P01 Shopping Cart

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 and oversize arrays).

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P1 pre-assignment quiz is accessible through Canvas before having access to this specification by **11:59PM on Sunday 01/30/2022**. You CANNOT take the pre-assignment quiz for credit passing its deadline. But you can still access it. The pre-assignment quiz contains hints which can help you in the development of this assignment.
20 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is otherwise correct. To become more confident of this, you should run additional tests of your own.
15 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.
10 points	**Manual Grading Feedback:** After the deadline for an assignment has passed, the course staff will begin manually grading your submission. We will focus on looking at your algorithms, use of programming constructs, and the style and readability of your code. This grading usually takes about a week from the hard deadline, after which you will find feedback on Gradescope.

Learning Objectives

The goals of this assignment include:

- reviewing the use of procedure oriented code (prerequisites for this course),
- practicing the use of control structures, custom static methods, and arrays in methods.
- practicing how to manage a non-ordered collection of data (a bag) which may contain duplicates (multiple occurrences of the same element),
- learning how to approach an algorithm, and how to develop tests to demonstrate the functionality of code, and familiarizing yourself with the CS300 grading tests.

Additional Assignment Requirements and Notes

(Please read carefully!)

- Make sure to read carefully through the WHOLE specification provided in this write-up before starting its implementation. Samples of inputs/outputs are provided at the last section of this specification. Read TWICE the instructions and do not hesitate to ask for clarification on piazza if you find any ambiguity.
- Pair programming is **NOT ALLOWED** for this assignment. You MUST complete and submit your p01 INDIVIDUALLY.
- **NO import statements** are allowed in your `ShoppingCart` class.
- **You CAN import** java.util.Arrays in your `ShoppingCartTester` class.
- Only your submitted `ShoppingCartTester` class contains a **main** method.
- You are NOT allowed to add any constant or variable not defined or provided in this write-up outside of any method.
- You CAN define local variables that you may need to implement the methods defined in this program.
- You CAN define **private static** helper methods to help implement the different public static methods defined in this write-up.
- You MUST NOT add any **public methods** to your `ShoppingCart` class other than those defined in this write-up.
- All your test methods should be defined and implemented in your `ShoppingCartTester.java`.

- All your test methods must be **public static**. Also, they must take **zero arguments**, **return a boolean**, and must be defined and implemented in your `ShoppingCartTester` class.
- We DO NOT consider erroneous input or exceptional situations in this assignment. We suppose that all the input parameters provided to method calls are valid.
- 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.
- Make sure to submit your code (work in progress) of this assignment on Gradescope 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 implement additional testing to verify that the rest of your code is functioning in accordance with this write-up.**
- You can submit your work in progress multiple times on gradescope. Your submission may include methods not implemented or with partial implementation or with a default return statement. But avoid submitting a code which does not compile. A submission which contains compile errors won't pass any of the automated tests on gradescope.
- Feel free to **reuse** any of the provided source code in this write-up verbatim in your own submission.
- All implemented methods including the main method MUST have their own javadoc-style method headers, with accordance to the CS300 Course Style Guide.
- All your classes MUST have a javadoc-style class header.
- If you are not familiar with JavaDoc style comments, zyBook section 1.4 includes some additional details.
- You MUST adhere to the Academic Conduct Expectations and Advice

1 Getting Started

Start by creating a new Java Project in eclipse called P01 Shopping Cart, for instance. You MUST ensure that your new project uses Java 11, by setting the "Use an execution environment JRE:" drop down setting to "JavaSE-11" within the new Java Project dialog box. DO NOT create any module within to your java project. Then, create two Java classes/source files within that project's src folder (both inside the **default** source package) called `ShoppingCart` and `ShoppingCartTester`.

- ONLY the `ShoppingCartTester` class should include a main method.
- DO NOT generate or add a main method to the `ShoppingCart` class. If you need a reminder or instructions on creating projects with source files in Eclipse, please review these instructions.

