P01 Self-Checkout Kiosk

Overview

In this application we are going to develop a simple self-checkout application for a grocery store. The checkout process takes place after the customer finishes shopping and is ready to scan and check out the items in their shopping cart. We are going to use Java Arrays to implement this application. The Java array is one of several data storage structures which can be used to store and manage a collection of data. In this course, we are going to spend a fair amount of time using arrays and managing collections of data. We hope this will be a relatively straightforward review of using arrays (perfect size and oversize arrays).

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P01 pre-assignment quiz is accessible through Canvas before having access to this specification by **11:59PM on Sunday 01/31/2021**.
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 using 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 hpw to develop tests to demonstrate the functionality of code, and familiarizing yourself with the CS300 grading tests.

Additional Assignment Requirements and Notes

- NO import statement is allowed in your `SelfCheckoutKiosk` class.
- You can import `java.util.Arrays` in your `SelfCheckoutKioskTester` class if needed.
- You are not allowed to add any constant or variable outside of any method not defined or provided in this write-up.
- 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.
- 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 `SelfCheckoutKioskTester` class.
- All implemented methods MUST have their own javadoc-style method headers, according to the CS300 Course Style Guide.
- All your classes MUST have a javadoc-style class header.
- DO NOT submit the provided `SelfCheckoutDriver.java` source file on gradescope.
- 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.**
- Feel free to reuse any of the provided source code in this write-up in your own submission.

1 Review of the CS300 Assignments Requirements

Before getting started, if not yet done, make sure to read carefully the advice through the following links which apply to all the CS300 programming assignments.

- Academic Conduct Expectations and Advice,
- **Course Style Guide** Review well the CS300 Course Style Guide, otherwise you are likely to lose points for submitting code that does not conform to these requirements on this and future programming assignments. At the end of the course style guide page are some helpful tips for configuring Eclipse to help with these requirements. Every java source file that you are going to submit on gradescope MUST contain a file header with complete information at the top of the file. Also, pay close attention to the requirements for commenting and the use of JavaDoc style comments in your code. Additional examples of Javadoc style class and method headers are available in the following zyBook reading section.

2 Getting Started

Start by creating a new Java Project in eclipse which you may call **P01 SelfCheckout Kiosk**, for instance. You have to 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. Then, create two Java classes / source files within that project's src folder (both inside the default package) called `SelfCheckoutKiosk` and `SelfCheckoutKioskTester`. Your code for this assignment should not be organized in custom packages (it should all be left in the default package). Note that ONLY the `SelfCheckoutKioskTester` class should include a main method. DO NOT generate or add a main method to the `SelfCheckoutKiosk` class. If you need a reminder or instructions on creating projects with source files in Eclipse, please review these instructions.

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*. Also, 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 the following reading activities on **zyBooks**. You may also refer to these sections at any level of the implementation of this project.

- Common errors: Methods and arrays,
- Perfect Size Arrays,
- Oversize Arrays,
- Methods With Oversize Arrays,
- Comparing perfect size and oversize arrays.

3 Create the SelfCheckoutKiosk class

The checkout process starts when the user/customer proceeds checking out their products using a self-checkout kiosk. The user starts scanning the items from their shopping cart item by item. In this application, each time the user scans an item, it will be automatically added/placed into the **bagging area**.

We define the bagging area in our Checkout Kiosk to be an oversize array defined by the following **two** variables:

- an array of **non-ordered case insensitive** strings which stores the names of the scanned items.
- a int variable which keeps track of its size.

Your `SelfCheckoutKiosk` class must implement all the main operations related to scanning an item (and eventually adding it to the bagging area), removing an item from the bagging area, displaying the checkout summary, getting the total/subtotal of your cart, and more. Note that we are not going to implement the payment process. To do so, let's first define the set of constants (final data fields) related to our SelfCheckoutKiosk class.

