P01 – SHOPPING CART

Posted on January 19, 2019 by Mouna AYARI BEN HADJ KACEM

- **SUBMIT your completed assignment by 9:59PM on Wednesday, January 30th 2019.** We will accept and grade submissions made through 9:59PM on Thursday, January 31st 2019 (HARD DEADLINE). But, NO CREDIT will be granted for any work that is submitted even one second after this hard deadline.
- **Pair Programming is allowed but not required for this assignment. Register your partnership no later than Monday, January 28th to work with a partner. If you have problems accessing this form, try following the advice here.**

CHANGE LOG

If you copy the command menu directly from the write-up, newline will give you "\r\n" on windows but "\n" on another environment. I just updated the last test on zybooks to consider only "\n" as line separator. Make sure to consider "\n" instead of "\r\n" in your implementation. It is fine also to use System.out.println(). It works fine with "\n".

If the command menu is represented by a String, it will be the following:

```
"\nCOMMAND MENU:\n" +
" [P] print the market catalog\n" +
" [A <index>] add one occurrence of an item to the cart given its identifier\n" +
" [C] checkout\n" + " [D] display the cart content\n" +
" [O <index>] number of occurrences of an item in the cart given its identifier\n" +
" [R <index>] remove one occurrence of an item from the cart given its identifier\n" +
" [Q]uit the application\n"
```

OVERVIEW

In this assignment, we are going to implement a simple version of a Shopping Cart using Java Arrays. The Java array is one of several data storage structures that can be used to store and manage a collection of data. Throughout CS300, we are going to spend a fair amount of time using arrays and managing collections of data. This relatively simple programming assignment provides a review of using arrays (perfect size as well as oversize arrays).

OBJECTIVES AND GRADING CRITERIA

The goals of this assignment include:

- reviewing the use of arrays and procedure oriented code (prerequisites for this course),
- developing array-processing algorithms,
- practicing how to manage an unordered collection of data that may contain duplicates (multiple occurrences of the same element),
- developing tests to demonstrate the functionality of code, and familiarizing yourself with CS300 grading tests.

20	Online Tests: these automated grading test results are visible upon uploading your submission. You are allowed multiple opportunities to correct the organization and functionality of your code (if necessary).
20 points	Offline Tests: these automated grading tests are run after the assignment's deadline has passed. They check for similar functionality and organizational correctness as the Online Tests. Since you will not have opportunities to make corrections after seeing the feedback from these tests, you should consider and test the correctness of your own code as thoroughly as possible.
10 points	Code Readability: human graders will review the commenting, style, and organization of your final submission. They will be checking whether it conforms to the requirements of the CS300 Course Style Guide. Since you will not have opportunities to make corrections

after seeing the feedback from these graders, you should consider and review the readability of your own code as thoroughly as possible.

SUBMISSION

For this assignment, you will need to submit two files through zybooks: ShoppingCart.java and ShoppingCartTests.java. You can make as many submissions as you would like prior to the assignment deadline.

INTRODUCTION

We are going to implement a shopping cart that represents a collection of market items. This cart will be represented by an array. In this application, we assume the following:

- The cart stores a collection of items of type String (descriptions for market items),
- Each market item is identified by an index (unique number)
- The shopping cart may contain multiple occurrences of the same item,
- Users can ask how many occurrences of a given item are in the shopping cart (may be 0 or more)
- Users can remove an item from the shopping cart,
- Users can checkout the shopping cart (i.e. display number of items in the shopping cart together with total cost),
- Users can display the content of the shopping cart.

We note also that the market product catalog (set of products available for sale in the market) and the shopping cart (set of items selected by the user for purchase) are stored in two different places.

