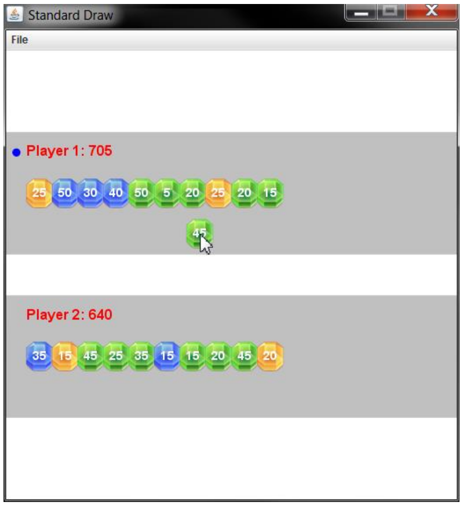
Gem Matching

In this assignment, you will be creating a Gem Matching game. Your job is to write two support classes used by the game. You will be implementing a linked list and using an enumerated type.



Gem Matching. The game is played by two players who take turns placing a gem. Each new gem is randomly one of three different colors and has a random point value. A player may place a gem in their own row of gems to increase their score. A player may also place a gem in their opponent's row to break up a gem *block*. A *block* is any group of consecutive gems of the same color.

A player's score is the sum of the point total of all gems in their row. Blocks in a player's row incur a score multiplier. The score multiplier is the number of gems in the block. For example, a block of three green gems with point values 10, 20, and 30 would earn 3 * (10 + 20 + 30) = 180 points. See here for a short demo of how the game works.

Due to multipliers, it can sometimes be advantageous to give a gem to your opponent since you can use it to break up a block. The game ends when either player hits 16 total gems; the player with the higher score wins.

Files. The lab folder contains the three image files for green, blue and orange gems. It also contains three Java classes. You have been given a completely finished version of `GemGame.java`. You will be implementing the two support classes `Gem.java` and `GemList.java`.

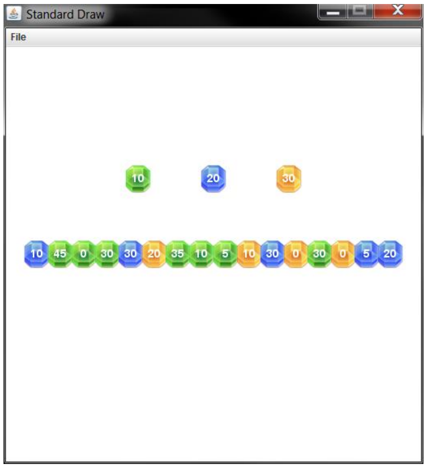
Gem. The `Gem` class represents a single gem in the game. It knows things like its color (green, blue, or orange) and its point value. You should use the provided ***enumerated type*** (more info in lab folder) to represent the color of a gem. A new gem is assigned one of the three colors at random. The point value of a gem is a random value in the set {0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50}.

Using the StdDraw class, Gems are drawn first using one of the image files **"gem_green.png"**, **"gem_blue.png"**, or **"gem_orange.png"**. The point value is then drawn in a white text over the image.

Here is the API you should implement for the `Gem` class:

```
class Gem
-------------------------------------------------------------------------------------
         Gem()                             // create a gem with random color and point value
         Gem(GemType type, int points)     // create a gem with the specified color and point value
  String toString()                        // return a string representation of the gem
 GemType getType()                         // get the type of the gem
     int getPoints()                       // get point value of the gem
    void draw(double x, double y)          // draw gem at (x, y) using StdDraw's methods (see docs/FAQ)
```

A number of tests inside a `main` method have been provided; feel free to add more as you see fit. You should see the console and StdDraw output shown below. **NOTE:** the row of 16 gems is using the default constructor (that creates a random gem), so you are unlikely to get the same thing!