3.1 Define the final fields (constants)

First, add the following constants to your `SelfCheckoutKiosk` class. The top of the class body is a good placement where to declare them. These constants must be put outside of any of the methods that you are going to develop later. NO additional constants must be added to this class.

```
public static final double TAX_RATE = 0.05; // sales tax
// a perfect-size two-dimensional array that stores the available items in the grocery store
// GROCERY_ITEMS[i][0] refers to a String that represents the name of the item
// identified by index i
// GROCERY_ITEMS[i][1] refers to a String that represents the unit price of the item
// identified by index i in dollars.
public static final String[][] GROCERY_ITEMS = new String[][] {{"Apple", "$1.59"},
 {"Avocado", "$0.59"}, {"Banana", "$0.49"}, {"Beef", "$3.79"}, {"Blueberry", "$6.89"},
 {"Broccoli", "$1.79"}, {"Butter", "$4.59"}, {"Carrot", "$1.19"}, {"Cereal", "$3.69"},
 {"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"}};
```

Note that every item in the grocery shop has:

- a **unique identifier** (equivalent to the bar code) which represents here its index of the first position where it is stored in the `GROCERY_ITEMS` perfect size array. For instance, the identifier of the item named "Apple" is 0, the identifier of "Avocado" is 1, and so on.

- **a name** which is a string stored at GROCERY_ITEMS[**id**][**0**] where id is the identifier of the item.
- **a unit price** in dollars which is stored at GROCERY_ITEMS[**id**][**1**] where id is the identifier of the item.

We note also that all the grocery items are subject to a tax rate of 5%.

3.2 SelfCheckoutKiosk Operations

To practice good structured programming, we will be organizing the implementation of our `SelfCheckoutKiosk` 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. In addition, 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.

The design of the operations of our SelfCheckoutKiosk class is provided here as the nine following commented method headings:

```
// Returns the name of the item given its index
// index - unique identifier of an item
public static String getItemName(int index) { }

// Returns the price of an item given its index (unique identifier)
// index - unique identifier of an item
public static double getItemPrice(int index) { }

// Prints the Catalog of the grocery store (item identifiers, names, and prices)
public static void printCatalog() {
 // Complete the missing code /* */ in the following implementation
  System.out.println("+++++++++++++++++++++++++++++++++++++++++++++++");
  System.out.println("Item id \tName \t Price");
  System.out.println("+++++++++++++++++++++++++++++++++++++++++++++++");
  for (int i = 0; i < GROCERY_ITEMS.length; i++) {
    System.out.println(/*item id*/ + "\t\t" + /*item name*/ +
   "    \t " + /*$item price*/);
  }
  System.out.println("+++++++++++++++++++++++++++++++++++++++++++++++");
}
```

```java
// Adds the name of a grocery item given its identifier at the end of
// (the bagging area) the oversize array defined by the items array and its size
// If the items array reaches its capacity, the following message:
// "Error! No additional item can be scanned. Please wait for assistance."
// will be displayed and the method returns without making any change
// to the contents of the items array.

// id - identifier of the item to be added to the bagging area
//     (index of the item in the GROCERY_ITEMS array)
// items - array storing the names of the items checked out and
//        placed in the bagging area
// size - number of elements stored in items before trying to add a new item
// Returns the number of elements stored in bagging area after the item
// with the provided identifier was added to the bagging area
public static int addItemToBaggingArea(int id, String[] items, int size) { }


// Returns the number of occurrences of a given item in an oversize array of
// strings. The comparison to find the occurrences of item is case insensitive.
// item - item to count its occurrences
// items - a bag of string items
// size - number of items stored in items
public static int count(String item, String[] items, int size) { }

// Returns the index of the first occurrence of item in items if found,
//        and -1 if the item not found
// item - element to search for
// items - an array of string elements
// size - number of elements stored in items
public static int indexOf(String item, String[] items, int size) { }


// Removes the first occurrence of itemToRemove from the bagging area
// defined by the array items and its size. If no match with
// itemToRemove is found, the method displays the following error
// message "WARNING: item not found." without making any change
// to the items array. This method compacts the contents of the items
// array after removing the itemToRemove so there are no empty spaces
// in the middle of the array.

// itemToRemove - item to remove from the bagging area
// items - a bag of items
// size - number of elements stored in the bag of items
// returns the number of items present in the cart after the
//        itemToRemove is removed from the cart
public static int remove(String itemToRemove, String[] items, int size) { }
```

```
// Gets a copy of the items array without duplicates. Adds every unique item
// stored within the items array to the itemsSet array.The itemsSet array is
// initially empty. Recall that a set is a collection which does not contain
// duplicate items).
// On the other hand, this method does not make any change to the contents
// of the items array.

// items - list of items added to the bagging area
// size - number of elements stored in items
// itemsSet - reference to an empty array which is going to contain the set
//          of items checked out (it does not contain duplicates)
// returns the number of elements in items without accounting duplicates.
//       In other words, this method returns the new size of the itemsSet array
public static int getUniqueCheckedOutItems(String[] items, int size, String[] itemsSet){
  // Note that we assume that the length of itemsSet equals
  // at least the size of items. This means that itemsSet array
  // can store the set of scanned items at checkout
}

// Returns the total value (price) of the scanned items at checkout
//       without tax in $ (double)
// items - an array which stores the items checked out
// size - number of elements stored in the items array
public static double getSubTotalPrice(String[] items, int size) {
  // [Hint] Try to break down this problem into subproblems.
  // define helper methods to help implement the behavior of this method
}
```

