)
public void fillRect(int x, int y, int width, int height)
public void setColor(Color color)
```

A `Color` is specified either as a constant such as `g.setColor(Color.YELLOW)`, or by a star's pixel color such as `g.setColor(theStar.pixelColor())` .

Style Guidelines and Grading:

We will grade your program's behavior and output to give you an **External Correctness** score. You can use the DrawingPanel's image comparison feature (File, Compare to Web File...) to check your output. Different operating systems draw shapes in slightly different ways, so it is normal to have some pixels different between your output and the expected output. You do not need to achieve 0 pixels difference to get full credit for your output. If there is no visible difference to the naked eye, your output is considered correct. (If your figure looks the same but has "thicker" letters or shapes or lines, you might be re-drawing the same shapes multiple times, which causes them to thicken on the screen.)

We will also grade your code quality (**Internal Correctness**). There are many general standards that you should follow, such as naming, indentation, comments, avoiding redundancy, etc. For a list of these standards, please see the **Style Guide** document posted on the class web site. Follow all of its guidelines as appropriate on this and future assignments.

A major component of internal correctness on this assignment comes from choosing appropriate **collections** to store your data and using efficient algorithms to process those collections. In this assignment you should use collections from the **Guava library** in addition to standard Java collections. See the course web site for instructions on setting up Guava.

You will want to use several collections as **private fields** inside your `StarChart` to solve the problem. Think about what kinds of collections would allow you to easily answer questions such as the ones in the list above. Here are some general guidelines you should follow in choosing your collections if you want full credit:

- **Favor execution speed** over memory usage. If you can create an extra collection that allows a faster look-up to implement a required method, it is worth the extra memory required.
- **Favor Guava** collections over compound collections where appropriate; for example, choose a `Multimap` over a map of lists/sets; choose a `BiMap` over a pair of inverse maps; choose a `Table` over a map of maps; etc.
- Favor collections over arrays throughout your code. `addConstellation` passes an array, but don't use others.
- In the absence of other constraints, favor the collection that implements the needed behavior most **efficiently**.
- When pairs of pieces of data are related to each other, use a map. When triples are related, use a table.
- When items of a map or set should be processed in a sorted order, favor a tree-based collection implementation. When order is irrelevant, favor a hash-based implementation.
- You should not need to loop over a collection to search it for a value (or call `indexOf`/`contains` on a list). If asked to find the *A* associated with a given *B*, use a direct map/table between *B*s and *A*s instead of looping. Searching for the stars that are destroyed in a supernova is one exception; you may loop over all stars there.
- You should not need to explicitly sort any collection after it has been filled with data.
- You should not need to re-compute a complex value that you have previously computed.

Our own sample solution uses four total collections, three of which are Guava collections. Our code does *not* use any Guava collections involving the `Range` class such as `RangeSet` or `RangeMap`.