

ISE 208 Homework 4/Final Project

Due: Monday, December 14, at 11:59 PM

Submission Information

When you are finished with the assignment, send your completed Java source code (the .java file(s), **not** the compiled .class file(s)) as an e-mail attachment to the graduate TA (e-mail address: bhejakak AT gmail DOT com). **Be sure to put your name in your e-mail message AND in a comment at the top of your source code.** Include “ISE 208 HW 4” in the subject of your message.

Instructions

For this assignment, choose **one** of the options that follow. You *will not* receive any additional credit for completing both of them! *If you choose to submit solutions to both project options anyway, we reserve the right to select which one we will count for this assignment.* Both options are described in detail in the sections that follow, along with extra credit extensions. This assignment is worth approximately twice as many points (8) as each of the previous homework assignments.

Option 1: Complete the Java Fiendz ordering application as described

Option 2: The Monty Hall problem

Detailed Instructions

Option 1: Java Fiendz, Part 2

Complete the order-taking system that you began to develop for Homework 3 by adding the following classes and features:

1. Create a `Coffee` class to represent a single hot beverage. Every `Coffee` object contains the following instance fields:
 - a. A protected `double` variable named `basePrice`. This variable holds the cost of the beverage without accounting for any special options (cream, sugar, etc.).
 - b. A protected `ArrayList` variable named `options`. This variable should only store `CoffeeOption` objects that have been added to the given beverage.
 - c. A protected `String` variable named `size`. This variable represents the size of the beverage.
 - d. A protected `boolean` variable named `isDecaf`. This variable will be true if the beverage is decaffeinated, and false otherwise.
 - e. A public constructor that takes two arguments: a `String` followed by a `boolean` value. The constructor should create a new, empty `ArrayList` and

- assign it to `options`. The constructor should set the value of `size` to the `String` argument. The constructor should set the value of `basePrice` depending on the value of the `String` argument (“small” = 1.50, “medium” = 2.00, “large” = 2.50, “extra large” = 3.00). You may assume that the `String` argument will always be one of these four choices, with that spelling and capitalization. Finally, the constructor should set the value of `isDecaf` to that of its `boolean` parameter.
- f. A public method named `addOption()`. This method takes a `CoffeeOption` as its argument, and does not return anything. This method adds its argument to the end of the `options` `ArrayList`.
 - g. A public method named `price()`. This method returns a `double` value, and does not take any arguments. The `price()` method returns the sum of `basePrice` and the prices of all of the elements in `options` (use the `price()` method that you developed for Homework 3). Note that decaffeinated and regular beverages are the same price.
 - h. A public `toString()` method. This method returns a `String`, but does not take any arguments. Your `toString()` method should return a `String` that contains a description of the `Coffee` object, in the following format:

```
<size> <decaf-status> Coffee base-price  
<list of options>  
Total: $total-price
```

<decaf-status> should be either “Regular” or “Decaf”

For example, a small decaffeinated coffee with no added options would produce the following output:

```
small Decaf Coffee 1.50  
Total: $1.50
```

A medium regular coffee with one cream and one sugar would produce the following output:

```
medium Regular Coffee 2.00  
add cream 0.10  
add sugar 0.05  
Total: $2.15
```

(do not worry about formatting the price to exactly two decimal places; 1.5 and 1.499999999 are equally acceptable substitutes for 1.50)

Hint: use the `toString()` method(s) that you developed for Homework 3.

- i. You may add any additional instance variables or methods to this class that you wish.

2. Create a class named `Order`. This class maintains a list of `Coffee` objects, and contains the following fields:
 - a. A private `ArrayList` named `items` that holds `Coffee` objects.
 - b. A private `int` named `orderNumber`.
 - c. A public constructor that assigns a new, empty `ArrayList` to `items` and assigns a random integer value (between 1 and 2000) to `orderNumber` (see the end of this document for information about Java's `Random` class).
 - d. A public method named `add()` that takes a `Coffee` object as its argument and does not return any value. This method adds its argument to the end of the `items` `ArrayList`.
 - e. A public method named `getNumber()` that returns the order number. This method does not take any arguments.
 - f. A public method named `getTotal()`. This method returns a `double` value and does not take any arguments. This method returns the total price of the items in the current order, including 8.625% sales tax for Suffolk County.
 - g. A public `toString()` method that returns a `String` and does not take any arguments. This method should return a `String` that lists the order number, the current number of `Coffee` objects in `items`, and the total price for the order.