The following is a demo of your program. Note that the user's input below is shown in green, and that the rest of the text below was printed out by the program.

```
=============   Welcome to the Shopping Cart App   =============


COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: p
++++++++++++++++++++++++++++++++++++++++++++
Item id         Description      Price
++++++++++++++++++++++++++++++++++++++++++++
0               Apple            $1.59
1               Avocado          $0.59
2               Banana           $0.49
3               Beef             $3.79
4               Blueberry        $6.89
5               Broccoli         $1.79
6               Butter           $4.59
7               Carrot           $1.19
8               Cereal           $3.69
9               Cheese           $3.49
10              Chicken          $5.09
11              Chocolate        $3.19
12              Cookie           $9.5
13              Cucumber         $0.79
14              Eggs             $3.09
15              Grape            $2.29
16              Ice Cream        $5.39
17              Milk             $2.09
18              Mushroom         $1.79
19              Onion            $0.79
20              Pepper           $1.99
21              Pizza            $11.5
22              Potato           $0.69
23              Spinach          $3.09
24              Tomato           $1.79
++++++++++++++++++++++++++++++++++++++++++++
```

```
COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: a 0

COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: a 24

COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: a 0

COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: A 17

COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: d
Cart Content: Apple, Tomato, Apple, Milk,

COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: O 0
The number of occurrences of Apple (id #0) is: 2

COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: r 0

COMMAND MENU:
 [P] print the market catalog
```

```
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: D
Cart Content: Milk, Tomato, Apple,

COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: c
#items: 3 Subtotal: $5.47 Tax: $0.27 TOTAL: $5.74

COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application

ENTER COMMAND: Q
=============  Thank you for using this App!!!!!  =============
```

STEP1. GETTING STARTED

If you haven't already, either Install Eclipse with Java 8 on your computer, or find a computer that they are already installed on. If you ever experience any problems with your own computer throughout the semester, you are expected to use another computer to complete your work on time (possibly a machine in one of these labs). You are also responsible for maintaining secure back-ups of your progress as you work. The OneDrive and GoogleDrive accounts associated with your UW NetID are often convenient and secure places to store such backups.

Create a new Java8 project in Eclipse. You can name this project whatever you'd like, but P01 Shopping Cart is a descriptive choice. Create a new Class called ShoppingCart within a file named ShoppingCart.java. These exact names are required before you can submit your work for feedback from this assignment's automated grading tests. Your code for this assignment should not be organized in custom packages (it should all be left in the default package). If you need a reminder or introduction to doing this with Eclipse, please see these instructions.

It will be helpful to submit your code (work in progress) for this assignment to the automated grading tests within zybooks both early and often. This will 1) give you time before the deadline to fix any defects that are detected by the tests, 2) provide you with an additional backup of your work, and 3) help track your progress through the implementation of the assignment. These tests are designed to detect and provide feedback about only very specific kinds of defects. **It is your responsibility to do additional testing to verify that the rest of your code is functioning in accordance with this write-up.**

You should also review the CS300 Course Style Guide to review the requirements for styling and documenting your code (which accounts for 20% of this assignment's points). If you are not familiar with JavaDoc style comments, zyBook section 1.2 includes some additional details.

STEP2. DEFINE FINAL FIELDS (CONSTANTS)

Our shopping cart can store a set of market items. Each market item is defined by a unique identifier (index), a description, and a unit price. Let's first define the set of final java fields that will represent this environment. To do so, add the following lines of codes to your ShoppingCart class :

```
1  // Define final parameters
2  private static final int CART_CAPACITY = 20; // shopping cart max capacity
3  private static final double TAX_RATE = 0.05; // sales tax
4
5  // a perfect-size two-dimensional array that stores the available items in the market
6  // MARKET_ITEMS[i][0] refers to a String that represents the description of the item
7  //                    identified by index i
8  // MARKET_ITEMS[i][1] refers to a String that represents the unit price of the item
9  //                    identified by index i in dollars.
10 public static final String[][] MARKET_ITEMS = new String[][] {{"Apple", "$1.59"},
11    {"Avocado", "$0.59"}, {"Banana", "$0.49"}, {"Beef", "$3.79"}, {"Blueberry", "$6.89"},
12    {"Broccoli", "$1.79"}, {"Butter", "$4.59"}, {"Carrot", "$1.19"}, {"Cereal", "$3.69"},
13    {"Cheese", "$3.49"}, {"Chicken", "$5.09"}, {"Chocolate", "$3.19"}, {"Cookie", "$9.5"},
```

```
    {"Cucumber", "$0.79"}, {"Eggs", "$3.09"}, {"Grape", "$2.29"}, {"Ice Cream", "$5.39"},
    {"Milk", "$2.09"}, {"Mushroom", "$1.79"}, {"Onion", "$0.79"}, {"Pepper", "$1.99"},
    {"Pizza", "$11.5"}, {"Potato", "$0.69"}, {"Spinach", "$3.09"}, {"Tomato", "$1.79"}};
```

