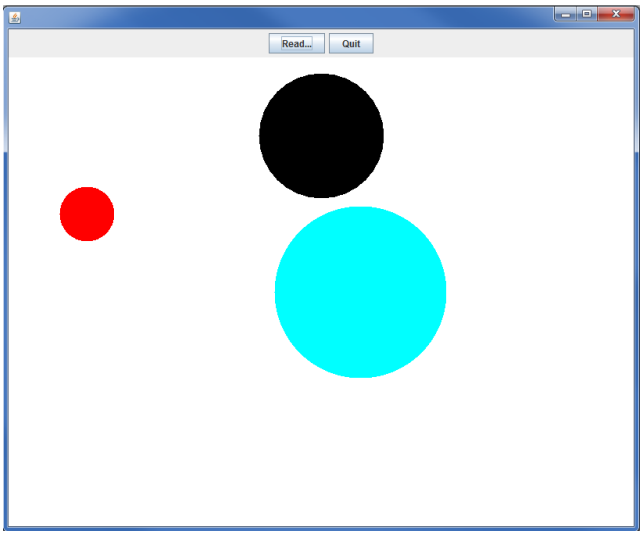
CS 46B
Lab 4: October 4 & 7, 2016



In this lab you'll use your knowledge of Java I/O and exceptions to complete a painting program that draws dots on the display.

Work in pairs. One of you will be the "driver", who types into Eclipse. The other is the "scribe". Alternate jobs, week by week.

You will both submit lab reports.

Driver: At the top of your lab report write

CS 46B Lab Report
Your name
I was the driver, XxxxYyyy was the scribe.

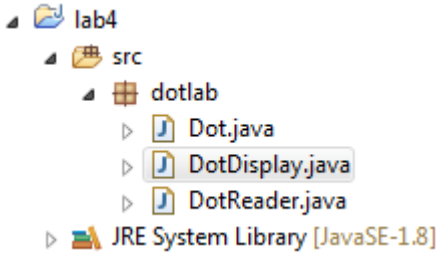
Scribe: At the top of your lab report write

CS 46B Lab Report
Your name
I was the scribe, XxxxYyyy was the driver.

As you work through the lab assignment, instructions/questions highlighted in yellow should be discussed by both of you and addressed by the driver in the driver's report; instructions/questions highlighted in blue should be discussed by both of you and addressed by the scribe in the scribe's report.

Part 1: Fixing the Dot class

STEP 1: Create a new Java Project called 'lab4'. In lab4->src, create the dotlab package and drag in the Dot.java, DotDisplay.java, and DotReader.java files into the package. Your Package Explorer should look like this (possibly with red error marks):



STEP 2: Let's edit the Dot class to properly store the essential data to draw a dot on the screen. In Dot.java, below the LEGAL_COLOR_NAMES code, **add the instance variables necessary to store the color name (a String), x and y coordinates (ints), and the radius (another int). Write a getter method for each of the instance variables. The getters should be called getColorName(), getX(), getY(), and getRadius().**

STEP 3: Immediately after the instance variables, **make a constructor for the Dot class that takes a color name, an x value, a y value, and a radius and sets the corresponding instance variables to these values.**

STEP 4: Already provided in this class are the color names we'll allow for the dots as a String[]. This will be important because we will only want to allow dots to be drawn of these colors. **In your constructor, before setting your color instance variable, ensure that it is a color that can be found in LEGAL_COLOR_NAMES. If the color cannot be found in LEGAL_COLOR_NAMES, throw an IllegalArgumentException whose message reports the name and explains that it is a bad color name.**

Driver: Paste your Dot constructor into your report

STEP 5: Let's test that we're catching bad color names correctly. **Create a main() method in Dot.java that creates 2 instances of the Dot class. The first Dot should use a legal color name with some legal values for its x, y, and radius and should be created without a problem. The second Dot should use an illegal color name with some legal values for its x, y, and radius and should raise an exception.**

Driver: Paste the exception printed in the console into your report

STEP 6: Printing out an object can be extremely useful during debugging because it can provide clues about the cause of a bug. **In main(). delete the code that creates your illegal Dot, and**

print out the contents of your legal Dot using .toString(). (Remember, .toString() simply returns a string, it doesn't actually print it out)

Driver: Paste the contents of your Dot into your report

STEP 7: That's weird. That looks nothing like the values we provided our Dot. **Find the API page for the Object class and look under the toString() method.** (Note: there are lots of ways to learn what Object's toString() method does, including asking someone, googling, or checking StackOverflow. The only official, guaranteed correct, source of the information is the API page, and reading API pages is how professionals get their information.)

