Homework 5

GWU CSCI 1112 - Spring 2019

November 3, 2019

1 Introduction

This homework is intended to reinforce your understanding of working with lists in general and to explore implementations of lists using both arrays and linked lists. Your task is to implement and test a list that supports a shopping cart application using the provided framework.

1.1 Deadline

October 30, 2019 at 11:59pm

1.2 Submission

You must create a `.zip` file containing all of the files that you develop and work with and you must submit only the `.zip` file to blackboard before the deadline.

Your `.zip` file must be named using the following naming convention: `<yourNetId>-hw-05.zip`
For example, my NetId is `jrt`. If I submitted the homework, I would name my `.zip` file: `jrt-hw-05.zip`

Do not submit the compiled `.class` files. You must submit any `.java` files that support the application.

1.3 Grading Rubric

- 30% For successful compilation
- 15% For sufficient comments (file headers, function headers, line comments)
- 10% For consistent coding style
- 25% For base implementation
- 10% For extension implementations
- 10% For unit testing

1.4 Unit Testing

In this homework, you will take responsibility for unit testing your implementation and you must implement your own unit test file modeled on those provided in the previous homeworks. Your unit tests will be evaluated in terms of comprehensiveness and correctness. We are also withholding a set of unit tests that will be used to independently evaluate your implementation.

Your class implementations must therefore fulfill the prescribed interfaces so that our unit tests can use your classes. You may add additional functions as needed; however, for each new function you add, you should provide a unit test and evaluate it in your test application. You must unit test all of the prescribed interfaces.

1.5 Comments

As warned, the comments in the framework are now your responsibility. You must provide a file header for any files that you submit that documents the author and what is defined in the file. You must also document all functions that you implement with a function block header which must include information on what the function does, what the inputs to the function represent, and what the function returns. Finally, you must also provide relevant line comments inside any functions.

1.6 Plagarism

We will use a set of automated tools specifically designed to analyze code for plagarism. If you copy code from another source, classmate or website, there is a very high probability that these tools will flag your work as plagarized. You are permitted to discuss the problems at a high level; however, you must code your own solution. If you do not share code or outright borrow code from a website, you will have no problem with the plagarism filter.

2 Shopping Cart

Website based markets commonly model a physical shopping cart common to box-store retailers in order to collect items as a customer shops and then to checkout those items in one operation once the customer has decided to purchase. This model is used by virtually all web markets; however, the details of how these virtual shopping carts are implemented are opaque to the customer and are probably implemented by programmers using a variety of approaches. The ubiquity of the virtual shopping cart has made it a target for patent trolls.

A situated shopping cart is easy to visualize. It is a basket in which a customer places items that they are considering for purchase. Items on the shelf are instances of products and a customer is free to transfer one or more of those product instances from the shelf to their shopping cart. They may also choose to remove one or more items from their shopping cart at any time. Once a customer has decided that they are finished shopping, they proceed to checkout where a total of all the items is calculated (including any discounts).

We will develop our own virtual shopping carts. One of them uses a singly linked-list with only a head pointer to maintain items that a customer has selected. The other one will be based on an array based list you create. When an item is inserted into the shopping cart, it is inserted at the end of the list. Items can be removed from anywhere in the list by specifing the product that they wish to remove. We will also provide a number of methods to interact with the virtual shopping cart that will require iteration over the list including a checkout function to emulate the process of calculating the total cost of all items in the shopping cart.

2.1 Framework

You will note that a number of classes have been provided as a framework: `Product`, `ListItem`, `List`, `ShoppingCart`, `InventoryHelper`, and `ShoppingCartUnitTests`. You will primarily focus on implementing methods in the `List` class for the base implementation and to the `ShoppingCart` class with some slight modifications to the `List` class for the extension.

In general, our principle goal will be to implement a `List` class that can be supported by either an array or a linked-list. The abstract idea of a list will be the core data structure used to maintain the features of a virtual shopping cart. When products are added to the cart, they are added to a list. When products are removed from the cart, they are removed from a list. There are also a number of other interfaces for our list that are detailed below.

As noted, a shopping cart is basically a list. To reflect this relationship, you will find that the `ShoppingCart` class consists primarily of a reference to a `List` and contains many methods that simply pass through interactions to the list object. However, a shopping cart is also more than just a list, so there are also additional

methods dedicated to the `ShoppingCart` class that operate in novel ways on the list. Before we can worry about higher shopping cart functions, we need to first implement and validate our list implementation, so the Base implementation will focus on implementing and testing the list itself. The higher shopping cart functions will be addressed in the Extension.

