Nick Troccoli
CS 106A

Practice Final 2
August 16, 2017

Practice Final Examination #2

Final Exam Time: Friday, August 18th, 12:15P.M.–3:15P.M.
Final Exam Location: Various (see website)

Based on handouts by Marty Stepp, Mehran Sahami, Eric Roberts and Patrick Young

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the final examination. We do not guarantee that the number of questions in this sample exam will match the number on the real exam, nor that every kind of problem shown here will exactly match the kinds of problems shown on the final exam (though the real exam will be generally similar overall).

The final exam is open-textbook, closed-notes and closed-electronic-device. It will cover all material covered in the course, with an emphasis on material covered after the midterm. A "syntax reference sheet" will be provided during the exam (it is omitted here, but available on the course website). Please see the course website for a complete list of exam details and logistics.

General instructions
Answer each of the questions included in the exam. If a problem asks you to write a method, you should write only that method, not a complete program. Write all of your answers directly on the *answer pages provided for that specific problem*, including any work that you wish to be considered for partial credit. Work for a problem not included in a problem's specified answer pages will not be graded.

Each question is marked with the number of points assigned to that problem. The total number of points is 180. We intend for the number of points to be roughly comparable to the number of minutes you should spend on that problem.

In all questions, if you would like to use methods or definitions implemented in the textbook (e.g. from an example problem), you may do so by giving the name of the method and the chapter number in which that definition appears.

Unless otherwise indicated as part of the instructions for a specific problem, your code will not be graded on style – only on functionality. On the other hand, good style (comments, etc.) may help you to get partial credit if they help us determine what you were trying to do.

Blank pages for solutions omitted in practice exam
In an effort to save trees, the blank pages following each problem that would be provided in a regular exam for writing your solutions have been omitted from this practice exam.

Problem 1: Java expressions, statements, and methods (20 points)

Write the output produced when the following method is passed each of the following maps listed below. It does not matter what order the key/value pairs appear in your answer, so long as you have the right overall set of key/value pairs. You may assume when you iterate over the map, it is iterated over in the same order the key/value pairs are listed below.

```
private void collectionMystery2(HashMap<String, String> map) {
    ArrayList<String> list = new ArrayList<>();
    for (String key : map.keySet()) {
        if (map.get(key).length() > key.length()) {
            list.add(map.get(key));
        } else {
            list.add(0, key);
            list.remove(map.get(key));
        }
    }
    println(list);
}
```

a) {horse=cow, cow=horse, dog=cat, ok=yo}
b) {bye=hello, bird=dog, hi=hello, hyena=apple, fruit=meat}
c) {a=b, c=d, e=a, ff=a, gg=c, hhh=ff}
d) {karel=robot, robot=karel, daisy=dog, doggie=daisy}

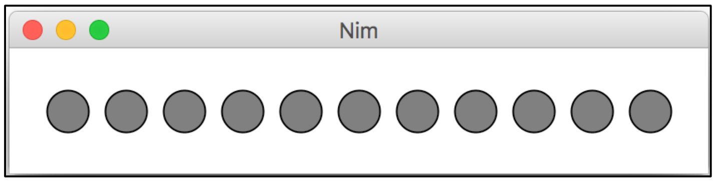
Problem 2: Nim (30 points)

Write a complete program named **Nim** that implements the graphical game Nim. In this game, two players start with a pile of 11 coins on the table between them. The players take turns removing 1, 2 or 3 coins from the pile. The player who is forced to take the last coin loses.

The following constants are provided for you (and do not have to define them in your answer):

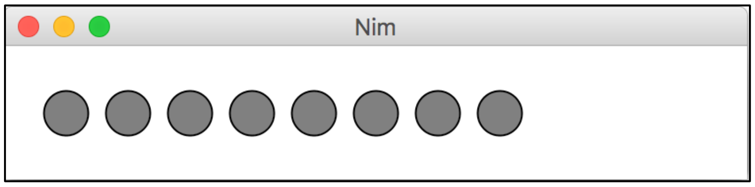
```
public static final int N_COINS = 11;          /* Number of coins     */
public static final int COIN_SIZE = 25;       /* Diameter of a coin  */
public static final int COIN_SEP = 10;         /* Space between coins */
```

At launch, your program should display a line of coins horizontally on the screen, like this:

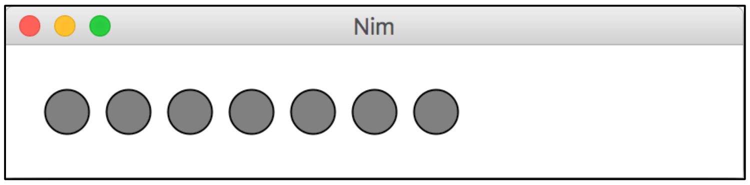


In constructing your display, you should make use of named constants provided for you. You should also make sure that each coin is filled in **GRAY** and outlined in **BLACK**. The line of coins should be centered both **horizontally** and **vertically** in the window, whatever its size may be.