For example, `toString()` might return a `String` like the following:

Order #212 3 item(s) \$8.25

- h. A public method named `receipt()`. This method does not take any arguments. It returns a `String` containing a neatly-formatted order receipt that includes the following information:
 - i. The order number, with an appropriate label
 - ii. The total number of items in the order, with an appropriate label. Note that this value *only* includes whole beverages; do not include coffee options as separate items!
 - iii. A detailed list of the items in the order (HINT: use `Coffee`'s `toString()` method)
 - iv. The subtotal for the order, with an appropriate label
 - v. The tax amount, with an appropriate label
 - vi. The total price of the order, with an appropriate label.
 - i. You may add any additional instance variables or methods to this class that you wish.
 3. Create a subclass of `Coffee` named `IcedCoffee`. An `IcedCoffee` object is available in the same sizes as a regular `Coffee` object, except it costs \$0.50 more for the equivalent size (for example, a large `IcedCoffee` has a base price of \$3.00 instead of \$2.50). Be sure to provide the following functionality for your `IcedCoffee` class:

- a. A `public` constructor that, like the `Coffee` constructor, takes a `size` (a `String`) and a `regular/decaf` value (a `boolean`) as its arguments.
 - b. An overridden version of `toString()` that replaces the word “Coffee” with “Iced Coffee”. Otherwise, `toString()` provides exactly the same output as its superclass version.
4. Using the sample driver from Homework 3 as a model, create a menu-based program that allows the user to perform the following actions:
- a. Create a new `Order`. If there is an order currently in progress, it is discarded/replaced without warning.
 - b. Add a new `Coffee` object to the current order. This option should let the user do the following:
 - i. Specify a size for the beverage
 - ii. Specify whether the beverage is regular or decaffeinated
 - iii. Add coffee options (cream, sugar, and flavor shots) to the beverage until the user indicates that he/she is done. (HINT: use a `while` or `do-while` loop)

This option is only available if there is current order.

- c. Add a new `IcedCoffee` object to the order. This option should provide the same functionality as the preceding option. (HINT: instead of writing the same code twice, can you avoid duplication by using a method or `if` statement(s)?)
- d. Display the contents of the current order. This option is only available if there is an order in progress.
- e. Place the current order. Selecting this option causes the program to print out the receipt for the current order, and then delete it (by setting any references to this `Order` to `null`). This option is only available if there is an order in progress.
- f. Cancel the current order. Selecting this option causes any references to the current `Order` to be set to `null`. This option is only available if there is an order in progress.
- g. Quit the program.

Each time the menu is displayed, if there is an order currently in progress, a short summary of the order should be displayed as well (use `Order`'s `toString()` method for this).

5. Extra Credit options

You may complete one or more of the following additions to your Java Fiendz program for extra credit. Each of the options below is worth 1 additional point toward your final grade (e.g., earning a perfect grade on Homework 4 plus satisfactorily completing two of the options below will earn 10 points for your final grade instead of the normal 8 for Homework 4).

- a. Java Fiendz is currently promoting a new coffee option: the Javanator™ extreme caffeine shot (warning: keep away from children, pregnant women, the elderly, and those with a nervous disposition). Create a new `Javanator` subclass of the

`CoffeeOption` class. Each Javanator option costs \$0.75. Note that, on the advice of the United States Surgeon General and the Java Fiendz legal department, no more than two (2) `Javanator` shots can be added to a single beverage. You will need to find a way to modify your `Coffee` class to enforce this restriction.

- b. Coffee isn't the only beverage sold at Java Fiendz, despite their name; they also sell tea (at the same prices as hot coffee), and plan to expand to other drinks in the future. Add a `Tea` class to your program. Your `Tea` class should follow the same model as your `Coffee` class, with just one critical difference: tea can only accept cream (milk) and sugar as options, not flavoring shots (or the Javanator). This means that you will need to modify the `Tea` class so that it only accepts `Cream` and `Sugar` objects via its `addOption()` method.