The other class that you will need to develop in detail is the `ShoppingCartUnitTests` class which will be used to validate methods implemented in the `ShoppingCart` class. Unlike previous homework, there are no unit tests provided in this file; however, there are very basic examples of using the `ShoppingCart` class to create an array based cart and a linked-list based cart. In Section 3 of this document, you will find a suggested prescription for unit testing.

A number of other classes have been provided and they are generally comprehensive enough that you do not need to edit them for this assignment. Instead, you should focus on using these classes as is. The `Product` class encapsulates data associated with products that may be selected from the store and placed into the cart. The `InventoryHelper` class is provided to aid you in testing and provides two static methods, `getSingleProduct` and `getMultipleProducts` which return a single product and an array of products respectively. The `ListItem` class acts as a container for `Product` instances so it can be used as a node in a linked-list or as a element in an array, and it also allows us to add other list related information to an entry without modifying the `Product` class itself.

2.2 Organization

You will note that the files are included in a `base` directory. Your base implementation must use the files in the `base` folder. Once you have completed, fully unit tested, and debugged the base implementation, copy all of the files in the `base` directory into a directory at the same level named `extension`. All of your extension implementation is to be completed in the `extension` directory. You will submit a zip file that contains both the `base` and `extension` directories.

2.3 Base

You must implement the following `List` methods using the following function signatures:

```
public void add(Product product)
public boolean remove(Product product)
public void clear()
public boolean isEmpty()
public int length()
public Product get(int i)
public String toString()
```

2.3.1 List Type

Given that you are expected to implement this class using both an array based list and a linked-list, you might be wondering how this will be possible with only these interfaces. The provided member variables and the constructor implementation may help clarify this little detail:

```
public class List {

    public enum Type {
        ARRAY,
        LINKEDLIST
    }

    private final Type type;
    private ListItem head;
    private ListItem[] array;
```

```
    private int count;

    public List(Type type) {
        this.type = type;
        count = 0;

        if(type == Type.ARRAY) {
            array = new ListItem[2];
        } else if(type == Type.LINKEDLIST) {
            head = null;
        }
    }
...
}
```

The `enum Type` declaration and constructor parameter forces a user of the `List` class to pass in one of the two values, `Type.LINKEDLIST` or `Type.ARRAY`, when constructing a `List` object. Whatever value was passed in is stored internally in the `type` variable which is immutable, so you can always check to see what type of list this is inside any of the class methods. For example the `clear` method looks like this:

```
    public void clear() {
        if(type == Type.ARRAY) {
            array = new ListItem[2];
        } else if(type == Type.LINKEDLIST) {
            head = null;
        }
        count = 0;
    }
```

This structure allows us to sequester any array dedicated code in the `type == Type.ARRAY` branch and any linked-list dedicated code in the `type== Type.LINKEDLIST` branch. You will probably need to follow this model in all methods inside the `List` class.

2.3.2 Base List Requirements

For the base implementation, your array and linked-list must meet the following requirements.

An empty array will have two buckets allocated by default. The number of elements in the array is tracked by the `count` variable while the size of the array can be retrived by querying the `length` property associated with Java arrays. If the array is full and needs to grow, the size of the new array should be twice the size of the previous array.

The base linked-list will only have a `head` pointer. Therefore, in order to insert an item into the list, you must traverse the entire list. In the extension, you will add a `tail` pointer which will significantly improve insertion time.

2.3.3 Base Methods

This section gives a general explanation of the requirements for each function that must be implemented to satisfy the base requirements.

`public void add(Product product)`

`addProduct` inserts the specified product at the end of the list. Note that the same product can be added to the list multiple times.

`public boolean remove(Product product)`

`removeProduct` searches through the list for a matching product. If the product is found, remove the product from the list and return `true`. If the product is not found, return `false`.

```
public void clear()
```

`clear` removes all items from the list. This can be accomplished a variety of ways, some are faster than others.

```
public boolean isEmpty()
```

`isEmpty` returns `true` if there are no items in the list. If there are items in the list, `isEmpty` returns `false`.

```
public int length()
```

`length` returns the total number of items in the list.

```
public Product get(int i)
```

`get` accesses the item at the index `i` and returns that item. If the index is not in the legal range of values for the list, `get` returns `null`.

```
public String toString()
```

`toString` builds a string of information about the list. This string is primarily for debugging purposes, so it should include information such as list type, length, and the names of the products as they are ordered in the list. There is other information that you might return such as the amount of allocated space for an array-based list.