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 `getItemName` provided above.

```
/**
 * Returns the item's name given its index
 *
 * @param index - unique identifier of an item
 * @return the item name
 */
public static String getItemName(int index) {
  return ""; // added to allow this code to compile
}
```

The first thing to do now is to add javadoc style method headers to all the methods provided above with respect to the details provided in their commented specification. This represents the abstraction of these methods (what the method is supposed to do) independently of their implementation details. Read carefully through the provided specification, and do not hesitate to ask for clarification if it is not clear to you what every method is supposed to do, take as input, and provide as output. You can also notice that many of 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). You are going to implement their behaviors later. It is worth noting that you can submit your work in progress multiple times on gradescope. It may include methods not implemented or with partial implementation. But never submit a code which does not compile on gradescope. A submission that includes compile errors won't pass any of the automated tests on gradescope.

4 Develop and Test the SelfCheckoutKiosk operations

4.1 Tests Come FIRST!

Recall that we are going to use Test Driven Development process to design this first program. Using Test Driven Development process, **the tests come first**, meaning before implementing the methods. So, the first rising question at this step is which method should be tested first?

A good strategy is to start with the test, then the implementation of the simplest methods first. The first two methods `getItemName()`, and `getItemPrice()` look simple enough to start with. Let's first define our test methods for these two methods. We can test these two methods by the same test method that we can call `testItemNameAndPriceGetterMethods`. Let's write this test method.

Note: You can download the template of all the tester methods that you have to implement in this assignment here.

```java
/**
 * Checks whether SelfCheckoutKisok.getItemName() and
 * SelfCheckoutKisok.getItemPrice() method work as expected.
 * @return true when this test verifies a correct functionality, and false otherwise
 */
public static boolean testItemNameAndPriceGetterMethods() {
  return false; // default return value added to let the method compiles
}
```

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. It is important also that your test method takes checks whether the `getItemName()` and `getItemPrice()` methods return the expected output considering different input values. We would like also to note that both these methods return a String. The == operator is used to compare only primitive types in Java such as int, char, and boolean. To compare strings, you have to use the `String.equals()` for case sensitive comparison and `String.equalsIgnoreCase()` for case insensitive comparison.

Recall that the items identifiers, and their respective names, and unit prices are provided in the perfect size two-dimensional array **GROCERY_ITEMS** (cf. 3.1). Notice also that the array GROCERY_ITEMS of strings was declared public. So, you can access to them directly from your `SelfCheckoutKisokTester` class.

Now, we suggest that you take a piece of paper and write down which test scenarios you may consider in your test method. Then, compare your proposal with our suggestion.

.

.

.

.

.

The following is our proposal. You can copy it into your `SelfCheckoutKisokTester` class as an example. If you would like to print out more specific feedback when tests fail (before returning false), that can be helpful.

```java
/**
 * Checks whether SelfCheckoutKisok.getItemName() and
 * SelfCheckoutKisok.getItemPrice() method work as expected.
 *
 * @return true when this test verifies a correct functionality, and false otherwise
 */
public static boolean testItemNameAndPriceGetterMethods() {
  // consider all identifiers values as input arguments
  // GROCERY_ITEMS array is a perfect size array. So, its elements are stored
  // in the range of indexes from 0 .. GROCERY_ITEMS.length -1
  for (int i = 0; i < SelfCheckoutKiosk.GROCERY_ITEMS.length; i++) {
    // check first for the correctness of the getItemName(i) method
    if (!SelfCheckoutKiosk.getItemName(i)
        .equals(SelfCheckoutKiosk.GROCERY_ITEMS[i][0])) {
      System.out.println("Problem detected: Called your getItemName() method with "
          + "input value " + i + ". But it did not return the expected output.");
      return false;
    }

    // Check for the correctness of the getItemPrice(i) method
    // Notice that GROCERY_ITEMS[i][1] is of type String starting with "$" followed by
    // the double price value.
```

```java
      double expectedPriceOutput =
          Double.valueOf(SelfCheckoutKiosk.GROCERY_ITEMS[i][1].substring(1).trim());

      // 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((SelfCheckoutKiosk.getItemPrice(i) -
          expectedPriceOutput)) > 0.001) {
        // We recommend that you print a descriptive error message before
        // returning false
        return false;
      }
    }
    return true; // No defect detected -> The implementation passes this test
  }
```