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

- The cart stores a collection of **case-sensitive** String items (descriptions or names for market items).
- Each market item is identified by a unique identifier (4-digit number).
- The shopping cart may contain multiple occurrences of the same item.
- Items can be added to and removed from the shopping cart.
- The shopping cart is a bag (not-ordered array). This means that the order of elements in the array does not matter.
- At checkout, all items in the shopping cart are subject to a tax rate of 5%.
- The set of products available for sale in the market are stored in a **perfect size** two-dimensional array. The market catalog represented by this array does not contain duplicate items (meaning items with the same identifier).

In this assignment, we are going to use procedural programming to develop our self-checkout application. This means that all the methods that you are going to implement will be *static methods*. Besides, this assignment involves the use of arrays within methods in Java. So, before you start working on the next steps, make sure that you have already completed chapter1 zybooks reading activities.

2 Define the final static fields (Constants)

Your `ShoppingCart` class MUST contain the following static final fields (constants) defined outside of any methods. The top of the class is a good placement to declare these constants. NO additional constants or variable must be added to this class (outside of any methods).

```
private static final double TAX_RATE = 0.05; // sales tax

// MarketItems: a perfect-size two-dimensional array that stores the list of
// available items in a given market
```

```
// MarketItems[i][0] refers to a String representation of the item identifiers
// MarketItems[i][1] refers the item name. Item names are also unique
// MarketItems[i][2] a String representation of the unit price
//                 of the item in dollars
private static String[][] marketItems =
    new String[][] {{"4390", "Apple", "$1.59"},
    {"4046", "Avocado", "$0.59"}, {"4011", "Banana", "$0.49"},
    {"4500", "Beef", "$3.79"}, {"4033", "Blueberry", "$6.89"},
    {"4129", "Broccoli", "$1.79"}, {"4131", "Butter", "$4.59"},
    {"4017", "Carrot", "$1.19"}, {"3240", "Cereal", "$3.69"},
    {"3560", "Cheese", "$3.49"}, {"3294", "Chicken", "$5.09"},
    {"4071", "Chocolate", "$3.19"}, {"4363", "Cookie", "$9.5"},
    {"4232", "Cucumber", "$0.79"}, {"3033", "Eggs", "$3.09"},
    {"4770", "Grape", "$2.29"}, {"3553", "Ice Cream", "$5.39"},
    {"3117", "Milk", "$2.09"}, {"3437", "Mushroom", "$1.79"},
    {"4663", "Onion", "$0.79"}, {"4030", "Pepper", "$1.99"},
    {"3890", "Pizza", "$11.5"}, {"4139", "Potato", "$0.69"},
    {"3044", "Spinach", "$3.09"}, {"4688", "Tomato", "$1.79"},
    null, null, null, null};
```

3 Design of the Shopping Cart application

Now, let's design the implementation of our `ShoppingCart` class. 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.

To practice good structured or procedural programming, we will be organizing the implementation of our `ShoppingCart` operations into several easy-to-digest sized methods. This means that we are going to start with the implementation of the easy to solve methods before getting to the relatively harder ones. Also, we want to test these methods before writing code that makes use of calling them. When we do find bugs in the future, we will add additional tests to demonstrate whether those defects exist in our code. This will help us see when a bug is fixed, and it will help us notice if similar bugs surface or return in the future. This design approach is known as **Test Driven Development** process (**Tests come FIRST!**).

We are going to consider two set of operations in the `ShoppingCart` class: operations related to the market items only, and operations related to the shopping cart as a bag of items. An example of the contents of the marketItems array is provided above. The following is an example of representation of a cart which contains the following items Carrot, Beef, Cookie, Tomato, and Cheese. This cart can contain multiple occurrences of the same item.

```
cart: {"Carrot", "Beef", "Cookie", "Tomato", "Cheese", "Cookie", null, null, null}
size: 6
```

3.1 Operations related to the market items

A user should be able to perform the following operations.