When the game starts, you should assume that **Player 1** makes the first move, followed by **Player 2**, **then Player 1**, and so on. If the current player clicks the mouse in one of the three rightmost coins in the row, those coins disappear. For example, if a player clicked on the third coin from the right end, the program should respond by removing the **last three coins** from the display, like this:

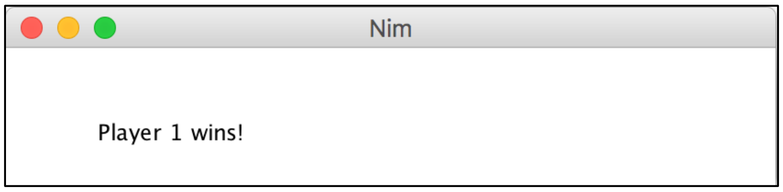


If the next player then clicked the rightmost coin, **only** that coin should go away:



If the mouse click does not occur inside a coin or if the coin is not one of the last three in the row, that click should **be ignored**.

At the end of the game, once all the coins are gone, you should display a **label with its left baseline at (50, 50)** saying which player won. For example:



Player 1 wins!

Problem 3: Sequences (20 points)

Write a method named **`contains`** that accepts two arrays of integers *a1* and *a2* as parameters and that returns a `boolean` value indicating whether or not *a2*'s sequence of elements appears in *a1* (`true` for yes, `false` for no). The sequence of elements in *a2* may appear anywhere in *a1* but must appear consecutively and in the same order. For example, if variables called **`a1`** and **`a2`** store the following values:

```
int[] a1 = {1, 6, 2, 1, 4, 1, 2, 1, 8};
int[] a2 = {1, 2, 1};
```

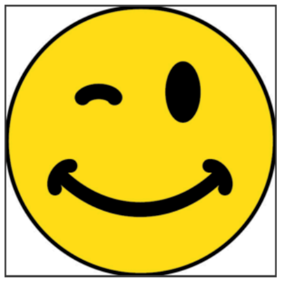
Then the call of **`contains(a1, a2)`** should return `true` because *a2*'s sequence of values is contained in *a1* starting at index 5. If *a2* had stored the values **`{2, 1, 2}`**, the call of **`contains(a1, a2)`** would return `false` because *a1* does not contain that sequence of values. Any two arrays with identical elements are considered to contain each other, so a call such as **`contains(a1, a1)`** should return `true`.

Constraints: You may not use any auxiliary data structures (arrays, lists, strings, etc.) to solve this problem. You may not use any Strings to help you solve this problem, nor methods that produce Strings such as `Arrays.toString`. Your method should not modify the arrays that are passed in. You may assume that both arrays passed to your method will have lengths of at least 1.

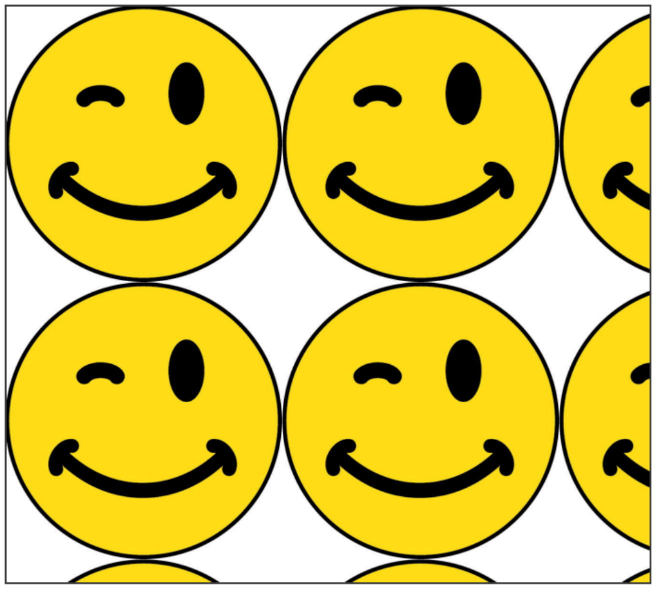
Problem 4: Image Tiling (25 points)

Write a method named `tile` that creates repeated copies of an image, like the algorithms seen in the ImageShop assignment. The method accepts three parameters: a `GImage`, and a width and height as integers. Your code should modify the image so that its pixels are looped and repeated ("tiled") over an area with the given width and height. The screenshots below show the state of an image before and after a call to your method. (The screenshots have thin black borders around their edges to help you see the bounds, but no black border pixels are present in the images.)

