Nick Troccoli
CS 106A

Practice Final
August 11, 2017

Practice Final Examination

Final Exam Time: Friday, August 18th, 12:15P.M.–3:15P.M.
Final Exam Location: To Be Announced

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)

(1a) What output is printed by the following program?

```
public class Mystery1 extends ConsoleProgram {
   public void run() {
         int y = 1;
         int x = 3;
         int[] a = new int[4];

         mystery(a, y, x);
         println(x + " " + y + " " + Arrays.toString(a));

         x = y - 1;
         mystery(a, y, x);
         println(x + " " + y + " " + Arrays.toString(a));
   }
   private void mystery(int[] a, int x, int y) {
      if (x < y) {
         x++;
         a[x] = 17;
      } else {
         a[y] = 17;
      }
      println(x + " " + y + " " + Arrays.toString(a));
   }
}
```

(1b) 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) {
   HashMap<String, String> result = new HashMap<>();
   for (String k : map.keySet()) {
         String v = map.get(k);
         if (k.charAt(0) <= v.charAt(0)) {
             result.put(k, v);
         } else {
             result.put(v, k);
         }
   }
     println(result);
}
```

a) {two=deux, five=cinq, one=un, three=trois, four=quatre}

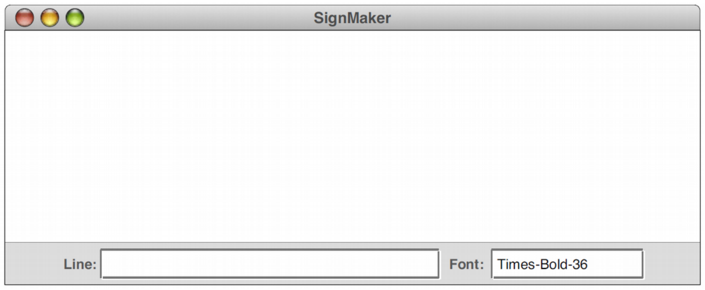
b) {skate=board, drive=car, play=computer, program=computer}

c) {siskel=ebert, girl=boy, heads=tails, ready=begin, first=last, begin=end}

d) {cotton=rain, tree=violin, seed=tree, light=tree, shirt=cotton}

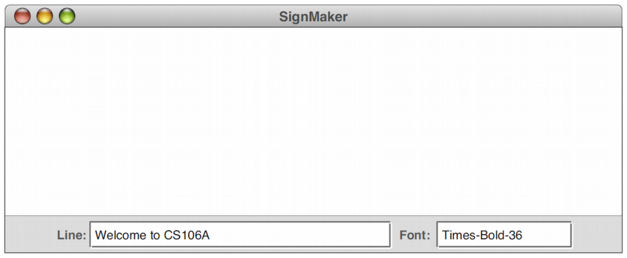
Problem 2: SignMaker (25 points)

Write a complete program named **SignMaker** that implements a graphical user interface for creating simple signs, each of which consists of lines of centered text displayed in different fonts. When you start the program, the user interface looks like the screenshot below. In its bottom region it contains a 30-character-wide text field labeled "Line:" and a 15-character-wide text field labeled "Font:". The initial text of the "Line" field is blank, and the initial text of the "Font" Field is "Times-Bold-36".

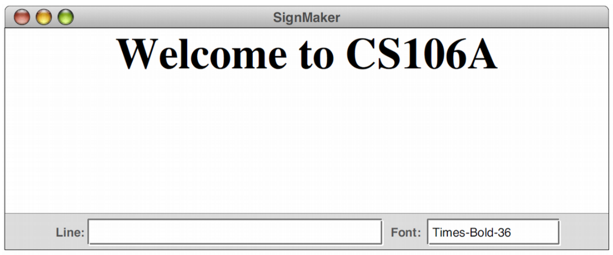


Times-Bold-36

You can then add a line to the display by entering text into the Line field and pressing Enter in the line text field only, as shown in the screenshots below. That label should be centered in the window and set in the font specified in the 15-character-wide Font text field. The first label you add should be positioned so that its text is at the very top of the window. Pressing ENTER also clears the text field in the control strip making it easier for the user to enter the next line of the sign.

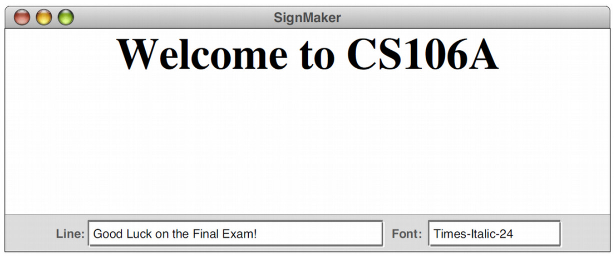


Welcome to CS106A
Times-Bold-36

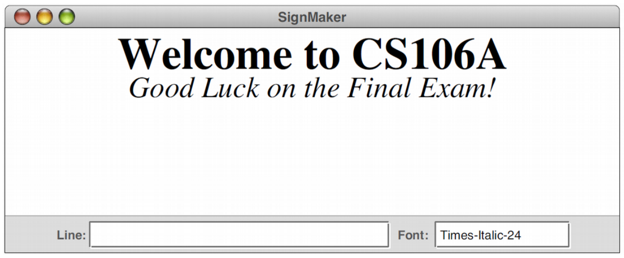


Welcome to CS106A
Times-Bold-36

The user can change the font of each subsequently added line by typing a new font name in the Font field. For example, if the user wanted to add a second line to the message is a smaller, italic font, that user could do so by changing the contents of the Font field to Times-Italic-24 and then typing in a new message. Typing Enter at this point would add a new centered label below the first one. The distance to the next baseline from the previous one should be the height of the new label you're adding. The screenshots below show the window state before and after adding a second label. You may assume that the user types valid font strings into the Font field.



Welcome to CS106A
Good Luck on the Final Exam!
Times-Italic-24



Welcome to CS106A
Good Luck on the Final Exam!
Times-Italic-24

Hint: the .setFont method on GLabels takes a string of the format “Times-Italic-24”.

Problem 3: Never-Ending Birthday Party (25 points)

Suppose you want to hold a never-ending birthday party, where every day of the year someone at the party has a birthday. How many people do you need to get together to have such a party?

Your task in this program is to write a program that simulates building a group of people one person at a time. Each person is presumed to have a birthday that is randomly chosen from all possible birthdays. Once it becomes the case that each day of the year, someone in your group has a birthday, your program should print out how many people are in the group, then should exit.

In writing your solution, you should assume the following:

- There are 366 possible birthdays (this includes February 29).
- All birthdays are equally likely, including February 29.

You might find it useful to represent birthdays as integers between 0 and 365, inclusive.

Problem 4: Magic Squares (35 points)

A *magic square* is an $n \times n$ grid of numbers with the following properties:

1. Each of the numbers $1, 2, 3, \ldots, n^2$ appears exactly once, and
2. The sum of each row and column is the same.

For example, here is a 3×3 magic square, which uses the numbers between 1 and $3^2 = 9$:

4	9	2
3	5	7
8	1	6

and here is a 5×5 magic square, which uses the numbers between 1 and $5^2 = 25$:

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

Write a method