Scribe: Summarize what the documentation explained in your own words. Why are we getting what we're getting when we call .toString() on our Dot object? (You don't need to explain what a hexadecimal representation means)

STEP 8: Let's go ahead and override the .toString() method so that it'll print out something useful for our Dot object. **In Dot.java, write a .toString() method that will print out the dot color, x and y coordinate, and radius in a readable format.**

Driver: Paste your .toString() method into your report.

STEP 9: Let's make sure that the .toString() method is working correctly. **In your main() method of Dot.java, print out a valid Dot object.**

Scribe: What is printed out from your .toString() method?

Part 2: Fixing the DotReader class and creating the DotException class

STEP 10: Our next goal is to be able to read descriptions of dots from a file, and convert the descriptions to instances. Each line of our file will have 4 properties separated by commas: the color name, an X position, a Y position, and a radius. **In DotReader.java, write the method readDot() that returns type Dot, takes no arguments, and throws IOException. Within the readDot() method, use the BufferedReader instance variable provided to read a single line and break it at the commas into an array of Strings. For example, the input line "BLUE,100,200,300" should result in the array { "BLUE", "100", "200", "300" }. Note there are no blank spaces in thr input. If you have trouble with parsing the line, try looking up split() in the String API page; use "," as the regex arg. If the line read from the BufferedReader is null, indicating end-of-file, your readDot() method should also return null.**

STEP 11: Using the values in the String array, readDot() should return a new Dot using the four values. (Be careful about passing in the last 3 parameters. Remember that our Dot constructor is looking for int types, not Strings; you saw in lecture how to convert a String to an int.)

STEP 12: But wait, what if we have an irresponsible user that tries to add more properties, like flavor, onto one line? (Maybe that user likes Dots candies.) We have to check that our array has only 4 properties in it, no more, no less. If the array is a different size than what we expected, we'll throw our own exception: a DotException. **Create a new class called DotException that's a subclass of Exception, has a constructor that accepts a single String argument, and calls the superclass constructor using that String argument.**

STEP 13: Now that we have our own specific Exception we can use when making Dots, let's use it in our DotReader class. **In DotReader.java, make sure that the array from the parsed line is of exactly size 4. If it's not, throw a new DotException that gives a brief description and the line that's causing a problem. You'll have to change the readDot() declaration to declare that the method throws DotException. It's ok to have multiple exception types after "throws" – just separate the types with commas. To throw the exception, do something like this:**

```
DotException de = new DotException(a good message);
throw de;
```

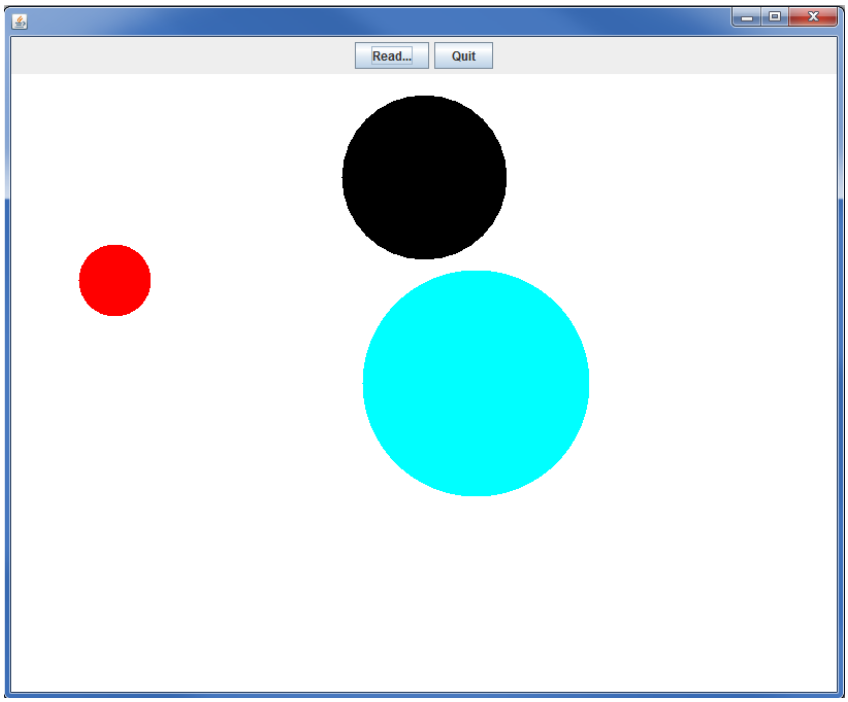
Scribe: Why is it useful to have created our own DotException? What happens if the readDot() declaration doesn't say that the method throws DotException?
Driver: Paste your readDot() method into your report