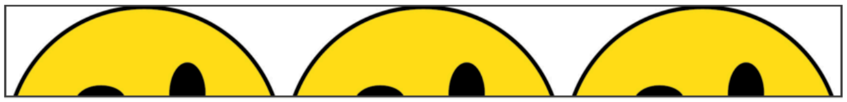
Before:



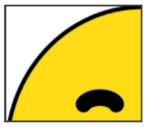
After `tile(myImage, 700, 630)`



After `tile(myImage, 900, 100)`



After `tile(myImage, 150, 121)`



Your code should work for an image of any size of at least 1x1 pixels. The width and height might not be equal. Note that the destination width and height integers passed might be larger or smaller than the image's original size. You may assume that the width and height passed will be positive integers.

Constraints: You may use **one** auxiliary 2D array to help you solve this problem. (Asking for the pixel array of an existing image does not count as an auxiliary data structure; when your code says 'new', that is what counts.)

Problem 5: Teacher (25 points)

Write a method named `teacher` that produces a mapping of students' grades. Your method accepts two parameters: a class **roster** telling you each student's name and percentage earned in the course, and a grade **mapping** telling you the minimum percentage needed to earn each unique grade mark. The class roster is a hash map from students' names (strings) to the percentage of points they earned in the course (integers). The grade mapping is a hash map from percentages (integers) to grades (strings) and indicates the minimum percentage needed to earn each kind of grade. The task of your method is to look at the student roster and grade mapping and use them to build and return a new `HashMap` from students' names to the letter grades they have earned in the class, based on their percentage and the grade mapping. Each student should be given the grade that corresponds to the highest value from the grade mapping that is less than or equal to the percentage that the student earned.

Suppose that the class roster is stored in a map variable named `roster` that contains the following key/value pairs, and that the grade mapping is stored in a map variable named `gradeMap` that contains the key/value pairs below it:

```
roster:
  {Mort=77, Dan=81, Alyssa=98, Kim=52, Lisa=87, Bob=43, Jeff=70, Sylvia=92,
Vikram=90}

gradeMap:
  {52=D, 70=C-, 73=C, 76=C+, 80=B-, 84=B, 87=B+, 89=A-, 91=A, 98=A+}
```

The idea is that Mort earned a C+ because his grade is at least 76%; Dan earned a B- because he earned at least 80%; and so on. If a given student's percentage is not as large as any of the percentages in the map, give them a default grade of "F". So your method should build and return the following map from students' names to their letter grades when passed the above data:

```
return value:
  {Mort=C+, Dan=B-, Alyssa=A+, Kim=D, Lisa=B+, Bob=F, Jeff=C-, Sylvia=A, Vikram=A-}
```

Though maps often store their elements in unpredictable order, for this problem you may assume that the grade mapping's key/value pairs are stored in ascending order by keys (percentages).

If either map passed to your method is empty, your method should return an empty map. You should not make any assumptions about the number of key/value pairs in the map or the range of possible percentages that could be in the map.

Constraints: You may create one new **map** as storage to solve this problem. (This is the map you will return.) You may not declare any other data structures. You can have as many simple variables as you like, such as integers or strings. Do not modify the maps that are passed in to your method as a parameter.

Problem 6: Let's Go For A Drive (35 points)

This is a two-part question

Part 1: Write a class called `Car` that keeps track of its mileage (number of miles it's driven), the number of gallons of gas left in its tank, and whether or not it is broken down. You should be able to create a `Car` by specifying the initial amount of gas in the tank (decimal), and the initial mileage (decimal), like so:

```
Car myCar = new Car(10.0, 1000.0);   // 10.0 gallons, 1000.0 miles driven
```

The main functionality of a `Car` is the `turnOnAndDrive` method that takes as a parameter the number of miles the car should drive. This method should do the following:

- Every time a car turns on to drive, there is a chance it breaks down. Specifically, the chance of a breakdown is 0.1 (10%) if you've driven 0 to 10,000 miles, 0.2 (20%) if you've driven 10,001 to 20,000 miles, etc. If the car breaks down, **it can no longer drive**.
- If the car turns on successfully, it should drive the number of miles passed in as a parameter. Its mileage and gas left should be updated accordingly to indicate the car has driven additional miles, and has used up some gas (there is a provided `MILES_PER_GALLON` constant that indicates how many miles the car can go on one gallon of gas. You do not have to define this constant in your answer).
- If the car can drive, but does not have enough gas to drive the full specified distance, it should drive **until it runs out of gas**.
- The method should return `true` if the car was able to drive the specified distance, and `false` otherwise (if it breaks down or does not have enough gas).

For example, using our `myCar` from earlier, assuming our car does not break down, and assuming `MILES_PER_GALLON = 25`, `myCar.turnOnAndDrive(25)` would return `true` and our car would now have 9 gallons of gas and 1025 miles driven.