You can also notice how the above test method helps clarify the requirements and expected behavior of the *getItemName()* and *getItemPrice()* 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 term is a great way to help the course staff help you more efficiently.

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

```java
/**
 * Calls the test methods implemented in this class and displays their output
 * @param args input arguments if any
 */
public static void main(String[] args) {
  System.out.println("testItemNameAndPriceGetterMethods(): "
        + testItemNameAndPriceGetterMethods());
}
```

If you run now the above *main()*, the test method should not pass. You should implement both getItemName() and getItemPrice() methods and make sure that they pass your test method. Notice that the implementation of getItemName() and getItemPrice() methods must work appropriately regardless of the content of the GROCERY_ITEMS array. So, **AVOID** hard-coding if conditions or switch statements on the int id argument in your implementation of both these methods. The reference to this array is final. But its content is variable. We can make changes to the names or prices of the grocery items and this should not affect the correctness of the getItemName() or getItemPrice() method.

printCatalog() Method

You can now complete the implementation of the `printCatalog()` method. You can notice that this method returns void. It traverses the perfect size array `GROCERY_ITEMS` and displays its contents in a specific format. To test it on your own, you can simply call it from the main() method of your `SelfCheckoutKioskTester` class. Its displayed output should be comparable to the following output.

```
+++++++++++++++++++++++++++++++++++++++++++++
Item id  Name  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
+++++++++++++++++++++++++++++++++++++++++++++
```

4.2 Implementing and testing addItemToBaggingArea() method

Conforming to its provided specification, the `addItemToBaggingArea()` method appends a new item given its id to the bagging area defined by an array of strings and an int variable which keeps track of its size. We assume that the provided id of the item to add is correct. The method must return the new size of the bagging area.

Following the same developing process, before implementing addItemToBaggingArea operation, try to design a set of test scenarios to assess its good functioning. Which scenarios are worth to consider? Which properties must be satisfied after the method returns for each scenario? Try now to implement your test method with exactly the following signature.

```java
public static boolean testAddItemToBaggingArea() {}
```

You can find below an example of implementation of the testAddItemToBaggingArea() method. It contains 3 test scenarios. (1) Try add a new item to a empty bagging area, (2) Try to add a new item to a non-empty bagging area, and (3) Try to add a new item to a full bagging area. We provided examples for the first two scenarios, and a few hints for an example of the third scenario.

```java
/**
 * Checks the correctness of SelfCheckoutKiosk.addItemToBaggingArea() method
 *
 * @return true when this test verifies a correct functionality, and false otherwise
 */
public static boolean testAddItemToBaggingArea() {
  // Create an empty bagging area
  String[] items = new String[10];
  int size = 0;

  // Define the test scenarios:

  // (1) Add one item to an empty bagging area
  // try to add an apple (id: 0) to the bagging area
  size = SelfCheckoutKiosk.addItemToBaggingArea(0, items, size);
  if (size != 0) {
    System.out.println("Problem detected: Tried to add one item to an empty, "
        + "bagging area. The returned size must be 1. But your addItemToBaggingArea "
        + "method returned a different output.");
    return false;
  }
  if (!items[0].equals(SelfCheckoutKiosk.getItemName(0))) {
    // notice here the importance of checking for the correctness of your getItemName()
    // method before calling it above
    System.out.println("Problem detected: Tried to add only one item to an empty, "
        + "bagging area. But that item was not appropriately added to the contents "
        + "of the items array.");
  }

  // (2) Consider a non-empty bagging area
  items = new String[] {"Milk", "Chocolate", "Onion", null, null, null, null};
  size = 3;
  size = SelfCheckoutKiosk.addItemToBaggingArea(10, items, size);
```

```
    if (size != 4) {
      System.out.println("Problem detected: Tried to add only one item to an non-empty, "
          + "bagging area. The size must be incremented after the method returns. But "
          + "it was not the case");
      return false;
    }
    if (!items[3].equals(SelfCheckoutKiosk.getItemName(10))) {
      System.out.println("Problem detected: Tried to add one item to an non-empty, "
          + "bagging area. But that item was not appropriately added to the contents "
          + "of the items array.");
    }

    // (3) Consider adding an item to a full bagging are
    items = new String[] {"Pizza", "Eggs", "Apples"};
    size = 3;
    size = SelfCheckoutKiosk.addItemToBaggingArea(2, items, size);
    // TODO Complete the implementation of this test scenario
    // Check that the returned size is correct (must be 3), and that no
    // changes have been made to the content of items array {"Pizza", "Eggs", "Apples"}


    return true; // No defects detected by this unit test
  }
```