2.3.4 Extension

Before beginning any work on an extension, get the base implementation completed and tested. Remember that the extension is a copy of your base implementation and any detailed debugging you do of the base implementation in the extension will not be evaluated. Therefore, you should bulletproof your base implementation before beginning work on the extension.

Each extension must be validated with unit tests and associated extension unit tests will be considered part of the credit for a given extension.

Extension - Add a `tail` pointer to `List`

In the `List` class add a `tail` pointer and refactor all linked-list based logic where appropriate to use the `tail` pointer. This may seem to be rather simple, but there are a number of important changes that must be made to handle all edge cases.

Extension - Add quantities to `List`

For this extension, add a quantity field to the shopping cart. Keep in mind that the `Product` is a template item, like a showroom or catalog item, and should not have a quantity. Instead, the quantity should be stored in the list itself. *Hint*: `ListItem` is probably a good place to add this field.

The interfaces provided for `add` in both the `List` class and the `ShoppingCart` class must be modified to the following:

```
public void add(Product product, int quantity)
```

When a product is added to the `List`, the list must be checked for the existence of a matching product. If a matching product is found, the quantity should be accumulated rather than adding a new product. If a matching product is not found, the product is added with the associated quantity.

Extension - Apply discounts to `List`

For this extension, add a discount field to the shopping cart. Again, keep in mind that the `Product` is a template item, like a showroom or catalog item, and should not have a quantity. Instead, the quantity should be stored in the list itself. *Hint*: `ListItem` is probably a good place to add this field.

The interfaces provided for `applyDiscount` in both the `List` class and the `ShoppingCart` class must be:

```
public boolean applyDiscount(Product product, double discount)
```

When a product is initially added to the `List`, the discount is zero by default. When a discount is applied, the list must be checked for the existence of a matching product. If a matching product is found, the discount is applied to all matching products and true is returned. If a matching product is not found, false is returned. The discount must also be clamped in the range [0.0,1.0] so that a discount can never inflate the price and a discount can never be more than 100% off.

Extension - Compute subtotals in `ShoppingCart`

For this extension, implement the subtotal feature in the `ShoppingCart` class. The `subtotal` method must have the following signature:

```
public double subtotal()
```

`subtotal` computes the total cost of all items in the cart including any discounts and returns the calculated subtotal. If the cart is empty, `subtotal` returns zero. The value returned by `subtotal` can never be negative.

Extension - Checkout a `ShoppingCart`

For this extension, implement the checkout feature in the `ShoppingCart` class. The `checkout` method must have the following signature:

```
public double checkout(double salestax)
```

`checkout` computes the subtotal of all items in the cart, applies a sales tax rate to the subtotal, computes the final total, clears the cart, and returns the computed total. If the cart is empty, `checkout` returns zero. The `salestax` parameter can never be negative and must be a unfactored real number (not a percentage). The value returned by `checkout` can never be negative.

3 Unit Tests

It is a difficult task to validate software; however, untested software is risk for developers and users alike. Untested software can exhibit errors that can negatively affect reputations, financing, and even lives. We try to mitigate the possibility of errors through unit testing.

Unit testing is a challenge and requires a comprehensive and methodological approach. As a student, you need to practice these skills; however, it is easy to get overwhelmed and stuck by the process.

To help you analyze the approach, the following is a prescription for the Base implementation that you might follow for unit testing. Keep in mind that this is a rough outline and you may discover that some tests have been overlooked. Try to follow the logic of this analysis and see if you can identify where this analysis has made too large a leap from on test to another and fill in the gaps. Also, you should consider why some of

these tests are repeated multiple times.

For example why is there an explicit set of tests to add products back to a cart once it has been emptied? Adding items was already tested, so why is this test different than the initial adding test? If you can answer these questions correctly, you will have developed an important understanding of how assumptions change as the state of data structure changes.

1. Test - constructing a new Cart of both types
   (a) Validate that both carts are empty
   (b) Validate the length of both carts is zero
2. Test - adding a set of products to both carts
   (a) Validate that both carts are not empty
   (b) Validate the length of both carts is the expected length
3. Test - getting all products that were added to both carts
   (a) Validate that products are in the expected positions in both carts
4. Test - removing one product from both carts
   (a) Validate that both carts are not empty
   (b) Validate the length of both carts is the expected length
   (c) Validate that products are in the expected positions in both carts
5. Test - removing all products from both carts
   (a) Validate that both carts are empty
   (b) Validate the length of both carts is zero
6. Test - adding products back into a cart previously emptied
   (a) Validate that both carts are not empty
   (b) Validate the length of both carts is the expected length