Console output:

```
GREEN 10, GREEN, 10
BLUE 20, BLUE, 20
ORANGE 30, ORANGE, 30
```

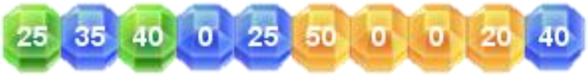
GemList. The `GemList` class represents all the gems held by a player, as well as their order. The class allows new gems to be inserted anywhere in the list by specifying the integer index of the gem to insert before. You are required to implement this using a *linked list* data structure you create. Each `Node` in the linked list will contain a `Gem` and a reference to the next `Node` in the list. Within `GemList.java`, define a nested class `Node` in the standard way:

```
private class Node
{
    private Gem  gem;
    private Node next;
}
```

Your `GemList` data type must implement the following API (see the **FAQ** for more help):

```
class GemList
---------------------------------------------------------------------------------------------------
     int size()                          // return the number of gems in the list
    void draw(double y)                  // draw all gems in the list at the given Y-coordinate
                                         //   see FAQ for help drawing gems at a particular loc.
  String toString()                      // return a string representation of the list
    void insertBefore(Gem gem, int index) // insert the given gem before the 0-based index
                                         //   if index is >= size, the new gem is inserted at end
     int score()                         // calculate the total score of the list
```

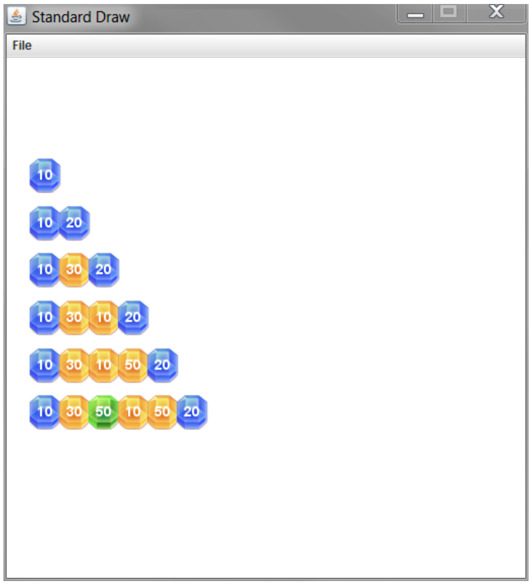
As mentioned previously, to score a list of gems, you need to apply score multipliers for any blocks of gems of the same color. Given the following Gem list:



This gem list would be scored as follows:

- First block one green gem, 25 = 1 * (25)
- Second block one blue gem, 35 = 1 * (35)
- Third block one green gem, 40 = 1 * (40)
- Fourth block two blue gems, 50 = 2 * (0 + 25)
- Fifth block four yellow gems, 280 = 4 * (50 + 0 + 0 + 20)
- Sixth block one blue gem, 40 = 1 * (40)
- **Total 470 (25 + 35 + 40 + 50 + 280 + 40)**

A `main` method containing various tests has been provided; feel free to add tests as you see fit. Here is the console and `StdDraw` output from our program:



Console output:

```
<none>
size = 0, score = 0

BLUE
size = 1, score = 10

BLUE -> BLUE
size = 2, score = 60

BLUE -> ORANGE -> BLUE
size = 3, score = 60

BLUE -> ORANGE -> ORANGE -> BLUE
size = 4, score = 110

BLUE -> ORANGE -> ORANGE -> ORANGE -> BLUE
size = 5, score = 300

BLUE -> ORANGE -> GREEN -> ORANGE -> ORANGE -> BLUE
size = 6, score = 230
```

(Advanced) Game improvements

Have some time and want to make a better game? Try the following:

- Create a new version of the game in which the second player is the computer. When it is the computer's turn, it should evaluate the best possible location to put the gem. It should maximize the difference between the computer's score and the human's score.

    - You'll probably want to implement a method in your GemList class that lets you delete an item at a specific index. This will allow you to temporarily insert the current gem into all possible locations in both players' gem lists.

- Make the game more interesting by adding additional types of playing pieces. For example, there could be a *bomb* piece that can be used to blow up part of your opponent's row. Other ideas would be a *wildcard* piece that matches any color or that increases a block's score multiplier.

- Change the game so you only score points when you make a run of so many gems. Once a run is made, the gems blow up and the player scores points based on the gems in the run. This will make the game continue as long as both players are able to complete runs to free up space in their gem row.

***Gem Matching** project from*
https://katie.mtech.edu/classes/csci135-online/assign/gem/