Part 3: Fixing the DotDisplay class

STEP 14: Now that we have a functional Dot and DotReader class, let's put those to work in making our application complete. Notice that DotDisplay doesn't compile because there's a call to a read() method that doesn't exist yet. You probably won't understand much about this source. That's ok, all you need to do is give it a read() method. **In DotDisplay.java create a private read() method that takes a File as an argument, returns nothing, and throws IOException and DotException. Within the method body, create a new FileReader that takes the argument file as a parameter, a new BufferedReader that takes the FileReader as a parameter, and a new DotReader that takes the BufferedReader as a parameter.**

STEP 15: We're going to read the data from our DotReader and keep constructing new Dots as long as we have information to read. **Use a while loop to continuously read from the DotReader until there is nothing else to read. Each pass through the loop should call addDot() with the newly read Dot as an argument. When the loop is done, close the BufferedReader and the FileReader.**

STEP 16: Let's test out our program. **Run DotDisplay.java and click 'Read...' Use the navigator to find the '3dots.txt' you downloaded earlier and open it. You should have a window that looks like this:**

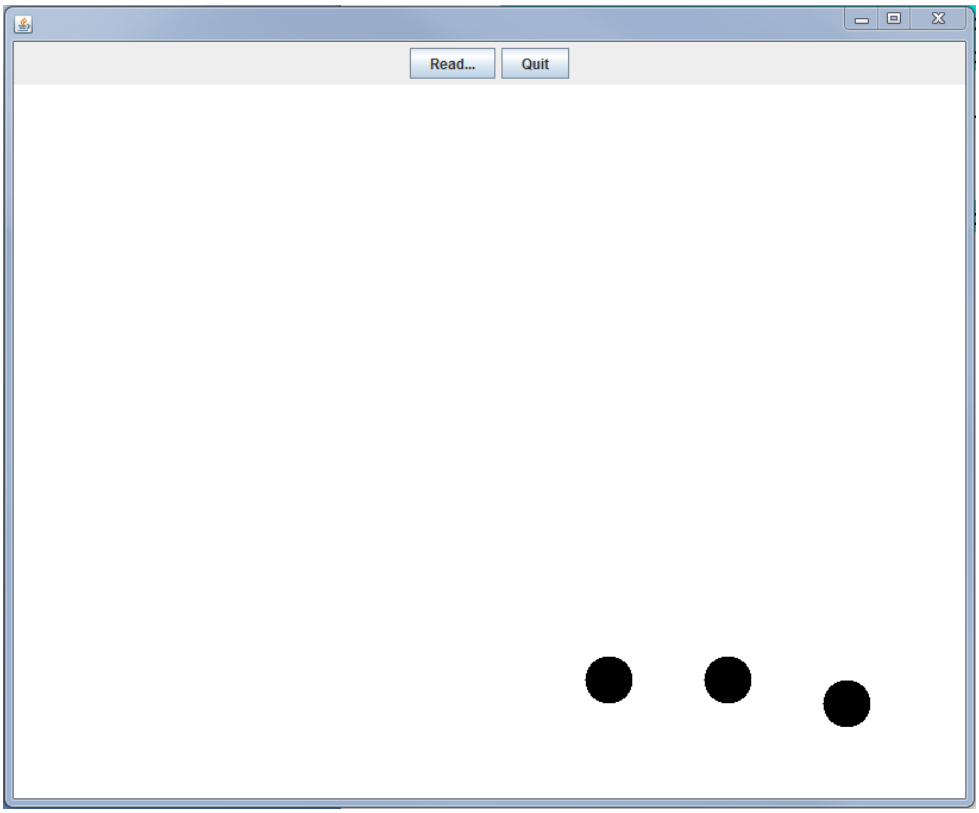


STEP 17: Great! Now let's make sure our IllegalArgumentException and DotException are working properly. **Edit the 3dot.txt so that one of the lines causes an IllegalArgumentException.** Driver: Paste the contents of your 3dot.txt. **Undo those changes**

then edit it once more so that one of the lines causes a DotException. Driver: Paste the contents of your 3dots.txt to raise a DotException.

Step 18: To ensure everything is working correctly, **edit 3dots.txt to make your own modern art. If you're proud of your art, email your 3dots.txt to philip.heller@sjsu.edu with both your names and an optional title.**

Driver: Paste the contents of your 3dots.txt to create your artwork and paste a screenshot into your report.



Awkward Silence