Programming Exercise:
Repulsive Behavior

Objective: To gain experience implementing classes and methods.

The Scenario. For this lab, we would like you to write a program that simulates the action of two bar magnets. Each magnet will be represented by a simple rectangle with one end labeled "N" for north and the other labeled "S" for south. Your program should allow the person using it to move either magnet around the screen by dragging it with the mouse. You know that opposite poles attract, while similar poles repel each other. So, if one magnet is dragged to a position where one or both of its poles is close to the similar poles of the other magnet, the other magnet should move away as if repelled by magnetic forces. If, on the other hand, opposite poles come close to one another, the free magnet should move closer and become stuck to the magnet being dragged.

To make things a bit more interesting one should be allowed to flip a magnet from end to end (swapping the poles) by clicking on the magnet without moving the mouse. This will provide a way to separate the two magnets if they get stuck together (since as soon as one of them is reversed it will repel the other).

Some Video Game Physics If you are worried that you might not remember (or never knew) enough about magnetic fields to do this assignment, relax. First, we will be providing you with most of the code that does the "physics". Even if you had to write all the code yourself, you still would not need a deep knowledge of magnetism and mechanics. Instead, you could exploit something every special effects expert or video game author must know. Most humans observe the world carelessly enough that even a very coarse approximation of reality will appear "realistic". Our code takes advantage of this by simplifying the behavior of our magnets. We never compute the force between two magnets, just the distance between them. If they get too close together, our code moves them apart or makes them stick together.

Despite the fact that we will provide most of the code to handle this approximate version of physics, there are two aspects of the magnet's behavior that will impact the code you write. The first is a simplifying assumption. The magnets will not be allowed to rotate. They only slide up, down and across the screen.

More significantly, there is one aspect of the behavior of the real magnets that we must model fairly well. Above, we said that we just compute the distance between two magnets. This would not really be accurate enough, since it is not the distance between the magnets that matters, but the distances between their similar and opposite poles.

Consider the two pairs of magnets shown below:

The magnets shown in the left pair are the same distance apart as the magnets in the right pair. In the pair on the left, however, the opposite poles are close together while the similar poles are relatively far apart. The situation is reversed in the other pair. In one case, one would expect the magnets to attract, in the other to repel.

So it is the poles rather than the magnets that really matter when deciding whether something should be attracted or repelled. As a result, instead of just manipulating magnet objects in your program, you will also need objects that explicitly represent poles.

Design of the program.   We will help you design this program by identifying the classes and methods needed. In particular, you will need two classes named `Magnet` and `Pole` and a class that is an extension of `WindowController`. We will provide the code for the `Pole` class. You will write the other two class definitions.

Pole.   You will be able to use the `Pole` class we have defined much like one of the built-in graphics classes provided by objectdraw. In this handout, we explain how to construct a new `Pole` and describe the methods that can be used to manipulate `Pole`s. You can then write code to work with `Pole`s just as you wrote code to work with `FilledRect`s. We will see, however, that the interaction of `Pole`s with the rest of your program is a little more complex than that of rectangles.

A `Pole`'s constructor will expect you to specify the coordinate position at which it should initially appear and the label that should be displayed (i.e. "N" or "S"). It will also require you to provide as parameters the `canvas` and the `Magnet` to which the new pole will belong. The signature of the constructor for the `Pole` class is:

```
public Pole( Magnet container,
             double x, double y,
             String poleName,
             DrawingCanvas canvas )
```

Since you will usually create the `Pole`s within the code of the `Magnet` constructor, the name `this` will refer to the `Magnet` that contains the `Pole`. Thus, the code to construct a `Pole` might look like:

```
new Pole( this, poleX, poleY, "N", canvas);
```

where `poleX` and `poleY` are the coordinates around which the label "N" should be displayed.

The `Pole` class provides several methods similar to those associated with graphical objects. In particular, `Pole`'s methods will include `getX`, `getY`, `getLocation`, and `move`, which all behave like the similarly named methods associated with basic graphic classes.