The shopping cart has a final capacity set to 20 items. You can change this capacity to another value if you want. We note that the CART_CAPACITY field should be used only to create your cart array. You can access the capacity of the cart array elsewhere in your program through the field **length** defined for Java Arrays (for instance cart.length).

We note also that all the market items are subject to a tax rate of 5%. These market items are stored in a perfect size two dimensional array as shown above. Each item is identified by an identifier that represents the index of the first dimension where it is stored in the *MARKET_ITEMS* array. For more details about perfect size arrays, please refer to section 1.5 on zybooks.

STEP3. DESIGN OF THE SHOPPING CART APPLICATION

Now, let's design our ShoppingCart. We are going to use procedural programming paradigm to develop our application. This means that all the operations that you are going to implement will be *static methods*. Based on the properties provided in the introduction section above, the ShoppingCart driver application should understand the methods such as add, occurrencesOf, remove, printMarketCatalog, displayCartContent, and checkout. Given that, the content of the ShoppingCart is variable (may grow by adding new items, or get decreased by removing items), we are going to implement the cart as an oversize array of elements of type String (items descriptions) and keep track of a count variable that represents the number of items present in the cart so far. For more details about oversize arrays, please refer to section 1.6 on zybooks. We also note that the cart represents a collection of **case insensitive** String items.

The design of the ShoppingCart operations is provided here as the six following commented method headings:

```
// adds the item with the given its identifier (index) at the end of the cart
public static int add(int index, String[] cart, int count) {}

// Returns how many occurrences of the item with index itemIndex are present in the shopping cart
public static int occurrencesOf(int itemIndex, String[] cart, int count) {}

// Removes the first (only one) occurrence of itemToRemove if found and returns the number of
// items in the cart after remove operation is completed either successfully or not
public static int remove(String itemToRemove, String[] cart, int count) {}

// returns the total value (cost) of the cart without tax in $ (double)
public static double getSubTotalPrice(String[] cart, int count) {}

// prints the Market Catalog (item identifiers, description, and unit prices)
public static void printMarketCatalog() {}

// Displays the cart content (items separated by commas)
public static void displayCartContent(String[] cart, int count) {}
```

Note that the comments provided above do not represent Javadoc methods comments. Your final submission must be commented with respect to the course style guide.

We are going to use **Test Driven Development** process to design this first program. Using Test Driven Development process, the **tests come first**. So, the first rising question at this step is which method should be tested first? It's difficult to implement only one method and know it works. For instance, if we work on *add* alone, how can we check or assess that an item (**the desired one**) has actually been added to the cart?

To answer this question, I recommend that you take a few minutes to think of which methods should be developed and tested first? Take a piece of paper and write down your suggestion. Then, compare and discuss your notes with your partner if you have one. After that, scroll down to compare your solution with the one that we suggest.

.

.

.

.

.

.

.

.

.

I hope that you came up with a good plan!

One solution is to develop *OccurrencesOf* at the same time as *add* method and verify both are working together. For instance, a test method could add several elements and verify they are present in the cart by calling *occurrencesOf*. So, we are going to develop *add()* and *occurrencesOf()* methods first. Before starting their implementation, let's design the first unit test to assess the good functioning of both of them with respect to a simple scenario.

