CS 051G Homework Laboratory # 11
Repulsive Behavior

Objective: To gain experience implementing classes and methods in Java

—

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).

Wait a minute, didn't we already do this lab? Yes, but now you are doing it in Java instead of Grace. However, you get to to use your Grace solution to guide your Java implementation!

Using Eclipse to write Java programs You must follow the following instructions carefully as they are different from those that worked in Grace. First, copy the folder Lab11−MagnetsRedux from the folder cs051G/labs. Copy this into your CSC51GWorkspace folder. Don't neglect to copy this folder. If you don't copy it you won't be able to save your program and may lose it.

Now click on the Eclipse item in the menu bar at the bottom of the screen. When it comes up, it may ask you which folder to use at the workspace. Tell it to use your CSC51GWorkspace folder.

For the very first lab, you are going to need to do some configuration of Eclipse to make it work better for the rest of the term. Here is what you need to do:

1. From the eclipse menu item on the tool bar at the top of the screen, select "Preferences...". Along the left edge of the pop-up window, click on the triangle next to the word "Java". That will open a submenu. Click on the triangle next to "Build Path" and then click on "Classpath Variables".

2. On the right side of that window, click on "New...". A new window will pop up. In the "Name" field type "OBJECTDRAW" (in all caps, but without the quotes). In the "Path" field type `/common/cs/cs051/objectdraw/objectdraw.jar` This must be exact or you will not be able to run objectdraw programs. (Alternatively, you can click on the "File..." button and navigate to your CS51Workspace folder and then objectdraw and the objectdraw.jar file.)

   Once this is done, click OK a few times until you are back on the preferences window.

3. We need you to do one last thing to eliminate an annoying warning message. In the same left pane of the Preferences window as before, under Java, click on the triangle next to "Compiler" and then on "Errors/Warnings". On the right side of the window you will see categories of issues. Click on the triangle next to "Potential programming problems" and scroll down to "Serializable class without serialVersionUID" and select "Ignore" as the option. Click OK and you are done with this one time setting of preferences and ready to work!

In the Eclipse window, pull down the "File" menu and select "Import..." from near the bottom of the menu. Select "Existing Project into Workspace" and click the "Next" button. [You may have to click on the triangle next to "General" to see the "Existing Project . . . " entry.] Now, click the "Browse" button to the right of the "Select root directory" entry, and navigate to your CSC51GWorkspace folder. (Hint: double-click on "Desktop" if you seem lost.). Double click on CS51Workspace and then click on "Lab11-MagnetsRedux'. Push "OK." Then push the "Finish" button.

The eclipse workspace window may be obscured by a welcome screen. If so, click on the "x" to remove it.

On the left side of the Eclipse window, in an area called the "Package Explorer" you should now see an entry for "magnets". Click once on the triangle to the left of "magnet" and then once on the triangle to the left of "(default package)". Then double-click on "MagnetGame.java".

This window has three main regions. On the left, it displays the names of files and their contents in an outline form. The large area on the top right side of the screen displays the contents of "MangetGame.java" file. You will edit your files here.

The simplest way to compile and run your program is using the white triangle in the green circle above your program file. The first time you want to run your program you should make sure that the main program (the one extending `WindowController`) is open and showing in the editor window. Then use the white triangle as a popup menu. When you select the menu, one of the options should be `Run as ...`. Select that and follow the walking menu to `Java application`. Once you have done this, the next time you can just click on the button and it will compile your program.

This "Run" button will first compile your program and then, assuming there are no errors, will begin executing it. One nice thing about Java is that it will statically type check your program to see if you are doing something that makes no sense from the point of view of the types you have declared. You will find lots of errors that way, and the system will print all of them that it can detect rather than being limited to one like the Grace compiler.

Compile-time errors will show up as red X's in the left margin of the editor window. If you let the mouse hover over the x, it will show a brief error message. You will often get even more information if you click on it.

OK, that should be enough to get you going in using Eclipse. Here are some last tips to remind you of the differences between Java and Grace:

1. There are minor differences in the names of methods in the Java objectdraw library compared to Grace. Keep the objectdraw API documentation open at http://www.cs.pomona.edu/classes/cs051/handouts/objectdraw-api.html.

2. You must put a semicolon at the end of lines in Java. However, never put a semicolon before a { or after a }.

3. Parameterless method requests must include () even though there are no parameters.

4. Every Java class must be in a separate file.

5. Every instance variable, constant, and method must have a visibility annotation. For now, use one of `private` and `protected`. Do not put these in front of local variables.