- Users can **lookup for items by name or by identifier** within the market catalog (represented by the two dimensional perfect size marketItems array. The returned string will be as follows.
  - if a match with the search item is found in the marketItems array:

    `item_id + " " + item_Name + " " + item_price`
  - if no match found, the following string will be returned:

    `"No match found"`
- Users can **get the price** (in dollars) of an item given its name.
- Users can get a **deep copy** of the market catalog (a deep copy of the marketItems array). A deep copy of an array has the exact same length and contents as the original array, but different memory location (meaning different reference). This copy of the catalog can be used for testing purpose since the `marketItems` array is declared to be a **private** data field, meaning not visible from the outside of the class `ShoppingCart`. We do not also want to allow users to change the contents of the marketItems from the outside of the class `ShoppingCart`.

The four above operations will be implemented by the following methods with the **EXACT** following signatures.

```
// Returns a string representation of the item whose name is
// provided as input if a match was found.
// The format of the returned string is presented above
// name - the name of the product or item to search
public static String lookupProductByName(String name) {}

// Returns a string representation of the item whose id is
// provided as input if a match was found.
// The format of the returned string is presented above
// id - the identifier of the product or item to search
public static String lookupProductById(int id) {}

// Returns the price in dollars (a double value) of a market item
// given its name. If no match was found in the market catalog, this
// method returns -1.0
// name - the name of the product to check its price
public static double getProductPrice(String name) {}
```

```
// Returns a deep copy of the marketItems array
public static String[][] getCopyOfMarketItems(){}
```

You can also notice that the above methods do not compile. You can add a default return value to these methods to let them compile without errors (for instance, empty string or null for type String, 0 for return type int and 0.0 for type double). Note that the comments provided above do not represent Javadoc methods comments. Your final submission must be commented with respect to the CS300 Course Style Guide. Please find below an example of a javadoc method style header for the method `lookupProductByName` provided above.

```
/**
 * Returns details of a specific product in the market given its name
 *
 * @param name name of the product to search
 * @return A string representation of the product to search including
 *         the identifier of the product, its name, and its price in
 *         dollars if match found.
 */
public static String lookupProductByName(String name) {
  return ""; // default statement added to allow this code to compile
}
```

Note that a javadoc style method header represents the **abstraction** of the method (what the method is supposed to do, what it does take as input, and what is expected to return as output) **independently of its implementation details**. Read carefully through the provided specification, and do not hesitate to ask for clarification on piazza if it is not clear to you what every method is supposed to do, take as input, and provide as output.

Now, before writing any code for the above methods, let's implement tester methods to assess the correctness of any implementation of the lookup and get item price methods based on the details provided in the comments of their specification. In your `ShoppingCartTester` class, add the following two tester methods with the exact following signatures. A tester method should return true if and only if no bug is detected. If it detects any bug it returns false. A tester method is better than another tester method if it can detect more bugs within a broken implementation.

```
// Checks whether ShoppingCart.lookupProductByName() and
// ShoppingCart.lookupProductById() methods work as expected.
// Returns true when this test verifies a correct functionality,
// and false otherwise
public static boolean testLookupMethods() { }
```

```
//Checks the correctness of ShoppingCart.getProductPrice() method
// Returns true when this test verifies a correct functionality,
// and false if any bug is detected with respect to the test
// scenarios defined in this tester
 public static boolean testGetProductPrice() { }
```

