Eric Roberts and Jerry Cain
CS 106J

Handout #52
June 7, 2017

Practice Final Examination #2

Review session: **Sunday, June 11, 6:00–8:00 P.M. (Gates B-12)**
Scheduled final: **Wednesday, June 14, 8:30–11:30 A.M. (Lathrop 282)**

Problem 1—Short answer (10 points)

1a) Suppose that the function `conundrum` is defined as follows:

```
function conundrum() {
   var array = [ ];
   for (var i = 0; i <= 12; i++) {
      array.push(0);
      for (var j = i; j > 0; j--) {
         array[j] += j;
      }
   }
   return array;
}
```

If you look at the code, you'll see that the function pushes a new value onto the array for every value between 0 and 12, which means that the array will eventually contain 13 elements with indices ranging from 0 to 12. Work through the function carefully and indicate the value of each of the elements in the array that **conundrum** returns:

0	1	2	3	4	5	6	7	8	9	10	11	12

1b) What value is printed if you call the function **confusion** in the following code:

```
function confusion() {
   var obj = exasperating(6);
   console.log(obj.fn(10));
}

function exasperating(x) {
   return {
      fn: function(x) { return "" + x + this.x; },
      x: x * 11
   };
}
```

Problem 2—Simple graphics (15 points)

If you want to build interactive programs, it is useful to have a library of interactive elements such as buttons. When you are working with JavaScript on the web, you can use the `<button>` HTML element, as Jerry showed you in class. When you are working in SJS, you have to simulate buttons using `GCompound` objects, just as you did in the Enigma assignment.

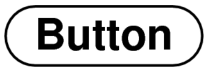
Your task in this problem is to write a function

```
function createButton(label)
```

that creates a `GCompound` object suitable for use as a button. For example, if you run the program

```
function TestButton() {
   var gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   var button = createButton("Button");
   var x = (GWINDOW_WIDTH - button.getWidth()) / 2;
   var y = (GWINDOW_HEIGHT - button.getHeight()) / 2;
   gw.add(button, x, y);
}
```

it should display a button in the center of the graphics window that looks like this:



The `GCompound` that represents the button is composed of two semicircular `GArc`s that form the ends of the button, two `GLine`s that form the top and bottom edges of the button, and a `GLabel` that shows the button label. The appearance of the button is controlled by the following constants:

```
const BUTTON_HEIGHT = 22;
const BUTTON_FONT = "SansSerif-Bold-16";
const BUTTON_LABEL_DY = 6;
```

The constant `BUTTON_HEIGHT` specifies the vertical extent of the button and also gives the diameter of the semicircles. The width of the lines on the top and bottom are determined by the size of the `GLabel` when rendered in `BUTTON_FONT`. Thus, if you create a button by calling `createButton("A Long Button Name")`, the button would be longer in the horizontal dimension so that the label fits inside the center of the button, with the semicircles adding a bit of margin around the label, like this:

A Long Button Name

The constant `BUTTON_LABEL_DY` specifies the distance between the center line of the button and baseline for the `GLabel` (using `getAscent` to implement the vertical centering produces a button that looks terrible, so this constant is used to position the label at just the right place for this point size).

As you write your solution to this problem, you should keep the following points in mind:

- As the sample program indicates, the origin for the **`GCompound`** is the upper left corner of the rectangle that encloses the button. Although it might simplify some programs to have the origin be in the center, it probably makes more sense to define the geometry for buttons so that it is consistent with the other graphical objects.
- For the purposes of this problem, you don't have to worry about making the button active. All you have to do is create the **`GCompound`** that displays it. The easiest way to turn the button into an active element is to assign a **`clickAction`** method to the **`GCompound`**, just as you did for the Enigma assignment, but you don't need to worry about that for this problem.

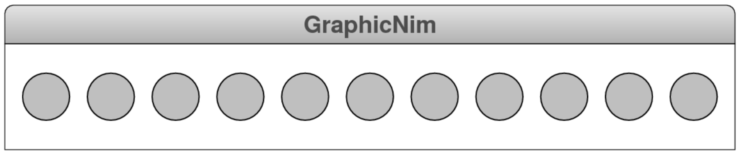
Problem 3—Interactive graphics (20 points)

Back before we had a graphics library, one of the early assignments in CS 106A was to write a program that played a simple game called Nim. In the simplest version of the game, two players start with a pile of 11 coins on the table between them. The players then take turns removing 1, 2, or 3 coins from the pile. The player who is forced to take the last coin loses. In the old days, the point of this assignment was to figure out a strategy for winning the game. With our modern attachment to fancier user interfaces, however, it is just as easy to focus on the problem of representing the game graphically on the screen.