We'll begin with a simple unit test defined by one method that adds one item to the shopping cart. *OccurrencesOf* should return 0 before the call of *add* and 1 after that. To do so, let's first create the ShoppingCartTests class and add one unit test method called *testAddAndOccurrencesOfForOnlyOneItem()*. Since it is the first test method to be developed in this course, we provide you first with this simple test method called *testCountIncrementedAfterAddingOnlyOneItem()*. This method returns true if *count* is incremented after adding only one item to the cart, false otherwise.

```
1  // File Header comes here
2
3  // JavaDoc class Header comes here
4  public class ShoppingCartTests {
5
6    /**
7     * Checks whether the total number of items within the cart is incremented after adding one item
8     * @return true if the test passes without problems, false otherwise
9     */
10   public static boolean testCountIncrementedAfterAddingOnlyOneItem() {
11     boolean testPassed = true; // boolean local variable evaluated to true if this test passed,
12                                // false otherwise
13     String[] cart = new String[20]; // shopping cart
14     int count = 0; // number of items present in the cart (initially the cart is empty)
15
16     // Add an item to the cart
17     count = ShoppingCart.add(3, cart, count); // add an item of index 3 to the cart
18     // Check that count was incremented
19     if (count != 1) {
20       System.out.println("Problem detected: After adding only one item to the cart, "
21           + "the cart count should be incremented. But, it was not the case.");
22       testPassed = false;
23     }
24     return testPassed;
25   }
26
27 }
```

Now, we provide you with an example of implementation for *testAddAndOccurrencesOfForOnlyOneItem()* method to get better familiar with unit test methods implementations. You can use the provided code within your ShoppingCartTests file.

```
1  // File Header comes here
2
3  // JavaDoc class Header comes here
4  public class ShoppingCartTests {
5
6    /**
7     * Checks whether the total number of items within the cart is incremented after adding one item
8     * @return true if the test passes without problems, false otherwise
9     */
10   public static boolean testCountIncrementedAfterAddingOnlyOneItem() {
11     boolean testPassed = true; // boolean local variable evaluated to true if this test passed,
12                                // false otherwise
13     String[] cart = new String[20]; // shopping cart
14     int count = 0; // number of items present in the cart (initially the cart is empty)
15
16     // Add an item to the cart
17     count = ShoppingCart.add(3, cart, count); // add an item of index 3 to the cart
18     // Check that count was incremented
19     if (count != 1) {
20       System.out.println("Problem detected: After adding only one item to the cart, "
21           + "the cart count should be incremented. But, it was not the case.");
22       testPassed = false;
23     }
24     return testPassed;
25   }
26
27   /**
28    * Checks whether add and OccurrencesOf return the correct output when only one item is added to
29    * the cart
30    *
31    * @return true if test passed without problems, false otherwise
32    */
33   public static boolean testAddAndOccurrencesOfForOnlyOneItem() {
34     boolean testPassed = true; // evaluated to true if test passed without problems, false otherwise
35     // define the shopping cart as an oversize array of elements of type String
36     // we can set an arbitrary capacity for the cart - for instance 10
37     String[] cart = new String[10]; // shopping cart
38     int count = 0; // number of items present in the cart (initially the cart is empty)
39
40     // check that OccurrencesOf returns 0 when called with an empty cart
41     if (ShoppingCart.occurrencesOf(10, cart, count) != 0) {
42       System.out.println("Problem detected: Tried calling OccurrencesOf() method when the cart is "
43           + "empty. The result should be 0. But, it was not.");
44       testPassed = false;
```

```
    }

    // add one item to the cart
    count = ShoppingCart.add(0, cart, count); // add an item of index 0 to the cart

    // check that OccurrencesOf("Apples", cart, count) returns 1 after adding the item with key 0
    if (ShoppingCart.occurrencesOf(0, cart, count) != 1) {
      System.out.println("Problem detected: After adding only one item with key 0 to the cart, "
          + "OccurrencesOf to count how many of that item the cart contains should return 1. "
          + "But, it was not the case.");
      testPassed = false;
    }

    return testPassed;
  }

  /**
   * main method used to call the unit tests
   *
   * @param args
   */
  public static void main(String[] args) {
    System.out.println("testCountIncrementedAfterAddingOnlyOneItem(): "
        + testCountIncrementedAfterAddingOnlyOneItem());
    System.out.println(
        "testAddAndOccurrencesOfForOnlyOneItem(): " + testAddAndOccurrencesOfForOnlyOneItem());
  }
}
```

You can notice from the above two test methods that we varied the cart capacity from a test method to another. We considered the value of 20 within *testCountIncrementedAfterAddingOnlyOneItem()* and 10 within *testAddAndOccurrencesOfForOnlyOneItem()*. This is simply to highlight that all your methods should work for any valid length value for the cart (non-zero positive value). They should not depend on a given constant for that length. We also recall that the provided CART_CAPACITY static field is defined within the ShoppingCart class to be used only to create the cart array. Its value can be easily accessible through the length field defined for any java array.