You can add default return statement (return false for boolean type) to allow the above tester methods to compile. A common trap when writing tests is to make the test code as complex or even more complex than the code that it is meant to test. This can lead to there being more bugs and more development time required for testing code, than for the code being tested. To avoid this trap, we aim to make our test code as simple as possible while varying our test scenarios to account edge cases. Since this is the first assignment, we provide you in the following with a template for the `testLookupMethods()` that you can use and complete. Notice that the `lookupProductByName()` method returns a String. We use the `.equals()` method to compare two strings and not the == operator used to compare references and primitive type variables.

```
/**
 * Checks whether ShoppingCart.lookupProductByName() and
 * ShoppingCart.lookupProductById() methods work as expected.
 * @return true when this test verifies a correct functionality,
 *         and false otherwise
 */
public static boolean testLookupMethods() {
  // define test scenarios.

  // 1. The item to find is at index 0 of the marketItems array
  String expectedOutput = "4390 Apple $1.59";
  if(!ShoppingCart.lookupProductByName("Apple").equals(expectedOutput)) {
    System.out.println("Problem detected: Your lookupProductByName() method "
        + "failed to return the expected output when passed Apple as input");
    return false;
  }
  if(!ShoppingCart.lookupProductById(4390).equals(expectedOutput)) {
    System.out.println("Problem detected: Your lookupProductById() method "
        + "failed to return the expected output when passed the id "
        + "of Apple as input");
    return false;
  }

  // TODO complete the following tester scenarios
  // 2. The item to find is at the last non-null position of
  // the marketItems array
```

```
    // 3. The item to find is at an arbitrary position of the
    // middle of the marketItems array


    // 4. The item to find is not found in the market
    expectedOutput = "No match found";
    if(!ShoppingCart.lookupProductByName("NOT FOUND").equals(expectedOutput)) {
      System.out.println("Problem detected: Your lookupProductByName() method "
          + "failed to return the expected output when passed the name of "
          + "a product not found in the market.");
      return false;
    }
    if(!ShoppingCart.lookupProductById(1000).equals(expectedOutput)) {
      System.out.println("Problem detected: Your lookupProductById() method "
          + "failed to return the expected output when passed the identifier"
          + "of a product not found in the market.");
      return false;
    }
    // You may add other test scenarios

    return true; // NO BUGS detected by this tester method
  }
```

Notice carefully that the tester method includes only ONE `return true` statement at the end (last statement). Any additional test scenario must be added before that `return true` statement. If any bug is detected, return false. It is recommended to print out more specific feedback when tests fail (before returning false). This helps you detect which test scenario was failed by the broken implementation, locate the bug, and fix it.

You can also notice how the above test method helps clarify the requirements and expected behavior of the *lookupProductByName()* and *lookupProductById()* methods. Note that when you encounter a problem or question about your code, start by creating a test like this to verify your understanding of what is happening, versus what should be happening when your code runs. Sharing tests like this with the course staff who are helping you throughout this semester is a great way to help the course staff assist you more efficiently.

Now, you can add a call to this test method from the *main* method of your `ShoppingCartTester` class. The following is an example.

```
  /**
   * Main method
   * @param args input arguments if any
   */
  public static void main(String[] args) {
    System.out.println("testLookupMethods(): " + testLookupMethods());
  }
```

If you run your tester class, the testLookupMethods() call must return false since the lookup methods are not yet implemented in your `ShoppingCart` class.

Now, think about the test scenarios to consider in your `testGetProductPrice()` method. Please find below a template that you can complete. Notice that the `getProductPrice()` method defined in the `ShoppingCart` class returns a double. Be careful while comparing two double values (floating-point numbers) in java. We do not use the == operator for this purpose. Please find below a template for the `testGetProductPrice()` method that you can use and expand.

```
/**
 * Checks the correctness of ShoppingCart.getProductPrice() method
 * @return true when this test verifies a correct functionality,
 *         and false otherwise
 */
public static boolean testGetProductPrice() {
  // define test scenarios

  // first test scenario: get the price of Apple
  double expectedPrice = 1.59; // price of the product Apple in the market
  // Note that we do not use the == operator to check whether two
  // floating-point numbers (double or float) in java are equal.
  // Two variables a and b of type double are equal if the absolute
  // value of their difference is less or equal to a small threshold epsilon.
  // For instance, if Math.abs(a - b) <= 0.001, then a equals b

  if(Math.abs(ShoppingCart.getProductPrice("Apple") - expectedPrice) > 0.001) {
    // print feedback to report the detected bug
    return false;
  }

  // TODO add at least two other test scenarios while varying the list
  // of input passed to the getProductPrice method calls.
  // Make sure to consider the case of a product not found in the market

  return true; // No bug detected. The ShoppingCart.getProductPrice()
               // passed this tester.
}
```

Now, you can go and implement the three methods that we already defined in the `ShoppingCart` class. Run your tester methods to check the correctness of your implementation with respect to the test scenarios that you defined. Make sure to avoid or fix IndexOutOfBoundsExceptions or NullPointerExceptions which can be thrown from your code as a result of a misuse of the reference of the array, its indexes, or the references to its elements.

