CS 106A Practice Midterm Exam

Midterm Time: Monday, July 23rd, 7:00PM–9:00PM
Midterm Location: Hewlett 200

Based on handouts by Nick Troccoli

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the midterm 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).

This practice exam is intentionally *more difficult* than the real exam will be – think of this as an upper ceiling of difficulty for the midterm exam.

The midterm exam is on your computer but closed-textbook and closed-notes. A "syntax reference sheet" will be provided during the exam (it is omitted here, but available on the course website). It will cover all material presented up to the midterm date itself. Please see the course website for a complete list of midterm 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. Type all of your answers directly on the *answer page provided for that specific problem*, including any work that you wish to be considered for partial credit.

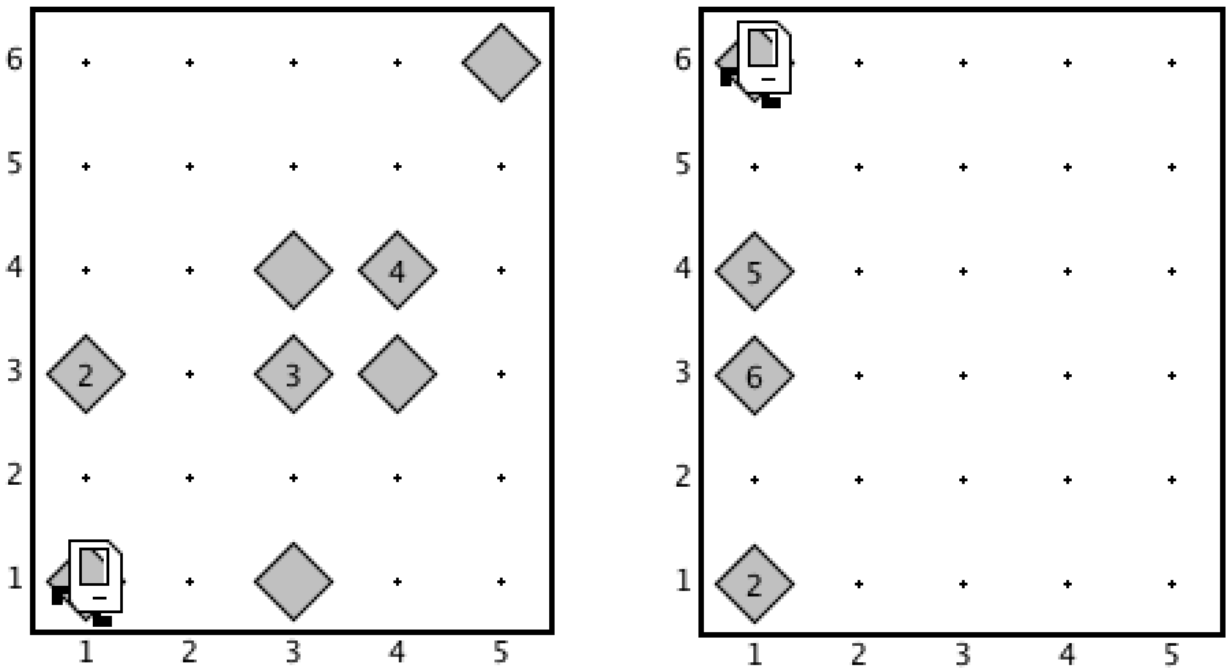
Each question is marked with the number of points assigned to that problem. The total number of points is 120. We intend for the number of points to be roughly comparable to the number of minutes you should spend on that problem.

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.

Problem 1: Karel the Farmer [20 points]

Karel has recently started a new job as a farmer, and needs you to write a program called **`FarmerKarel`** to help it gather up crops (represented as beepers, of course). Karel starts off in a world of any size, with crops (beepers) scattered around this world. Your program should have Karel, for each row in this world, gather up the beepers in that row and place them on the *leftmost square of that row*. Here is a before-and-after example:

before: *after:*



Note that, in each row, Karel has gathered all beepers in that row and placed them on the leftmost square. If there are no beepers in a row, Karel should not place any beepers in that row.

You may assume the following facts about the world:

- Karel starts at (1, 1) facing East, with **an infinite number of beepers in its beeper bag**. This means Karel cannot put down exactly the number of beepers it previously collected.
- The world may be *any* size, and your program should work for all world sizes.
- There may be any number of beepers, or no beepers, on a given square, and beepers may be placed anywhere in the world.
- There are no walls in the world.
- Karel's ending location and direction do not matter.

Note that you are limited to the Java instructions shown in the Karel reader. For example, the only variables allowed are loop control variables used within the control section of the **`for`** loop. You are *not* allowed to use syntax like local variables, instance variables, parameters, return values, Strings, **`return`** or **`break`**, etc.