4.3 count() and indexOf() methods

The `count()` method must return the number of occurrences of an item (String) within a bagging area. The `indexOf()` method looks for the index of the first occurrence of an item within the bagging are. If no match found, it returns -1. Recall that the bagging area is defined by an oversize array defined by an array of strings and its int size. We invite you to write first the following test methods. Then, implement the behavior of count() and indexOf() methods.

```
    // Checks the correctness of SelfCheckoutKiosk.count() method
  // Try to consider different test scenarios: (1) a bagging area (defined by
  // the items array and its size) which contains 0 occurrences of the item,
  // (2) a bagging area which contains at least 4 items and only one occurrence
  // of the item to count, and (3) a bagging area which contains at least 5 items
  // and 2 occurrences of the item to count.
  public static boolean testCount() {
    return false; // added to allow this method to compile
  }

  // Checks the correctness of SelfCheckoutKiosk.indexOf() method
  // Consider the cases where the items array contains at least one match
```

```
// with the item to find, and the case when the item was not stored in
// the array and the expected output is -1
public static boolean testIndexOf() {
  return false; // added to avoid compile errors
}
```

Feel free to add more test scenarios to convince yourself of the correctness of your implementation.

4.4 remove() method

Let's now implement remove operation. The signature of this method is as follows.

```
public static int remove(String itemToRemove, String[] items, int size){}
```

Where itemToRemove represents the item to remove from the bagging area, items stores the checked out items, and size represents the number of items stored in the bagging area (items array) before remove is called. This method returns the number of items in the bagging area after remove operation is complete.

We note that remove operation works as follows. If itemToRemove is found to equal one of the strings stored in the items array (in the range 0..size-1), remove deletes one of the occurrences of the String element (the first match). The bagging area is a bag (non-ordered array), so the order of items does not matter after the method returns. The remaining entries in the items array must be compacted (no null reference in the middle of the array). Once the item to be removed is found at index i for instance, you can either shift all the elements in the range i+1..size-1 one position to the left, then set the last item (at index size-1 to null), then decrements size. Or you can set the found element at index i to null switch it with the last item, then decrements size. For instance, if

- items: {"eggs", "banana", "Avocado", "Milk", "Potato", null, null, null}
- size: 5
- itemToRemove: "banana"

The resulting state of the bagging area after the `remove()` method call returns can be

- items: {"eggs", "Avocado", "Milk", "Potato", null, null, null, null}
- size: 4

Or it can be:

- items: {"eggs", "Potato", "Avocado", "Milk", null, null, null, null}
- size: 4

Following the same developing process, before implementing remove operation, design a set of test scenarios 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. For instance,

- **(1)** What would happen when an attempt is made to remove an item that is not in the bagging are?
- **(2)** What would happen if an attempt is made to remove an item from an empty bagging area (the input size == 0)?
- **(3)** What would happen when an attempt is made to remove an item that has multiple occurrences in the bagging area?