Recall that you are not allowed to import java.util.Arrays in your `ShoppingCart` class. To

implement the `getCopyOfMarketItems()` method, you must create a new two dimensional array of the same type and length of the `marketItems` array and use a for-loop to copy all its elements to the deep copy that you created. You are not allowed to use `java.util.Arrays.copyOf()` method.

3.2 Operations related to the shopping cart bag

In addition to the functionalities implemented in the previous section, a user of the shopping cart can perform the following operations.

- Users can add items to the cart.
- Users can ask how many occurrences of a given item are in the cart (may be 0 or more).
- Users can check whether a cart contains a given item.
- Users can remove items from the cart.
- Users can checkout the cart (get its total cost).
- Users can display a summary of the content of the cart.
- Users can empty the cart (remove all items from the cart).

The above operations will be implemented by the following methods with the exact following signatures in your `ShoppingCart` class. Recall that:

- The shopping cart bag is going to be represented by an oversize array (a reference to an array of sensitive-case strings and an int keeping track of the number of items in the cart). These two parameters will be passed as input to all these methods.
- Elements stored in the oversize array cart must be in the range of indexes 0 .. size-1 with no gaps. All null references must be at the range of indexes size .. cart.length-1.

```
// Appends an item to a given cart (appends means adding to the end).
// If the cart is already full (meaning its size equals its length),
// the item will not be added to the cart.
// item - the name of the product to be added to the cart
// cart - an array of strings which contains the names of items in the cart
// size - the number of items in the cart
// Returns the size of the oversize array cart after trying to add item
// to the cart. This method returns the same of size without making
// any change to the contents of the array if it is full.
public static int addItemToCart(String item, String[] cart, int size) {}
```

```
// Returns the number of occurrences of a given item within a cart. This
// method must not make any changes to the contents of the cart.
// item - the name of the item to search
// cart - an array of strings which contains the names of items in the cart
// size - the number of items in the cart
// Returns the number of occurrences of item (exact match) within the oversize
// array cart. Zero or more occurrences of item can be present in the cart.
public static int nbOccurrences(String item, String[] cart, int size) {}


// Checks whether a cart contains at least one occurrence of a given item.
// This method must not make any changes to the contents of the cart.
// item - the name of the item to search
// cart - an array of strings which contains the names of items in the cart
// size - the number of items in the cart
// Returns true if there is a match (exact match) of item within the
// provided cart, and false otherwise.
public static boolean contains(String item, String[] cart, int size) {}


// This method returns the total value in dollars of the cart. All
// products in the market are taxable (subject to TAX_RATE).
// cart - an array of strings which contains the names of items in the cart
// size - the number of items in the cart
// Returns the total value in dollars of the cart accounting taxes.
public static double checkout(String[] cart, int size) {}


// Removes one occurrence of item from a given cart. If no match with item
// was found in the cart, the method returns the same value of input size
// without making any change to the contents of the array.
// item - the name of the item to remove
// cart - an array of strings which contains the names of items in the cart
// size - the number of items in the cart
// Returns the size of the oversize array cart after trying to remove item
// from the cart.
public static int removeItem(String[] cart, String item, int size) {}


// Removes all items from a given cart. The array cart must be empty (contains
// only null references) after this method returns.
// cart - an array of strings which contains the names of items in the cart
// size - the number of items in the cart
// Returns the size of the cart after removing all its items.
public static int emptyCart(String[] cart, int size) {}
```

```
// Returns a string representation of the summary of the contents of a given cart.
// The format of the returned string contains a set of lines where each line contains
// the number of occurrences of a given item, between parentheses, followed by
// one space followed by the name of a unique item in the cart.
// (#occurrences) name1
// (#occurrences) name2
// etc.
// Further details about the format of the returned string is provided
// in the next section.
// cart - an array of strings which contains the names of items in the cart
// size - the number of items in the cart
// Returns a string representation of the summary of the contents of the cart
public static String getCartSummary(String[] cart, int size) {}
```