Hint: Use inheritance (like a `Beverage` parent class) to solve the problem of duplicated code. Clever use of inherited methods and overloading will also solve the `CoffeeOption` problem. This means that you will need to modify your `Coffee` class, but it shouldn't affect the `IcedCoffee` class.

- c. Add a graphical interface (GUI) to your program. This option is worth either 1 or 2 points, depending on the scale and complexity of your interface. Using simple interface widgets like `JButton` and `JTextField` qualifies for 1 point of extra credit. Adding more advanced things like menus and `JComboBox` widgets (see chapter 11 of the textbook) will increase the extra credit to 2 points.

Option 2: Goats and GUIs

The "Monty Hall Problem" takes its name from the classic television game show "Let's Make a Deal," which was hosted by Monty Hall. During the show, a contestant is shown three doors, labeled 1, 2, and 3. One of the doors hides a prize (like a brand new sports car); the remaining doors each conceal a (live) goat. The host (Monty Hall) is the only one who initially knows what is behind which door.

After the contestant selects one of the three doors, Monty opens one of the other doors to reveal a goat. The prize is still located behind one of the unselected doors, and the other door still conceals a goat. The contestant now has the option of sticking with his original choice or switching to the other door. Monty then opens the selected door to reveal what the contestant has ultimately won.

This problem is a favorite in probability and statistics classes. Should the contestant switch doors when he has the chance? Should he stay with his original choice? Does it make a difference? How often does the contestant leave with the sports car, and how often with a goat?

Write a simple GUI program that simulates this game. Use a random number (1, 2, or 3) to select the winning door (see the end of this document for a brief description of Java's `Random` class). The GUI should display three buttons, one for each door. The user can

click on a button to select that door. After the user chooses a door, the program should disable one of the other door buttons and change its label to reflect that it hid a goat (Hint: use `JButton`'s `setEnabled()` and `setText()` methods for this part). The user can then click on one of the two remaining buttons to open that door, at which point those door buttons are relabeled with their contents.

Your GUI should also include a "reset" button that will allow the player to try again. Include `JLabels` that show how many times the game has been played, how many times the contestant switched doors, how many times the player won the car, and how many times the player won a goat. Note that `JLabel` also has a `setText()` method.

Extra Credit: You may complete one or both of the following additions to your Monty Hall program for extra credit. Each of the options below is worth 1 additional point toward your final grade (e.g., earning a perfect grade on Homework 4 plus satisfactorily completing both options below will earn 10 points for your final grade instead of the normal 8 for Homework 4).

1. Improve your program's GUI by adding message dialog boxes (see chapter 11 of the textbook) to guide the user through the program and to report the results of a particular round.
2. Modify your program so that it creates a text file that keeps a record of the rounds of the game. For each round of the game, your log file should list the three doors with their respective prizes, the user's door choice(s), and whether the user won or lost. Your log file should also record the current statistics (as percentages) for wins and losses.

Appendix: Random Number Generation in Java

Java provides a class named `Random` (in the `java.util` package) that can be used to generate random numbers. Before you generate any random numbers, you must first create a `Random` object:

```
Random r = new Random();
```

Once you have done this, call the `nextInt()` method on your `Random` object to generate a new random integer value. If you supply an integer argument `n` to `nextInt()`, the method will return a random integer between 0 and `(n-1)`, inclusive. For example,

```
r.nextInt(5);
```

will return a value of 0, 1, 2, 3, or 4. Be sure to save this value in a variable so that you can use it later!

If you want to make sure that a random number falls within a certain range, add the minimum value of the range to the results of `nextInt()`. Use the total number of values in the desired range as the argument to `nextInt()`.

For example, if you wanted to generate a random value between 10 and 20 inclusive (with 11 possible values in all), you would write something like the following:

```
int value = r.nextInt(11) + 10; // minimum value desired is 10
```

In this case, `nextInt()` will generate a random integer between 0 and 10. Adding 10 to that value will shift the result into the range 10–20.