Problem 2: Java Statements and Expressions [20 points]

(a) For each expression in the left-hand column, indicate its value. Be sure to list a constant of the appropriate type (e.g. 7.0 rather than 7 for a `double`, `String`s in double quotes, `char`s in single quotes, true/false for a `boolean`, etc.).

i. `1 + (2 + "B") + 'A'`

ii. `11 / 2 > 5 || 5 % 2 == 1`

iii. `(char)('B' + 2) + "" + 4 + 27 / 3`

iv. `21 / 2.0 + 3 % 4 – 23 / 2`

v. `!(3 / 2 < 1.5) && (4 > 5 || 2 % 3 == 0)`

(b) What are the color, dimensions and location of **`rect`** on the canvas?

```
public class Problem2b extends GraphicsProgram {
     public void run() {
          int num2 = 13;
          int num1 = 9;
          int width = mystery1(num2, num1);
          int height = mystery1(5, num2*3);
          GRect rect = new GRect(0, 0, width*3, height);
          rect.setFilled(true);
          rect.setColor(Color.BLUE);
          mystery2(rect, num1);
          add(rect);
     }

     private int mystery1(int num1, int num2) {
          num2 += 3;
          String str = "Hello " + num1;
          int num3 = num2 - str.length();
          return num3;
     }

     private void mystery2(GRect otherRect, int num2) {
          otherRect.setLocation(num2, num2);
          otherRect.setColor(Color.RED);
     }
}
```

Problem 3: Movie Kiosk [25 points]

A local movie theater has hired you to write a **`ConsoleProgram`** for their ticket purchase kiosks called **`MovieKiosk`** that continually prompts the user for movie name, # tickets and price per ticket, and when they are done outputs the names of all movies they want to see, as well as the total cost (see input at right).

```
Movie name: Cars
# tickets: 2
Ticket price: 14.50

Movie name: WALL-E
# tickets: 4
Ticket price: 14.00

Movie name: Up
# tickets: 3
Ticket price: 10.50

Movie name: [ENTER]

Movies: Cars and WALL-E and Up
Total: $116.5
```

The program should stop prompting the user once they submit **ENTER** for the movie name. You can assume that if they enter a movie name, they will always enter a valid ticket quantity and price. At the end, you should print out all entered movie names, as well as the total price of all purchased tickets. You do not have to worry about the number of digits displayed for any numbers. If no movies are entered, you should print out `"Movies: None"` and **not print out the total price**.

For movie names, print `"Movies: "` followed by all movie names joined with `" and "`. For instance, if the user enters `"Cars"`, `"WALL-E"` and `"Up"`, you should print out `"Movies: Cars and WALL-E and Up"`. If they enter `"Cars"` you should print `"Movies: Cars"`.

The movie theater would also like you to award randomly-chosen customers *vouchers* (credits) towards their next movie to encourage future purchases. Specifically, every time the user selects tickets for a movie (after entering the price per ticket for a movie), there is a **10% chance this user gets a voucher towards their next entered movie (**if there is one). The voucher should be for a random integer dollar amount **between $5-25**.

```
Movie name: Cars
# tickets: 2
Ticket price: 14.50
You have won a $10 voucher for
your next purchase!

Movie name: WALL-E
# tickets: 2
Ticket price: 3.50

Movie name: Up
# tickets: 2
Ticket price: 10

Movie name: [ENTER]

Movies: Cars and WALL-E and Up
Total: $49.0
```

On their next purchase, if the voucher amount is *less than* the total for that movie, simply deduct the voucher amount from that total. If the voucher amount is *greater than or equal* to the total for that movie, then they get those tickets for free, and the remainder **disappears**. For instance, at right, the user receives a $10 voucher for *WALL-E* after purchasing *Cars* tickets; they only spend $7 on *WALL-E* tickets, so they get those tickets for free and the remaining $3 disappears; it does **not** carry over to their *Up* ticket purchase. If, in this example, the user instead spent $14 on *WALL-E* tickets, then after the voucher is applied only $4 would be added to their total.

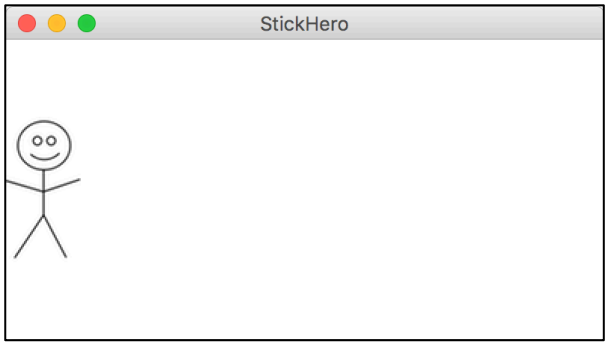
Hint: the RandomGenerator method ***nextBoolean*** *takes a double parameter which is the percent chance of returning true.*

Problem 4: StickHero [20 points]

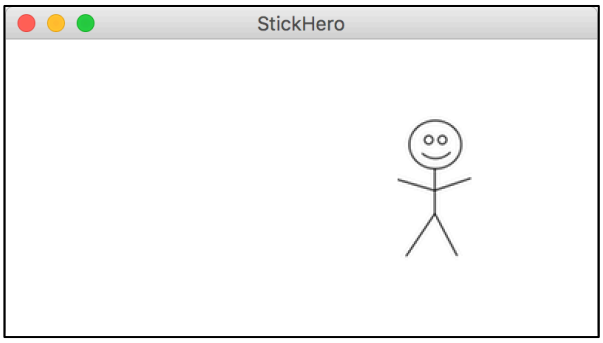
Your friends are working on a new video game and, knowing that you are in CS 106A, have enlisted your programming help for the main character, StickHero, who has the superpower to switch between "double size" and "original size". If the user clicks on StickHero when it is original size, it will go to double size, and vice versa. The player image, `player.png`, has been provided by your teammates in the `res` folder; original size is defined to be the size of this image, and double size is defined to be double the dimensions of this image.