In order to assess the good functioning of the above methods, you are required to add and implement the following tester methods with the exact following signatures. Examples of input/output are provided in the next section.

```
// This tester method checks the correctness of addItemToCart,
// contains, and nbOccurrences methods defined in the ShoppingCart class
public static boolean testAddItemToCartContainsNbOccurrences() {}

// This tester method checks the correctness of removeItem() method
// defined in the ShoppingCart class
public static boolean testRemoveItem() {}


// This tester method checks the correctness of checkout and
// getCartSummary() methods defined in the ShoppingCart class
public static boolean testCheckoutGetCartSummary() {}

// This tester runs all the tester methods defined in this tester class.
// For instance, if this tester class defines three tester methods
// named test1(), test2() and test3(), it will return test1() && test2() && test3()
// Returns false if any of the tester methods fails, and true if all
// the tests are passed.
public static boolean runAllTests() {}
```

In total, your tester class `ShoppingCartTester` must contain at least SIX test methods and a main() method. Please feel free to define further test methods to assess the correctness of the other methods defined in the `ShoppingCart` class.

Please find below some hints about the test scenarios to consider for each method.

- **testAddItemToCartContainsNbOccurrences():** Make sure to consider at least the following scenarios. (1) adding an item to an empty cart, (2) adding an item to a full cart, (3) adding successfully an item to a non-empty cart (you can consider cart arrays with different capacities and sizes. For successful add operations, check that the new item has been correctly added at the end of the array cart, that the size was updated (correct return value). Along, you can check the correctness of `nbOccurrences` and `contains` methods.

- **testRemoveItem():** Make sure to consider at least the following scenarios. (1) removing an item stored at index 0 of a non-empty cart, (2) removing an item whose first occurrence is stored at index size-1 (last index position) of a non-empty cart, (3) removing an item whose first occurrence is stored at an arbitrary position within a non-empty array cart (from 1 .. size-2), (4) trying to remove an item from an empty cart (whose size is zero), (5) trying to remove a non-existing item from the cart. For each scenario, check that the returned size value is as expected, and that the number of occurrences of each element at the cart is as expected after the removeItem() method call returns.

- **testCheckoutGetCartSummary():** We recommend considering at least three test scenarios. (1) Consider calling ShoppingCart.getCartSummary() on an empty cart. The method must return an empty string. (2) Consider calling ShoppingCart.getCartSummary() on a cart which contains non-duplicate items. (3) Consider calling ShoppingCart.getCartSummary() on a cart which contains items with multiple occurrences at different positions of the oversized array. Illustrative examples are provided in the last section 3.3.3 of this write-up.

3.3 Methods managing the cart bag: Samples of inputs/outputs

In the following, we provide you with a set of inputs and their expected outputs associated with the methods managing the shopping cart bag defined in the `ShoppingCart` class.

3.3.1 Adding an item to the cart

- **Empty cart:**

```
input item: "Banana"
input cart: String[] cart = new String[10]; // array containing 10 null references
input size = 0; // no item is stored in the cart

addItemToCart("Banana", cart, size); // must return 1 and the contents of the
cart array is expected to be:
cart: {"Banana", null, null, null, null, null, null, null, null, null}
```

- **Full cart:**

```
input item: "Eggs"
input cart: String[] cart = new String[]{"Milk", "Apple", "Banana", "Pizza"};
input size = 4; // full array (size == cart.length)

addItemToCart("Eggs", cart, size); // must return 4 without making any change
to the contents of array cart.
```

- **Successful adding operation:**

```
input item: "Eggs"
input cart: String[] cart = new String[]{"Milk", "Apple", "Banana", "Pizza",
                            null, null};
input size = 4; // 4 items are stored in the cart

addItemToCart("Eggs", cart, size); // must return 5 and the contents of the
cart array is expected to be:
cart: {"Milk", "Apple", "Banana", "Pizza", "Eggs", null}
```