6. Types are written before the identifiers in declarations. E.g., `int x = 5;`.

7. Assignments to variables are written with `=` instead of `:=`

8. Constants are written in all caps and are declared as `static final` if outside of methods. Inside methods (local constants), just declare them as `final`.

9. Uninitialized instance variables are given the value 0 (if a primitive type) or null (if an object type). If you forget to initialize an instance variable you will get a "null pointer error". Local variables are not automatically initialized. You must make sure to initialize them.

10. In Java we use keyword `this` in place of `self`. As you may recall, you will have to pass `this` as a parameter to the `Pole` constructor.

OK, the rest of this handout just repeats the previous description of Magnets, except this time from the point of view of Java. You do not have to bring in a design for this week's lab, but you should bring in a(n electronic) copy of your magnet program in Grace as you will use that to guide your Java solution.

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 we constructed for the class. 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 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 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. Just like last week, we want you to come to lab with a written design. At the end of this section, we will be more specific about what you should bring with you to lab. To give you a better sense of what we mean by a written design, you can see a sample design for the first part of the laundry lab at the end of this document.

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 )`

The headers 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 `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. To run your program, you need to create a new applet configuration using the Run... command from the Run menu as in the past two labs. For this lab, set the width of the applet to 420 and the height to 400 on the Parameters tab.

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.

As you were 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.

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 of the first part 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 start with the file MagnetsLabDesignStarter.txt and modify it to contain the complete design. (Notice that we left out the `Pole` class, because we are providing the complete code for it.)

Be sure to describe in your design (in English) what variables you feel are necessary for each class as well as filling in a detailed design for all methods of classes `Magnet` and `MagnetDesign`. 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. This week your design will be worth 10% of your lab grade.

Part 2: Flipping the magnet When you click on a magnet, it should reverse its north and south poles. Add method `flip` to 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 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 headers 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 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`

Getting Started 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 will initially contain skeletons of the code that you will need to complete. The final ".java" file will be `Pole.java`. It will hold our implementation of the `Pole` class. Remember, you should not change `Pole`.

Due Dates This assignment is due at 11 pm on Monday evening.

Submitting Your Work Before submitting your work, make sure that each of the .java files includes your name in the comment heading up the class. 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. Use the Format command in the Source menu to make sure your indentation is all consistent.

Turning in your program Your program is due at 11p.m. on Monday evening. Turning in your program is a little more elaborate in Eclipse and Java than it was for Grace.

- First, return to Eclipse and make sure you included your name and course number in a comment at the start of each of the Java classes.
- Next, click on the "magnet" project in the Package Explorer panel on the left side of the Eclipse window.
- Now, select "Export" from the "File menu.
- Select "File system" in the dialog box and click next.

- Make sure the folder with your project has a check mark next to it. (That will make all the files on the right have checks too!)
- Click the "Browse" button next to the "To directory:" entry.
- Double click on "Desktop".
- Click the "Finish" button. It will create a new folder on your desktop.
- Quit Eclipse.
- Click on the new folder's name and type in a name for the folder that identifies you and the lab you are working on, such as "*Yourname*-Lab11" or "*Yourname*-magnet" and then click "OK". (Dashes in names are OK, but don't include spaces or periods.)
- Open the cs051G folder icon to find the "dropbox" folder.
- Drag your new folder into the dropbox folder. When you do this, the computer will warn you that you will not be able to look at this folder. That is fine. Just click "OK".
- The new folder will still show up on your desktop. Drag it into the trash both to save space and to keep anyone from copying it. (Files in your CSC 051 folder are protected so that others cannot read them.) Select "empty trash" from the file menu to completely delete those files. Do NOT throw away the version of the program in your workspace. You should save that so you can study it later.

If you should accidentally turn in a bad version of your program, you may drag another copy in as long as you change the name to be slightly different from the one you used before (e.g. Jane Doe - lab 11a). The new name should make it clear which is the newer version.

Table 1: Grading Guidelines

Value	Feature
	Design (2 free pts!)
	Readability (6 pts total)
2 pts.	Descriptive comments
1 pts.	Good names
2 pts.	Good use of constants
1 pts.	Appropriate formatting
	Code Quality (4 pts total)
2 pts.	Good use of boolean expressions
1 pt.	Not doing more work than necessary
1 pt.	Using most appropriate methods
	Correctness (8 pts total)
1 pt.	Drawing magnets correctly at startup
1 pt.	Dragging a magnet
1 pt.	Ability to move either magnet
1 pt.	Moving a magnet to the right place when attracted
1 pt.	On attraction, moving the magnet not pointed to
1 pt.	Flipping a magnet
1 pt.	Attracting and repelling at the right times
1 pt.	No other problems

Quick Reference for 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);
```