```
private boolean isMagicSquare(int[][] square, int n);
```

that accepts as input a two-dimensional array of integers (which you can assume is of size $n \times n$) and returns whether or not it is a magic square.

Problem 5: Favorite Letters (25 points)

Write a complete program named **FavoriteLetters** that extends Program (it doesn't have a canvas) and implements a graphical user interface for maintaining a list of the user's favorite letters of the alphabet. The user can type one-letter strings into a text field and add/remove them from a list.

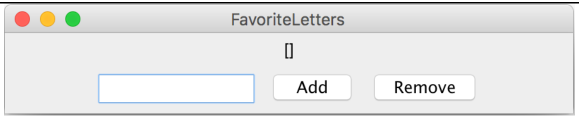
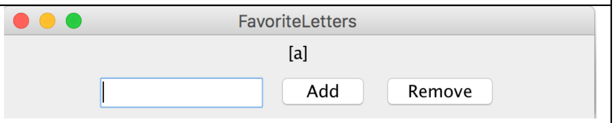
Components: The top region has a text label that initially contains the text "[]". The bottom region has a text field for typing a phrase (10 characters wide, initially empty text), a button labeled "Add", and a button labeled "Remove".

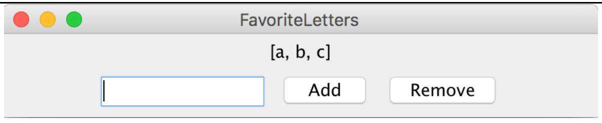
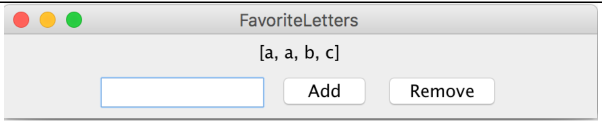
Event - Add: When the user presses Enter in the text field or clicks the **Add** button, your program should grab the text typed by the user, and if it is a single-letter string, you should add it to the **end** of a list of letters that is displayed in the central label. However, if the list *already contains* the entered letter, then this is definitely one of the user's favorites, and you should put it at the **front** instead.

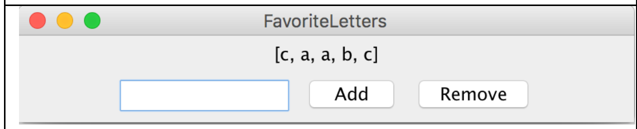
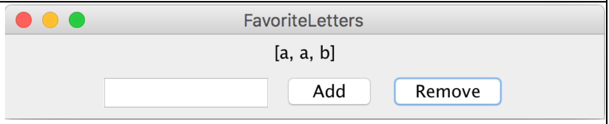
If the text field's text is not a single letter, your code should do nothing. You may assume that the user will type only letters from A-Z and not other characters. Regardless of the **capitalization** in which the user types the letter, you should add it to the list in **lowercase**.

Event - Remove: When the user clicks the **Remove** button, you should remove all occurrences of the current letter typed into the text box from the favorite letters list. For example, if the list is [a, b, b, a, c, a] and the user types "a" and clicks Remove, the list should become [b, b, c]. If the typed string is not found in the list, your code should not change the list and should not crash. As with Add, the **Remove** functionality should be **case-insensitive**; that is, the user should be able to type letters with any capitalization and have them be matched and removed successfully.

The screenshots below show the effect of typing various text and clicking the various buttons:

Initial appearance	*After typing "a" and clicking "Add":*
 FavoriteLetters [] Add Remove	 FavoriteLetters [a] Add Remove

Type "B", click "Add"; type "C", hit ENTER	*Add "a" a second time:*
 FavoriteLetters [a, b, c] Add Remove	 FavoriteLetters [a, a, b, c] Add Remove

Add "c" a second time:	*After typing "c" and clicking "Remove":*
 FavoriteLetters [c, a, a, b, c] Add Remove	 FavoriteLetters [a, a, b] Add Remove

Problem 6: SubMaps (25 points)

Write a method named **isSubMap** that accepts two hash maps from strings to strings as its parameters and returns true if every key in the first map is also contained in the second map and maps to the same value in the second map. For example,