Write a GraphicsProgram called **`StickHero`** that starts with original-size StickHero at x = 0, centered vertically. When the program starts, StickHero should start animating to the right **5 pixels each iteration**, with a delay of **30ms**.

initial state

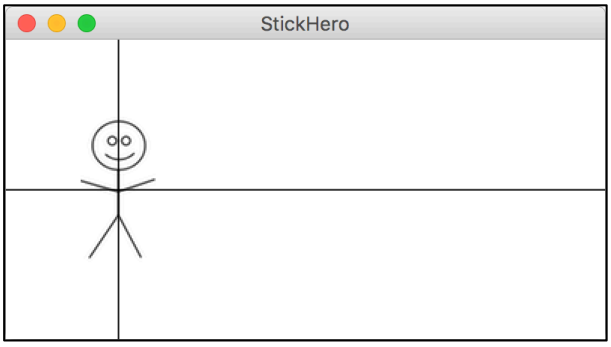


after a few seconds

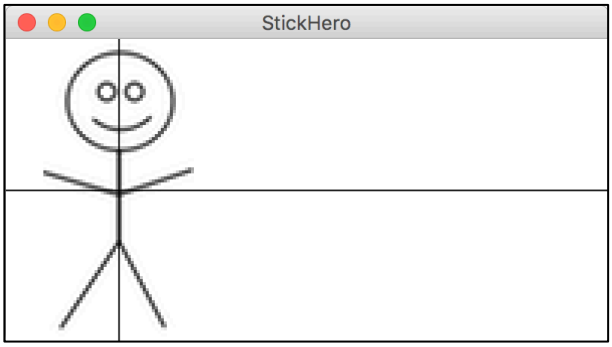


As soon as any part of StickHero goes off the right edge of the screen, it should go back to its starting position at the left side of the screen and continue animating to the right. If, at any point, the user clicks on StickHero, it should toggle between double size and original size. Notably, however, **the x/y center point of StickHero should never change**.

original size



double size



As shown above, if you mark StickHero's center (note: these lines do not actually appear onscreen), it remains unchanged as you toggle between original and double size.

Your program should work for any screen size and any reasonable size of `player.png` (e.g. that fits onscreen). You do not need to account for the screen resizing after the program launches.

Problem 5: Mentions [35 points]

After finishing up their StickHero video game, your friends have now moved on to working on a new social network. They need you to help them implement **mentions**, as described below. *Note that this is a two-part problem where you will implement a single method and a ConsoleProgram.*

(a) Write a method **`replaceMention`** that accepts a string as a parameter and, if it is a *mention*, returns a new string that is the mention's *expanded name*. A *mention* is defined as an '@' symbol followed by 1 or more upper-camel-cased names; upper-camel-case means the first letter is uppercase and all other letters are lowercase. For instance, **`@NickTroccoli`**, **`@Nolan`** and **`@AleksanderPaulDash`** are all valid mentions, But **`@`** and **`@nicktroccoli`** are not valid mentions.

The *expanded name* of a mention is all the upper-camel-case names in the mention with a space in between them, except for the *last* name in the mention, which should be abbreviated with only its first initial and a period. However, if the mention contains only 1 name, the expanded name is just that name (without the '@'). The following table shows some sample inputs and outputs to the **`replaceMention`** method.

Call	Value Returned
`replaceMention("@NickTroccoli")`	`"Nick T."`
`replaceMention("@AleksanderPaulDash")`	`"Aleksander Paul D."`
`replaceMention("@Nolan")`	`"Nolan"`
`replaceMention("hello!")`	`"hello!"`

The input is guaranteed to be a single token without any spaces, and is guaranteed to be entirely a mention, or not contain a mention at all. Note that if the input is *not* a mention (including the empty string), the output is simply the same as the input.

(b) Write a ConsoleProgram called **`ReplaceMentions`** that prompts the user for a valid text filename in the `res` folder, and prints out that file to the console with all of the mentions replaced with their expanded names. You should reprompt the user until they enter a valid filename. If an error occurs reading the file, print out an error message. You may assume that the text file specified has only 1 line of text, and that each word in the file is separated by a single space. For example, if the file `myinput.txt` contains the following text:

```
Head TA @RishiPaulBedi - friends with @Nick - rocks socks!
```

Then the output of the **`ReplaceMentions`** program would look like the following (user input bolded and underlined):

```
Filename: myinput.txt
Head TA Rishi Paul B. - friends with Nick - rocks socks!
```