3.3.2 Removing an item from the cart

- **Empty cart:**

```
input cart: String[] cart = new String[5]; // array containing 5 null
                                              references
input size = 0; // no item is stored in the cart

removeItem(cart, "Apple", size); // must return 0 without making any change
to the contents of array cart.
```

- **Not found item:**

```
input item: "Eggs"
input cart: String[] cart = new String[]{"Milk", "Apple", "Banana",
                            "Pizza", "Milk", null, null};
input size = 5; // 5 items are stored in the cart

removeItem(cart, "Eggs", size); // must return 5 without making any change
to the contents of array cart.
```

- **Successful removal operation:** Note that there are at least two ways to design the implementation of the `removeItem` operation defined in the `ShoppingCart` class. After finding the first occurrence of the item to be removed (for instant element at index i of the array cart), you can:

a. either make a shift of all the elements stored in the range of indexes from i+1 to size-1 one position to the left. Then, set the element at index size-1 to null. This way the order of precedence of the items in the array will be preserved. But this will require the use of a for-loop to implement the shift operation.

b. or since the cart is a bag (non-ordered collection of items), preserving the order of elements in the array cart is not required. Upon finding the first occurrence of the item to be removed (assuming at index i of the array cart), you can simply move the last element at index size-1 to the position i and set the element at index size-1 to null. No shift operation will be required.

```
input item: "Apple"
input cart: String[] cart = new String[]{"Milk", "Apple", "Banana",
                            "Apple", "Pizza", null, null};
input size = 5; // 5 items are stored in the cart

removeItem(cart, "Apple", size); // must return 4 and the contents of the
cart array is expected to be either:

(a) cart: {"Milk", "Banana", "Apple", "Pizza", null, null, null}, or:

(b) cart: {"Milk", "Pizza", "Banana", "Apple", null, null, null},

depending on how the removal operation has been designed.
```

To check the correctness of the removeItem() method in your tester class, it is recommended to rely on the nbOccurrences() of each item rather than the expected contents of the array cart.

3.3.3 Getting the cart summary

We consider the following scenarios to illustrate how the `getCartSummary()` is expected to operate. Recall that this method MUST NOT add or remove any item to or from the cart array provided as input.

- **Empty cart:**

```
input cart: String[] cart = new String[7]; // array containing 7 null
                                           references
input size = 0; // no item is stored in the cart

getCartSummary(cart, size); // must return an empty string ""
```

- **Cart containing unique items:**

```
input cart: String[] cart = new String[]{"Milk", "Apple", "Banana",
                            "Pizza", null, null};
input size = 4; // non-empty cart of size 4

getCartSummary(cart, size); // is expected to return a string which
consists of the following 4 lines in any order.
Output:
(1) Milk
(1) Apple
(1) Banana
(1) Pizza
```

- **Cart containing multiple occurrences of the same items:**

```
input cart: String[] cart = new String[] {"Tomato", "Milk", "Milk",
            "Eggs", "Tomato", "Onion", "Eggs", "Milk", "Banana",
            null, null};
int size = 9

getCartSummary(cart, size); // is expected to return a string which
consists of the following 5 lines. The order of these lines in the
returned string does not matter.
We provide for instance two samples of correct outputs.
Output (a):
(2) Tomato
(3) Milk
(2) Eggs
(1) Onion
(1) Banana

Output (b):
(2) Tomato
(1) Banana
(2) Eggs
(1) Onion
(3) Milk
```

Note that the ShoppingCart.getCartSummary() is the most complex method to implement in this assignment. We recommend start implementing the other methods first. Then, try to come-up with a good way on who to solve the problem associated with this method.

4 Assignment 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 CS300 Course Style Guide, you should submit your final work through Gradescope. The only TWO files that you must submit include: `ShoppingCart.java` and `ShoppingCartTester.java` . Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline. Finally, the third portion of your grade for your submission will be determined by humans looking for organization, clarity, commenting, and adherence to the CS300 Course Style Guide.