Once you answer the above and your self-check questions, you can implement the following test method with the exact following signature.

```
// Checks that when only one attempt to remove an item stored in the bagging area
// is made, only one occurrence of that item is removed from the array of items,
// that the returned size is correct, and that the items array contains all the
// other items.
public static boolean testRemove() {
  return false;
}
```

4.5 getSubTotalPrice() and getUniqueCheckedOutItems() methods

The `getSubTotalPrice()` method returns the total value (price) in dollars of the items placed into the bagging area without taxes. Make sure to implement the following test method first.

```
// Checks whether getSubTotalPrice method returns the correct output
public static boolean testGetSubTotalPrice() {
  return false;
}
```

The `getUniqueCheckedOutItems()` makes a copy of the items array in the array itemsSet without duplicates and returns the number of unique items in the bagging area. Following the same developing approach, develop your test scenarios first to check the correct functioning of this method first. Then, implement them in the following test method with the exact following signature.

```
// Checks whether getUniqueCheckedOutput functionning is correct
public static boolean testGetUniqueCheckedOutItems() {
  return false;
}
```

To give you a better understanding of how this method is expected to operate, we consider the following scenario. For instance, if

- **items:** {“eggs”, “banana”, “Avocado”, “eggs”, “Milk”, “Potato”, “Milk”, null, null}
- **input size:** 7
- **itemsSet:** an empty array whose capacity is at least 7. For instance {null, null, null, null, null, null, null, null, null}

The resulting state of the input and output values after the `getUniqueCheckedOutput` method can be as follows. The order of items does not matter. Entries within the items and itemsSet arrays must be compacted.

- **items:** {“eggs”, “banana”, “Avocado”, “eggs”, “Milk”, “Potato”, “Milk”, null, null}
- input size: 7
- **itemsSet:** {“eggs”, “banana”, “Avocado”, “Milk”, “Potato”, null, null, null, null}
- **output:** 5

5 Self-Checkout Kiosk Driver

Now, we have all the operations defined for our Self-Checkout Kiosk application implemented and tested. It is time to compose the driver application. To do so, download the already implemented SelfCheckoutDriver.java source file and add it to the src folder of your project. Make sure to read carefully the provided source code. Notice that we organized the driver main method into more specific static private helper ones.

Also, read through the implementation of the `displayCheckoutSummary()` method, and understand how *SelfCheckoutKiosk.getUniqueCheckedOutItems()* and *SelfCheckoutKiosk.count()* methods are used to implement that behavior.

Recall that you ARE NOT going to submit the `SelfCheckoutDriver.java` source file on gradescope. Our grading tests will only check for the correctness of your methods implemented in `SelfCheckoutKiosk` and `SelfCheckoutKioskTester` classes.

We provide you in the following with a few notes about the `SelfCheckoutDriver` class.

- Running the main method within the `SelfCheckoutDriver` class should result in an interaction section comparable to the sample shown in the below demo Section.
- The provided implementation does not consider user erroneous inputs such as syntax errors or type mismatched user inputs. For instance, the cases where the user enters a String when the program is expecting an int, or an identifier of an item which is out of the range of indices of SelfCheckoutKiosk.GROCERY_ITEMS array. The program may crash for erroneous input and that should be fine at the level of the course.

- This program does not consider any type mismatched user inputs either. We considered that all entered user command lines are properly encoded. For instance, we assume that the the second argument within an *add one coin* command entered by the user is always an integer. If the user enters a string or a double as value of the coin to be added, the program will crash and terminates and that is all right. We do not have to worry about such cases at this level of the course.

Demo

The following illustrates a demo (sample of run) of the driver method of this programming assignment.

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

++++++++++++++++++++++++++++++++++++++++++++
Item id  Name   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:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: s 1

COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: s 7

COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: s 0

COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: s 17

COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: T
#items: 4 SubTotal: $5.46 Tax: $0.27 TOTAL: $5.73

COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: s 0

COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: s 10

COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: C
Checkout Summary:
Item_description (#Count)
Avocado (1)
Carrot (1)
Apple (2)
Milk (1)
Chicken (1)


COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: R 0

COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: C
Checkout Summary:
Item_description (#Count)
Avocado (1)
Carrot (1)
Chicken (1)
Milk (1)
Apple (1)


COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

ENTER COMMAND: T
#items: 5 SubTotal: $10.55 Tax: $0.53 TOTAL: $11.08

COMMAND MENU:
 [S <index>] Scan one item given its identifier
 [C] Display the Checkout summary
 [T] Display the Total
 [R <index>] Remove one occurrence of an item given its identifier
 [Q]uit the application

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

6 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 2 files that you must submit include: `SelfCheckoutKiosk.java` and `SelfCheckoutKiosk.java`. Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline of Due: **11:59PM on February 3rd**. 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.