In addition, the `Pole` class has two more specialized methods: `attract` and `repel`. Each of these methods expects to be passed the `Pole` of another magnet as a parameter. If you say,

```
somePole.attract( anotherPole )
```

then `somePole` and `anotherPole` should have opposite polarities. If `somePole` is a north pole, then `anotherPole` must be a south pole and vice versa. The `repel` method, on the other hand, assumes that the pole provided as its parameter has the same polarity as the object to which the method is applied. Therefore, if you write:

```
somePole.repel( anotherPole )
```

and `somePole` is a north pole, then `anotherPole` should also be a north pole.

Each of these methods checks to see if the two `Pole`s involved are close enough together to exert enough force to move the magnets to which they belong. If so, they use the `move` and `moveTo` methods of the magnets to either bring the magnets closer together or move the free magnet so that they are far enough apart that they would no longer interact.

The good news is that the code for all the methods described above will be provided to you.

In summary, the `Pole` class provides the following methods. Note that we have given you complete method headers here, illustrating the format to follow in defining your own methods. Think carefully about how you will invoke each of the following methods.

- `public double getX()`

- `public double getY()`

- `public Location getLocation()`

- `public void move( double xoff, double yoff )`

- `public void attract( Pole opposite )`

- `public void repel( Pole similar )`

Important: Do not modify the provided `Pole` *class!*

Design of part 1   For the first part of this program, you should just worry about creating the magnets and moving them around. We'll worry about their interactions (attracting and repelling) later.

The key to this lab is the design of the `Magnet` class. A magnet is represented by a `FramedRect` and two poles. To ensure that our `Pole`s work well with your `Magnet`s, each magnet should be 150 by 50 pixels. The poles should be located near each end of the magnet. We recommend locating them so the distance from the pole to the closest end, top, and bottom, are all 1/2 the height of the magnet (*i.e.* 25 pixels away from each).

Your `Magnet` class will have to provide the methods that will enable someone running your program to drag magnets about within a window. The `Magnet` class will have to include the following methods:

- `public void move( double xoff, double yoff )`

- `public void moveTo( Location point )`

- `public Location getLocation()`

- `public boolean contains( Location point )`

These methods should behave just like the corresponding methods for rectangles and ovals. In particular, the offsets provided to the `move` method are `double`s, `someMagnet.getLocation()` should return a `Location` value, and `someMagnet.contains(point)` should return a `boolean`. (You will add other methods later, but we'll postpone discussing them until you need them.)

In order to write these methods, your magnet will need to contain several instance variables. A magnet should consist of a rectangle and two poles, and you will need instance variables for each of those. The constructor for a `Magnet` needs the following parameters:

- Coordinates of the upper-left corner of the magnet,

- The canvas that will hold the magnet

The header of the constructor for the `Magnet` class should be:

- `public Magnet(Location upperLeft, DrawingCanvas canvas)`

It should construct the framed rectangle forming the outline of the magnet (using the parameters), and should create two poles in the correct positions inside the magnet (see the earlier discussion on the constructor for `Pole`).

Once these instance variables have been declared and initialized, writing the methods should be easy. The `move` and `moveTo` methods should simply move the rectangle and poles to their new positions. The `move` method is pretty straightforward as all three items get moved the same distance, but `moveTo` takes a little thought as the `Pole` class does not have a `moveTo` method. Instead you'll need to calculate how far to move it. (Hint: check to see how far the rectangle is moving from its current position.) The method `getLocation` will simply return the location of the rectangle, while a magnet `contains` a point exactly when the rectangle does.

When you have this much of the `Magnet` class written, you can test it by writing `MagnetGame`, an extension of the `WindowController` class that creates a magnet, and then write methods `onMousePress` and `onMouseDrag` that will allow you to drag it around. Then, run your program to test and, if necessary, debug it.

Once `onMousePress` and `onMouseDrag` work, it should be pretty easy to add a second magnet and be able to drag either of them around. We suggest declaring a variable (`movingMagnet`) that can be associated with the appropriate magnet and used to remember which magnet to move whenever the mouse is dragged. This variable will be useful in other parts of your assignment as well.