For this problem, your task is to create a program that implements the graphical representation of the Nim game, which is easiest to do in two steps:

Step 1.

Write the code necessary to create a graphical display in which a line of coins is arranged horizontally on the screen, like this:



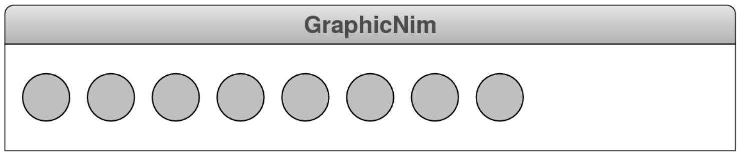
In creating this display, you should make use of the following constants:

```
const GWINDOW_WIDTH = 496;
const GWINDOW_HEIGHT = 75;
const N_COINS = 11;
const COIN_SIZE = 32;
const COIN_FILL_COLOR = "LightGray";
```

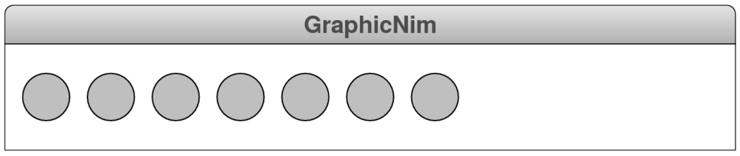
It is useful to note that these constants specify the number of coins and the coin size, but not the spacing. You should position the coins so that the space between each coin and between the coins and the border of the window is divided equally after determining how much space is left after displaying the coins. The line of coins should be centered both horizontally and vertically in the window. Each coin should be outlined in black and use the light-gray fill color specified by the constant **`COIN_FILL_COLOR`**.

Step 2.

The second part of the problem consists of making it possible to take coins away. Add the necessary code so that if the user clicks the mouse in one of the last three coins in the row, those coins disappear. For example, if the user 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 user then clicked on 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 simply be ignored.

In order to solve this problem, you will need to store the **`GOval`** objects in an array so that you can keep track of their order. When a mouse click occurs, your program needs to find the object at that mouse location (if any) and see whether it is one of the last three elements in the array. If so, your program should remove that element and any following elements from both the list and the window.

Problem 4—Strings (15 points)

The table of contents for a book typically consists of a list of chapter titles along the left margin of a page and the corresponding page numbers along the right. To make it easier for your eye to match up the chapter and page, the usual approach is to tie the two visually with a line of dots called a ***leader.*** Using this style, the table of contents for the draft JavaScript textbook would look like this:

```
1. A Gentle Introduction  . . . . . . . . . . . . . . . . .   1
2. Introducing JavaScript . . . . . . . . . . . . . . . . . .  39
3. Control Statements . . . . . . . . . . . . . . . . . . . .  77
4. Functions  . . . . . . . . . . . . . . . . . . . . . . .  113
5. Writing Interactive Programs . . . . . . . . . . . . . .  151
6. Strings  . . . . . . . . . . . . . . . . . . . . . . . .  191
7. Arrays . . . . . . . . . . . . . . . . . . . . . . . . .  231
8. Objects  . . . . . . . . . . . . . . . . . . . . . . . .  265
```

Write a function

```
function createTocEntry(title, page)
```

that takes a chapter title (which includes the chapter number in these examples) and the page number on which that chapter begins. Your function should returns a string formatted as an entry for the table of contents. Thus, if you were to call

```
createTocEntry("6. Strings", 191)
```

the function should return the following string:

```
"6. Strings  . . . . . . . . . . . . . . . . . . . . . . .  191"
```

In generating this string, your function should adhere to the following guidelines:

- The strings returned by **`createTocEntry`** should all have the same length, which is given by the constant **`TOC_LINE_LENGTH`**. In the earlier examples, **`TOC_LINE_LENGTH`** has the value 60.
- The chapter title must appear at the beginning of the result string and must be separated from the first dot in the leader by at least one space.
- The page number must appear at the end of the result string so that the last character of each page number will line up at the column specified by **`TOC_LINE_LENGTH`**. Like the title, the page number must be separated from the last dot in the leader by at least one space.
- The leader itself is composed of alternating spaces and dots, indicated by the period character `'.'`. Moreover, the dots must be arranged so that they line up vertically. If you simply start the leader one space after the chapter title, the dots would appear to weave back and forth on the page as illustrated by the following pair of lines:

  ```
  1. A Gentle Introduction . . . . . . . . . . . . . . . . . 1
  2. Introducing JavaScript . . . . . . . . . . . . . . . . 39
  ```

  An easy way to ensure that the dots are aligned correctly is to add an extra space after chapter titles—like `"1. A Gentle Introduction"`—with an even number of characters but not after those—like `"2. Introducing JavaScript"`—with an odd number.
- You may assume that the chapter title and page number fit in **`TOC_LINE_LENGTH`** character positions and need not make your function handle the situation when the title is too long for the line.

Problem 5—Arrays (10 points)

And the first one now
Will later be last
For the times, they are a-changin'.
—Bob Dylan

Write a function **`rotateArray(array, k)`** that takes an array and has the effect of shifting every element **`k`** positions toward the beginning of end of the array, and replacing them at the end of the array with the elements that were shifted out of the beginning of the array in their original order. For example, if the array **`digits`** has the contents

`digits`

0	1	2	3	4	5	6	7	8	9

calling

```
rotateArray(digits, 1);
```

should shift each of the values one position to the left and move the first value to the end:

`digits`

1	2	3	4	5	6	7	8	9	0

Similarly, if the array **`languages`** is

`languages`

`Pascal`	**`C`**	**`Java`**	**`JavaScript`**

calling

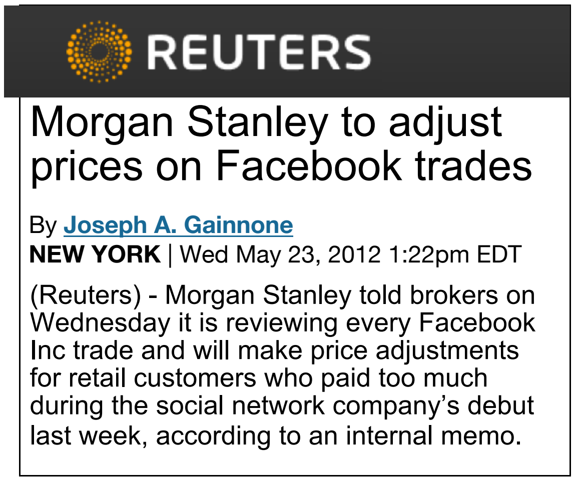
```
rotateArray(languages, 3);
```

should change the values in the array to

`languages`

`JavaScript`	**`Pascal`**	**`C`**	**`Java`**

You may assume that **`k`** is not negative and that it is not greater than the length of the array.

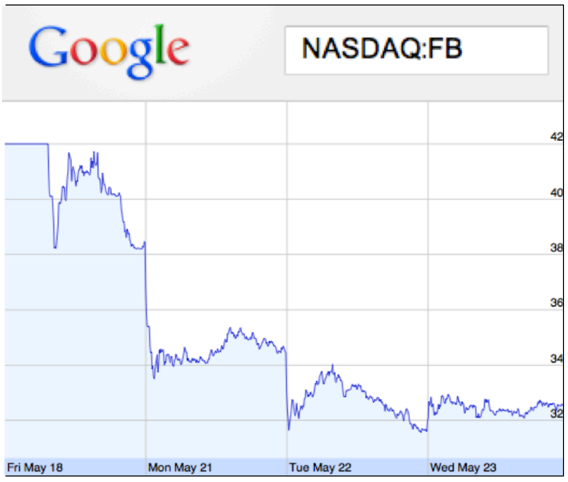


REUTERS

Morgan Stanley to adjust prices on Facebook trades

By Joseph A. Gainnone
NEW YORK | Wed May 23, 2012 1:22pm EDT

(Reuters) - Morgan Stanley told brokers on Wednesday it is reviewing every Facebook Inc trade and will make price adjustments for retail customers who paid too much during the social network company's debut last week, according to an internal memo.