Of course, these two unit tests do not compile yet. Your ShoppingCart class does not contain *add* and *occurrencesOf* methods yet. To allow this unit test method to compile, you can add the two methods *add* and *occurrencesOf* written as stubs (a temporary substitute for yet-to-be-developed code) to your ShoppingCart class. Make sure to add the javadoc method headers to these methods.

For instance, you may add the following lines of code to allow your first unit tests to compile. The tests won't pass, at least not yet.

```
/**
 * adds the item with the given identifier index at the end of the cart
 *
 * @param index of the item within the marketItems array
 * @param cart shopping cart
 * @param count number of items present within the cart before this add method is called
 * @return the number of items present in the cart after the item with identifier index is added
 */
public static int add(int index, String[] cart, int count) {
  // TODO complete this method
  return 0;
}

/**
 * Returns how many occurrences of the item with index itemIndex are present in the shopping cart
 *
 * @param itemIndex identifier of the item to count its occurrences in the cart
 * @param cart shopping cart
 * @param count number of items present within the cart
 * @return the number of occurrences of item in the cart
 */
public static int occurrencesOf(int itemIndex, String[] cart, int count) {
  // TODO complete this method
  return 0;
}
```

Important note:

We note that in this first programming assignment, we do not check for invalid input parameters or exceptional situations. We assume that all parameters and input arguments are correct and valid. It is OK if your program will crash for an unexpected or invalid input (for instance if the case of use of null array references, or invalid market item identifier, or indices or counts that are negative or beyond the array's length).

STEP4. DEVELOP AND TEST SHOPPING CART OPERATIONS

DEVELOP ADD AND OCCURRENCESOF METHODS

Let's now implement *add* then *OccurrencesOf* methods. *add()* method will simply add the item (description) with the given index to the end of the array cart. It should return the total number of items present within the cart array after the item with identifier index is added. If the cart is full (reaches its capacity) and the user tries to add new item, the following warning message MUST be displayed and the count of the cart should not change after the method returns:

```
"WARNING: The cart is full. You cannot add any new item."
```

occurrencesOf() method returns how many occurrences of the market item of the given itemIndex are present in the cart (may be 0 or more). You can also notice that the cart contains elements of type String (items descriptions). On the other hand, we can add a market item to the cart, only given its index. To help you implement *occurrencesOf()* method (and may be later other ones), you can develop a **private helper** method that returns the item description (String) given its index (int). For instance,

```
private static String getItemDescription(int index) {}
```

We note that you DO NOT need to check for the validity of the provided index (positive number within the length of the *MARKET_ITEMS* array).

OTHER TEST METHODS

You can now run your first unit test *testAddAndOccurrencesOfForOnlyOneItem* and check that your implementation passes the test. This does not guarantee that your code is correct. It assesses that your code is correct with respect to the input data defined in that test. To further check the correctness of your OccurrencesOf and add methods, we invite you to develop the following two test methods with exactly the following signatures:

```
1  // Checks that items can be added more than one time and are found
2  public static boolean testAddOccurrencesOfDuplicateItems() {}
3
4  // Checks that the correct output is returned when the user tries to add too much items to the cart
5  // exceeding its capacity
6  public static boolean testAddingTooMuchItems() {}
```

In addition, your are responsible for defining further test methods to check thoroughly the good functioning of your implementation.

DEVELOP REMOVE METHOD

Let's now implement *remove* operation. Recall that the signature of this method is exactly the following:

```
1  public static int remove(String itemToRemove, String[] cart, int count) {}
```

where *itemToRemove* represents the item to remove from the cart, *cart* represents the shopping cart, and *count* represents the number of market items present in the cart before remove is called. This method returns the number of items in the cart after remove operation is complete.