Before coming to lab, you should sketch out a plan or design for the work you will complete. The design should explain how you plan to organize your `Magnet` and `MagnetGame` classes to accomplish the actions required of the first part of this lab only. You should write (in English, not Java) your plan for how each method will provide the necessary behavior. You should also describe (in English) what variables you feel are necessary for each class. This level of preparation will allow you to progress much more quickly in lab.

Part 2: Flipping the magnet When you click on a magnet, it should reverse its north and south poles. Add a method named `flip` to the class `Magnet` that interchanges the north and south poles. Remember that you can move a `Pole`, and one possible way to implement `flip` is to just move the north pole to the south pole's position and vice versa.

Add an `onMouseClick` method to your `MagnetGame` class that invokes `flip`.

Part 3: Interacting magnets Finally, after you `move` or `flip` a magnet, you will need to tell the magnet to check if it is close enough to the other magnet to move it. To make this possible, include a method named `interact` in your `Magnet` class. The method `interact` should be invoked on the moving (or changing) magnet, and should take as a parameter the `Magnet` that has not just been moved or flipped. It should effect the interaction by calling the `attract` and `repel` methods of its poles with the poles of the other magnet passed as parameters appropriately. For simplicity, you might want to just check for attraction first, and only worry about repelling after the attraction works correctly.

When writing `interact` you will discover you need to add two more methods in the `Magnet` class to enable you to access the other magnet's poles: `getNorth` and `getSouth`. Both of these methods will return objects belonging to our `Pole` class. Also, note that the `attract` method that we have provided in the `Pole` class calls the `moveTo` method that you must define in the `Magnet` class. If you do not fill in the body of this method correctly, attraction will not work properly.

You will need to call the `interact` method every time one of the magnets is either moved or flipped. Because you want to invoke the `interact` method on the magnet that moved and provide the other magnet as the parameter, you will need to keep track of which is which. As we suggested above, whenever you start dragging a magnet (i.e., in the `onMousePressed` method), you should associate a name with the moving magnet. You will also find it convenient to associate a name with the resting magnet in order to call your `interact` method appropriately.

When your program is finished, your `Magnet` class should have a constructor and method bodies implemented for `getLocation`, `move`, `moveTo`, and `contains`, for which headers were provided above. In addition, you will need to provide the methods `interact`, `getNorth`, `getSouth`, and `flip`. You should think carefully about the structure of the method headers for each of these. To help you in formulating your ideas, the following gives typical uses of the methods:

- `someMagnet.interact( otherMagnet ); // someMagnet & otherMagnet are magnets`

- `Pole theNorthPole = someMagnet.getNorth();`

- `Pole theSouthPole = someMagnet.getSouth();`

- `someMagnet.flip(); // someMagnet is a magnet`

Submitting Your Work Before submitting your work, make sure that each of the .java files includes a comment containing your name. Also, before turning in your work, be sure to double check both its logical organization and your style of presentation. Make your code as clear as possible and include appropriate comments describing major sections of code and declarations.

Quick Reference of the Pole Class This section provides no new information. It is a quick reference to the constructor and methods provided in the `Pole` class that you will be using.

Constructor To create a new pole:

```
public Pole (Magnet parent, double x, double y, String name, DrawingCanvas canvas)
```
Example Usage
```
Pole myPole = new Pole (this, xLoc, yLoc, "N", canvas);
```

Accessor Methods To get information about a pole:

Getting the x coordinate of the pole's center:
```
public double getX()
```
Example Usage
```
double x = somePole.getX();
```

Getting the y coordinate of the pole's center:
```
public double getY()
```
Example Usage
```
double y = somePole.getY();
```

Getting the coordinate pair of the pole's center:
```
public Location getLocation()
```
Example Usage
```
Location loc = somePole.getLocation();
```

Mutator Methods To modify a pole:

Moving the pole relative to its current location:
```
public void move (double xoff, double yoff)
```
Example Usage
```
somePole.move (xOffset, yOffset);
```

Attracting another pole if close enough:
```
public void attract (Pole oppositePole)
```
Example Usage
```
somePole.attract (anotherPole);
```

Repelling another pole if close enough:
```
public void repel (Pole similarPole)
```
Example Usage
```
somePole.repel (anotherPole);
```