Problem 6—Working with data structures (15 points)

Although it is hard to imagine now, Facebook's IPO in 2012 didn't go as well as predicted, and the Morgan Stanley brokerage that handled the offering was forced to make restitution to some clients, primarily for late trades. Suppose, for example, that a client ordered a sale at 11:28am on May 18, when Facebook was selling at $40.00 a share. Given the many delays on that day, Morgan Stanley might not have been able to execute the sell order until 3:58pm, when Facebook shares had dropped to $38.07. That client therefore lost $1.93 per share, which adds up quickly if the trade involved a large block of shares.

Suppose that Morgan Stanley has hired you to write a simple application to calculate refunds due to its customers. You have access to a data structure that contains the complete history of the share price for Facebook in the early days of trading. The data structure is an array of aggregates, each of which has three fields: a `date` field containing the date as a string (as in `"5/18/2012"` for May 18, 2012), a `time` field indicating the time as a string (as in `"11:30pm"`), and a `price` field as a number. A few entries in that array look like this when represented in JSON form:

```
const FB_SHARE_PRICE_DATA = [
   { date:"5/18/2012", time:"11:30am", price:42.0000 },
   { date:"5/18/2012", time:"11:31am", price:42.0125 },
   { date:"5/18/2012", time:"11:32am", price:42.0250 },
   { date:"5/18/2012", time:"11:33am", price:42.0250 },
   { date:"5/18/2012", time:"11:34am", price:40.9474 },
   { date:"5/18/2012", time:"11:35am", price:40.8425 },
   { date:"5/18/2012", time:"11:36am", price:40.1500 },
   { date:"5/18/2012", time:"11:37am", price:40.0367 },
   { date:"5/18/2012", time:"11:38am", price:40.0000 },
         . . . more entries for May 18 . . .
   { date:"5/18/2012", time:"3:55pm", price:38.0685 },
   { date:"5/18/2012", time:"3:56pm", price:38.1050 },
   { date:"5/18/2012", time:"3:57pm", price:38.0997 },
   { date:"5/18/2012", time:"3:58pm", price:38.0700 },
   { date:"5/18/2012", time:"3:59pm", price:38.2599 },
   { date:"5/18/2012", time:"4:00pm", price:38.2699 },
];
```

Write a function

```
function facebookRefund(nShares, date, timeOrdered, timeExecuted)
```

that uses the data in **`FB_SHARE_PRICE_DATA`** to compute the refund due to a customer who tried to sell **`nShares`** of Facebook stock if the order was made at **`timeOrdered`** on the specified date but the order was not completed until **`timeExecuted`**.

As an example, suppose that a client tried to sell 1000 Facebook shares at 11:38am on May 18, but Morgan Stanley was unable to complete the transaction until 3:58pm. You can compute the necessary refund by calling

```
facebookRefund(1000, "5/18/2012", "11:36am", "3:58pm")
```

The implementation has to look through the **`FB_SHARE_PRICE_DATA`** array to find the price of Facebook stock at the two specified times. From the table on the preceding page, you can see that Facebook stock was selling for $40.00 per share at 11:36am but had dropped to $38.07 per share by 3:58pm. Morgan Stanley's delay therefore cost the client $1.93 per share. Since the client was selling 1000 shares, the total refund is $1,930.00, which is the value that **`facebookRefund`** should return. The value of **`facebookRefund`** should never be negative. If the client profited from the delay, **`facebookRefund`** should simply return 0.

Problem 7—Reading data structures from files (15 points)

Computing is changing the nature of scholarship in almost every discipline. In the humanities, for example, computers make it possible to collect and analyze historical material in ways that would be impossible to do by hand.