```
{Smith=949–0504, Marty=206–9024}
```

is a sub-map of

```
{Marty=206–9024, Hawking=123–4567, Smith=949–0504, Newton=123–
4567}.
```

Therefore if the first map is `map1` and the second is `map2`, here `isSubMap(map1, map2)` would return `true`.

The empty map is considered to be a sub-map of every map.

Constraints: You may not declare any auxiliary data structures in solving this problem.

Problem 7: StringQueue (25 points)

A useful type of data structure in the Java collections framework that we haven't worked with yet is the `Queue`, which models a collection in which objects are added at one end and removed from the other, much as in a waiting line. Queues are extremely useful for modeling things such as waiting lines (in hospitals, stores, transportation, etc.) and task priorities.

We can consider a queue of `String`s, where the fundamental operations are `add`, which adds a `String` to the end of the queue, `poll`, which removes and returns the `String` at the front of the queue (or `null` if the queue is empty), `randomAdd`, which adds a `String` to a random position within the queue, and `size`, which returns the number of `String`s in the queue. Your task in this problem is to write an implementation of the class `StringQueue`.

As an illustration of how `StringQueue` works, you could create an empty queue like so:

```
StringQueue queue = new StringQueue();
```

You could then add the three ghosts from Dickens's *A Christmas Carol* like this:

```
queue.add("Christmas Past");
queue.add("Christmas Present");
queue.add("Christmas Future");
```

At this point, calling `queue.size()` should return 3 because there are three entries in the queue. The first call to `queue.poll()` would return `"Christmas Past"`, the second would return `"Christmas Present"`, and the third would return `"Christmas Future"`. If you called `queue.poll()` a fourth time, the return value would be `null`.

We could also add a string randomly like so:

```
queue.addRandom("Christmas Super Future");
```

This would add the string with equal chance to a random index in the queue.

Figure 1: Required methods in the StringQueue Class

```
/** Adds a new String to the end of the queue */
   public void add(String str);

/** Adds a new String to a random position in the queue */
   public void addRandom(String str);

/** Removes and returns the first String (or null if queue is empty) */
   public String poll();

/** Returns the number of entries in the queue. */
   public int size();
```