Then, if we call `myCar.turnOnAndDrive(250)` it would return `false` because our car does not have enough gas to drive this distance, and our car would now have 0 gallons of gas and 1,250 miles driven (because 9 gallons of gas gets us 225 more miles).

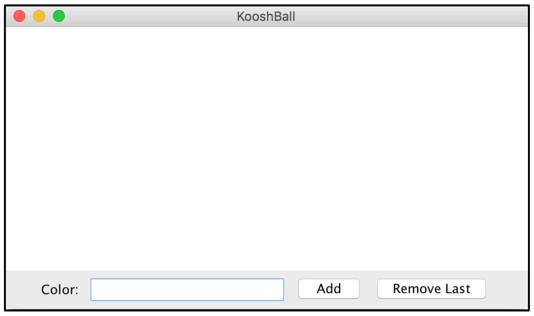
In total, your class must implement the following public methods:

```
/** Attempts to drive # miles, returns true for success or false for fail */
   public boolean turnOnAndDrive(int milestToDrive);

/** Returns the number of miles this car has driven. */
   public int getMileage();

/** Returns true if the car is broken down, and false otherwise. */
   public boolean isBrokenDown();

/** Sets the car as no longer broken down */
   public void repair();

/** Adds the given number of gallons of gas to the car's gas tank */
   public void fillGas(double numberOfGallons);
```

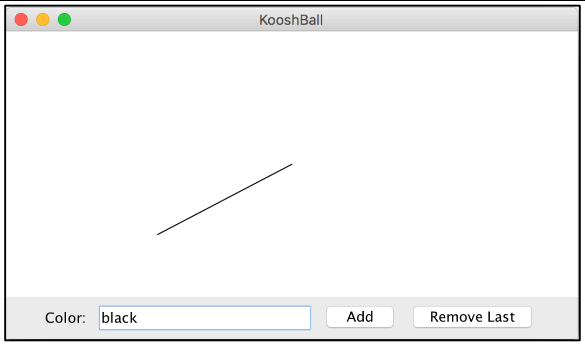
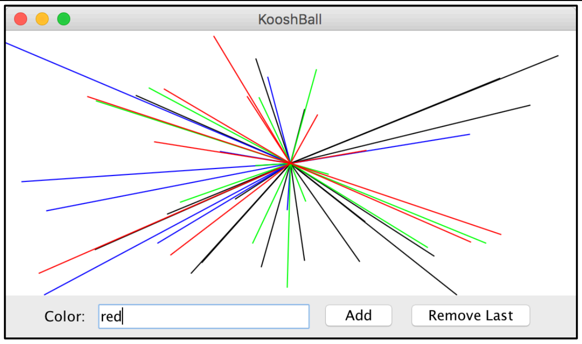
Part 2: You are employed by a car manufacturer who wants you to write software to help test how long their cars will go before they break down. Write a method called **testCar** that makes a new `Car` and returns the mileage of that car once it breaks down for the first time. The car should start out with 10 gallons of gas, and 0 miles driven, and you should drive the car in increments of **10 miles**. If the car runs out of gas, you should refill it to 10 gallons of gas. Once the car breaks down, you should return how many miles the car had driven.

Problem 7: KooshBall (25 points)

Write a complete program named **KooshBall** that implements a graphical user interface for drawing random lines of various colors. When you start the program, the user interface has a blank central canvas and a bottom region with the following graphical components: a label displaying the text "Color: ", a text field of width **16** for the user to input a color, an "Add" button, and a "Remove Last" button.



When the user clicks the "Add" button or presses ENTER in the text field, a new line is drawn on the central canvas. The line is drawn such that one of its endpoints is in the center of the canvas, and the other endpoint is chosen randomly to be any location in the canvas. The line should also be drawn in the **color entered in the text field (case insensitive)**. The screenshots below show the effect of clicking the Add button or hitting ENTER several times:

After clicking Add once (or hitting ENTER), with the color "black".	*After adding several times with various colors.*
	

The "Remove Last" button **removes the most recent line drawn**. However, it only does this **once** at a time. In other words, if you click "Remove Last" again, the button should not remove any more lines until the user has added another line to the canvas.

To help you implement this problem, you may assume that you have a constant called `colorMap` that is a map from `String` color names (such as `"black"`, `"blue"`, etc.) to their `Color` variable equivalent (e.g. `Color.BLACK`, `Color.BLUE`, etc.). In other words, `colorMap` is defined as a `HashMap<String, Color>`. You do not need to worry about declaring or creating this map in your program; just assume it is provided to you as a constant. If the user enters the name of a color that is not in the provided map, you should **not draw anything on the canvas**.