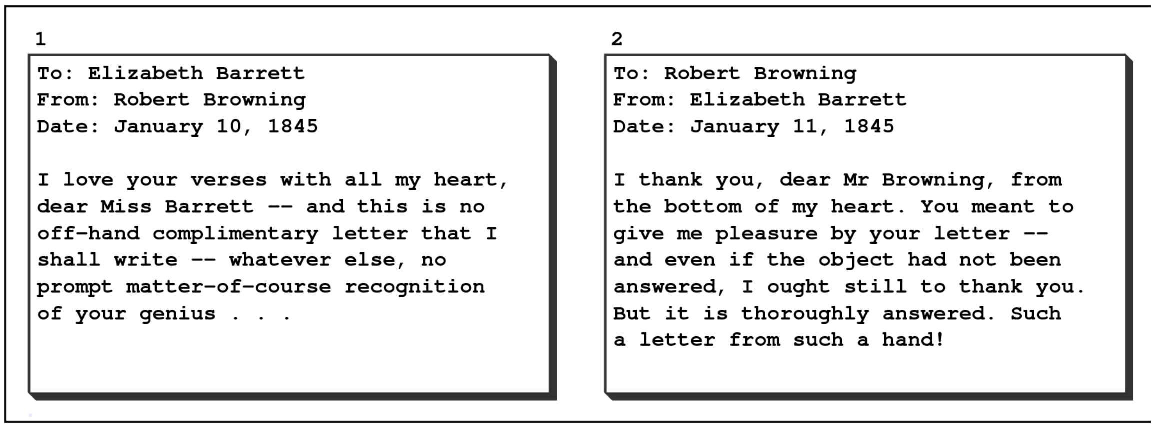
Suppose that you have been given the job of creating an internal data structure that represents correspondence, such as the 547 letters exchanged between the poets Robert Browning and Elizabeth Barrett prior to their marriage in 1846. Excerpts from the first two of those letters, annotated with header lines that resemble those used in modern email messages, appear in Figure 1. Assume that each of the letters has already been scanned and stored in a separate text file. The files all stored in a single folder and are given numeric file names numbered consecutively starting with 1. Thus, the letters shown in Figure 1 have the file names `1` and `2`, as shown.

Each message consists of any number of header lines, terminated by a blank line. Each header line contains a field name (`"To"`, `"From"`, and `"Date"` in these examples), followed by a colon and then any number of characters up to the end of the line. The lines following the blank line represent the body of the message.

Write a function `readLetters(dir)` that takes the name of a directory and reads all the messages in that directory into a single data structure. You may assume that there are no gaps in the sequence, so all you have to do is read the files `dir + "/" + 1`, `dir + "/" + 2`, and so on, until you encounter a file for which `File.readLines` returns `undefined`, which stops the process.

The `readLetters` function returns an array of aggregate structures, each of which represents a letter in the stream of correspondence. The value of that aggregate is a map containing the header fields and values, along with a special field named `body` that contains the body of the message as an array of lines. To ensure that the file numbers match the array indices, the array returned by `readLetters` should include an extra `null` value as element 0, since there is no message file numbered 0.

Figure 1. Letters between Robert Browning and Elizabeth Barrett



```
1
To: Elizabeth Barrett
From: Robert Browning
Date: January 10, 1845

I love your verses with all my heart,
dear Miss Barrett -- and this is no
off-hand complimentary letter that I
shall write -- whatever else, no
prompt matter-of-course recognition
of your genius . . .
```

```
2
To: Robert Browning
From: Elizabeth Barrett
Date: January 11, 1845

I thank you, dear Mr Browning, from
the bottom of my heart. You meant to
give me pleasure by your letter --
and even if the object had not been
answered, I ought still to thank you.
But it is thoroughly answered. Such
a letter from such a hand!
```

For example, if the directory **`Browning`** contains the files **`1`** and **`2`** as shown in Figure 1, the corresponding internal data would look like this in JSON form:

```
[
  null,
  { To: "Elizabeth Barrett",
    From: "Robert Browning",
    Date: "January 10, 1845",
    body: [
        "I love your verses with all my heart,",
        "dear Miss Barrett -- and this is no",
        "off-hand complimentary letter that I",
        "shall write -- whatever else, no",
        "prompt matter-of-course recognition",
        "of your genius . . ." ]
  },
  { To: "Robert Browning",
    From: "Elizabeth Barrett",
    Date: "January 11, 1845",
    body: [
        "I thank you, dear Mr Browning, from",
        "the bottom of my heart. You meant to",
        "give me pleasure by your letter --",
        "and even if the object had not been",
        "answered, I ought still to thank you.",
        "But it is thoroughly answered. Such",
        "a letter from such a hand!" ]
   }
]
```

In this problem, all you have to do is read the data into the internal structure. Any actual analyses of the letters are the responsibility of the clients of your **`readLetters`** function.