

Lab 3

Magnets

Objective To gain experience implementing classes and methods.

The Scenario This week, you will write a program that simulates the action of two bar magnets. Each magnet is represented by a simple rectangle with one end labeled “N” for north and the other labeled “S” for south. Your program allows the person using it to move either magnet around the screen by dragging it with the mouse. As with real magnets, opposite poles attract, while similar poles repel each other. 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 moves away as if repelled by magnetic forces. On the other hand, if opposite poles come close to one another, the free magnet moves closer and becomes stuck to the magnet being dragged. A magnet can be flipped from end to end (swapping the poles) by clicking on it. This provides a way to separate two magnets if they get stuck together (since as soon as one of them is reversed it will repel the other).

See the handouts web page for a demo. We will also provide a starter project. It is described in the “Getting Started” section below.

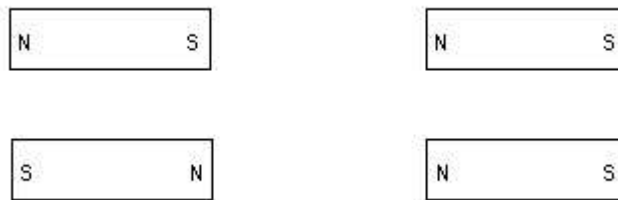
Note: You must bring a program design to lab this week!

How it Will Work

Do not worry if you do not remember (or never knew) the details of magnetic fields. We will provide 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 people 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 move too close together, our code moves them apart or makes them stick together. We assume magnets can not be rotated; They only slide up, down and across the screen.

There is one aspect of the behavior of the real magnets that we must model well. Above, we said that we just compute the distance between two magnets. This would not really be accurate enough, since it is not just the distance between the magnets that matters, but also 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 the case on the left, one would expect the magnets to attract, and in the case on the right, 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.

We will help you design this program by identifying the classes and methods needed. In particular, you will need two classes: `Magnet` and `Pole`. Additionally, you will need 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.

The Pole Class

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

A `Pole`'s constructor requires 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 also requires 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 thus:

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

Since you will usually create the `Poles` 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:

```
public Magnet(...formal parameters...) {
    ...
    northPole = new Pole(this, poleX, poleY, "N", canvas);
    ...
}
```

where `poleX` and `poleY` are the coordinates at 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`, all of which 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 `Poles` 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 we have already written the code for all the methods described above and will provide these methods to you.

In summary, the `Pole` class provides the following methods. Note that we have given you complete method signatures (or 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)`

Do not modify the provided `Pole` class!

Implementation Strategy

Step 1: Basic Magnets 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. To reinforce the notion that preparing for your lab section greatly increases the value of the time spent in lab, we want you to come to lab with a written design.

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 `Poles` work well with your `Magnets`, 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 provide methods that enable someone running your program to drag magnets around within a window. The `Magnet` class should include the following methods:

- `public void move(double xoff, double yoff)`
- `public void moveTo(Location point)`
- `public Location getLocation()`
- `public boolean contains(Location point)`

The signatures for these methods are already included in the starter file for the `Magnet` class. These methods should behave just like the corresponding methods for rectangles and ovals. In particular, the offsets provided to the `move` method are doubles, `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.

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

- Coordinates of the upper-left corner of the magnet.
- The canvas that will hold the magnet.

The signature of the `Magnet` constructor should be:

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

It should construct the framed rectangle forming the outline of the magnet (using the parameters) and the 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 will 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` should return the location of the rectangle. A magnet contains a point exactly when the rectangle does.

Now is a good time to test your `Magnet` class. To do this, you need to write a controller. We provide you with the stub of a `WindowController` class `MagnetGame`. To start with, have `MagnetGame` create and store a single magnet. Then write methods `onMousePress`, `onMouseDrag`, and `onMouseRelease` that allow you to drag it around.

Once `onMousePress`, `onMouseRelease`, and `onMouseDrag` work, you should add a second magnet. 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 project later.

While writing the methods for the `Magnet` class, you probably noticed that two additional methods are already included there. They are `getWidth` and `getHeight`. These are used by the `Pole` class in ensuring that the methods `attract` and `repel` draw the magnets appropriately in the window.