We note that *remove* operation works as follows. If *itemToRemove* is found to equal one of the strings referenced by the array cart, *remove* effectively takes one of the occurrences of the String element (the first match).

Following the same developing process, before implementing *remove* operation, let's first design a set of test methods (at least one) to assess its good functioning. To do so, you can begin by writing down a set of self-check questions you think you might need to answer to make sure remove operation works well. As usual, we encourage you to compare your notes with your partner. After you have done this, scroll down to see the list of questions we came up with. You may so enrich your list.

.

.

.

.

.

.

.

Sample of Self-check questions:

1. What would happen when an attempt is made to remove an item that is not in the cart?
2. What would happen if an attempt is made to remove an item from an empty cart (count == 0)?
3. Must *remove* maintain the cart items in the same order as in which they are originally added?
4. What would happen when an attempt is made to remove an item that has multiple occurrences in the cart?

Once you answer the above and your self-check questions, you can design a set of test scenarios (situations) that you can consider in order to check the correctness of your *remove* operation (which is not yet implemented).

Among others, you have to implement the following two test methods with exactly the following signatures in your ShoppingCartTests class:

```
// Checks that when only one attempt to remove an item present in the cart is made, only one occurrence of
// that item is removed from the cart
public static boolean testRemoveOnlyOneOccurrenceOfItem() {}

// Checks that remove does not make any change to count (number of items in the cart) when the user
// tries to remove an item not present within the cart
public static boolean testRemoveItemNotFoundInCart() {
```

Now, let's develop the *remove* operation. Take a few minutes to think of a good design for *remove* operation. Then, compare and discuss your solution with your partner if you have one. After that, scroll down to see our solution and compare it to yours.

.

.

.

.

.

.

.

.

.

We first note that our cart represents an unordered collection of Strings. This means that the order of items in the array cart is not important. So, *remove* operation should not maintain the items present in the cart in the same order as that in which they were originally added. [There other collections used to store elements in order]. Given that, the algorithm that we propose to remove an item from the shopping cart is as follows (other algorithms also work; but you are required to implement this one for this assignment):

- Find the <index of *itemToRemove* in the cart>, or set it to -1 if it does not exist in the cart
- If *itemToRemove* found,
    - move the element at the end of the array to this index
    - Update count
- If *itemToRemove* not found, display the following warning message
    - `"WARNING: " + itemToRemove + " not found in the shopping cart."`
- return count

To help you implement *remove* operation, you HAVE TO implement the following private helper method:

```
/**
   * Returns the index of an item within the shopping cart
   *
   * @param item description
   * @param cart Shopping cart
   * @param count number of items present in the shopping cart
   * @return index of the item within the shopping cart, and -1 if the item does not exist in the
   *         cart
   */
  private static int indexOf(String item, String[] cart, int count) {}
```

Note here that indexOf does not return the identifier of the provided item (i.e. its index within the *MARKET_ITEMS* array). It returns the index of item within the cart array.

Once your *remove* operation is implemented, you can test its functioning with accordance to the instructions provided in the write-up. Recall that it is your responsibility to verify the correctness of your code using the appropriate test methods.

DEVELOP THE REST OF YOUR SHOPPING CART OPERATIONS

Now, following the same development process, implement the following methods:

```
// returns the total value (cost) of the cart without tax in $
public static double getSubTotalPrice(String[] cart, int count) {}

// prints this Market Catalog (item identifiers, description, and prices)
public static void printMarketCatalog() {}

// Displays the cart content (items separated by commas)
public static void displayCartContent(String[] cart, int count) {}
```

You have to add at least one test method to your ShoppingTests java file to check that *getSubTotalPrice()* method provides the correct output (double that represents the total cost of the cart without tax). We note that you have to check the closeness of this double rather than comparing for exact equality (nearly equal number with respect to two significant decimal digits for instance).

We note also that you can add as many private helper methods as you judge necessary to implement your program. For instance, you can develop a method that returns the price of an item given its index (identifier). The following methods from the Java8 API may be also helpful:

- Double.valueOf(String s) returns a Double object holding the double value represented by the argument string s.
- substring(int beginIndex): Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.
- substring(int beginIndex, int endIndex): Returns a new string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex – 1. Thus the length of the substring is endIndex-beginIndex.

The expected output for *printMarketCatalog* and *displayCartContent* methods should be conform to the output format provided at the top of the write-up in the demo of the program. For your information, each line of product within the market catalog is represented by the following String:

```
<identifier> + "\t\t" + <description> + "    \t " + <price>
```

Recall that your are responsible for defining additional test methods to check thoroughly the good functioning of your implementation with respect to the write-up. But, your ShoppingTests class MUST include at least **seven static test methods** (including those provided in this write-up).

STEP5. DRIVER APPLICATION

The final step in this assignment is to implement the main method of the ShoppingCart class. This method serves as the driver of the application. Organize this functionality into whatever custom static methods you see fit. But, make sure that running the main method within your ShoppingCart class results in an interaction section comparable to the sample shown at the top of the write-up. Any new variables that you create for this driver must be local within some static method. Pay close attention to the details within that example, to ensure that your program's welcome message, thank you message, command prompts, the different outputs are all printed in the same manner. Here are some specific requirements for how this interactive session should proceed:

- You do not need to worry about erroneous input from the user, because all of our grading tests will focus on properly encoded commands, as described within this specification.
- You MUST create and use only one instance of the Scanner class in your entire program.
- All commands are case insensitive and are provided in the following command menu:

```
COMMAND MENU:
 [P] print the market catalog
 [A <index>] add one occurrence of an item to the cart given its identifier
 [C] checkout
 [D] display the cart content
 [O <index>] number of occurrences of an item in the cart given its identifier
 [R <index>] remove one occurrence of an item from the cart given its identifier
 [Q]uit the application
```

For instance,

- If the first non-white-space character within the command string is a P (upper or lower case), the market catalog should be displayed.
- If the first character in the user command line is a C (upper or lower case), a checkout operation will be proceeded and the number of items in the shopping cart together with subtotal (without tax) and total (including tax) should be displayed conforming to the following format:

```
"#items: " + <count> + " Subtotal: $" +  String.format("%.2f",
    <subtotalCost> ) + " Tax: $" + String.format("%.2f",<TAX>) + " TOTAL: $" +
    String.format("%.2f",<totalCost>)
```

The following methods from the Java8 API may be helpful while processing the user input command line:

- string.charAt(index) – returns the character at the specified zero-based index from within this string.
- string.toUpperCase() – returns a new copy of the string this method is called on, but with an upper case version of each alphabetic character.
- Integer.parseInt(String s): Parses the string argument as a signed decimal integer.

- White space is the command argument separator. Extra white spaces should be ignored while processing the user input command.
- String.split(): You can split the command line arguments using String.split(" ") and get the array of strings storing each argument of the command line in order.
- Finally, in order to avoid potential or solve Zybooks encoding problem that may occur when submitting your sourcecode files on zybooks, please refer to the following piazza note.

SUBMISSION

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the course style guide, you should submit your final work through zybooks. The most recent of your highest scoring submissions prior to the deadline of **9:59PM on Thursday January 31st (HARD DEADLINE)** will be recorded as your zybooks test score. NO CREDIT will be granted for any work that is submitted even one second after this hard deadline. The second portion of your grade for this assignment will be determined by running that same submission against additional automated grading tests after the submission deadline. Finally, the third portion of your grade for your P01 submission will be determined by humans looking for organization, clarity, commenting, and adherence to the course style guide.

EXTRA CHALLENGES

Here are some suggestions for interesting ways to extend this simulation, after you have completed, backed up, and submitted the graded portion of this assignment. **No extra credit will be awarded for implementing these features**, but they should provide you with some valuable practice and experience. DO NOT submit such extensions using zyBooks.

- When taking input from the user, you can check the validity of that input and print a warning message if the syntax of the command line is incorrect.
- You can add many occurrences of the same item using one command line. For instance, you can extend the add command as follows:

```
[A <index> <number>] add <number> occurrences of an item given its identifier index
```

- You can display the summary of the cart content organized by groups of duplicate items if they exist in the cart associated with the subtotal.