Preparing Your Design Before Lab: As mentioned earlier, **you should bring a design with you to lab**. The design should show us how you plan to organize your `Magnet` and `MagnetGame` classes to accomplish the actions required for “Step 1” of this lab only. We have told you what methods each class should have and the behavior that they should provide. 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 instance variables you feel are necessary for each class. This level of preparation will allow you to progress much more quickly in lab so that you can take better advantage of the presence of the instructors and TAs.

You may use the design template available on the handouts page to help you organize your thoughts, as you did last week for the laundry lab. We would like you to fill out a design template for each of the two classes you will write: `Magnet` and `MagnetGame`. We have included at the end of this handout the design of the laundry lab as another example of a program design.

Step 2: Flipping the magnet When you click on a magnet, it should reverse its north and south poles. Add a `flip` method to the `Magnet` class that transposes 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 `onMouseClicked` method to your `MagnetGame` class that invokes `flip`.

Step 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 work on attraction first, and only worry about repelling after the attraction works correctly.

When writing the `interact` method, you will discover that 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 send the `interact` message to 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 signatures were provided. In addition, you will need to provide the methods `interact`, `getNorth`, `getSouth`, and `flip`. You should think carefully about the structure of the method signatures for each of these. To help you in formulating your ideas, the following gives typical uses of the methods:

- `someMagnet.interact(otherMagnet);`
- `Pole theNorthPole = someMagnet.getNorth();`
- `Pole theSouthPole = someMagnet.getSouth();`
- `someMagnet.flip();`

Getting Started

To download the starter project, visit

<http://www.cs.williams.edu/~cs134/>

and then follow the link to the Handouts page. Click on the link for “Starter Code” under Lab 3.

This will download a file archive called `Lab3Magnets.tar.gz` to your `cs134` folder. The files of this archive will then be extracted to a folder in your `cs134` folder called `Lab3Magnets`. (If this does not happen automatically, simply double-click on the downloaded `Lab3Magnets.tar.gz` file to extract the archive.) You may delete the `Lab3Magnets.tar.gz` file at this point. Rename the “`Lab3Magnets`” folder to include your last name, as usual, and open the project folder in BlueJ.

The starter project contains several files intended to hold Java code. The file `MagnetGame.java` should be used to write the extension of the `WindowController` that will serve as your “main program.” The `Magnet.java` file should be used to hold your code for the `Magnet` class. Both of these files contain skeletons of the code that you need to complete. Finally, `Pole.java` holds our implementation of the `Pole` class. Remember, you should not change `Pole`.

Submitting Your Work

Once you have saved your work in BlueJ, please perform the following steps to submit your assignment:

- First, return to the Finder. You can do this by clicking on the smiling Macintosh icon in your dock.
- Click “Go” and select “Connect to Server.”
- For the server address, type in “Cortland” and click “Connect.”
- Select the button next to “Guest” and click “Connect.”
- A selection box should appear. Select “Courses” and click “Ok.”
- You should now see a Finder window with a “cs134” folder. Open this folder.
- You should now see the drop-off folders for the three labs sections. As with last week, drag your program folder into the appropriate lab section folder. When you do this, the Mac will warn you that you will not be able to look at this folder. That is fine. Just click “OK.”

- Log off of the computer before you leave.

You can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Lab, and 11 p.m. Thursday for those in the Tuesday Lab). If you submit and later discover that your submission was flawed, you can submit again. We will grade the latest submission made before the 11 p.m. deadline. The Mac will not let you submit again unless you change the name of your folder slightly. It does this to prevent another student from accidentally overwriting one of your submissions. Just add something to the folder name (like “v2”) and the resubmission will work fine.

Grading Guidelines

As on labs, we will evaluate your program for both style and correctness. Here are some specific items to keep in mind and focus on while writing your program:

Style

- Descriptive comments
- Good names
- Good use of constants
- Appropriate formatting

Logical Organization

- Good use of boolean expressions
- Not doing more work than necessary
- Using most appropriate methods

Correctness

- Drawing magnets correctly at startup
- Dragging a magnet
- Ability to move either magnet
- Moving a magnet to the right place when attracted
- On attraction, moving the magnet not pointed to
- Flipping a magnet
- Attracting and repelling at the right times
- No other problems

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);
```