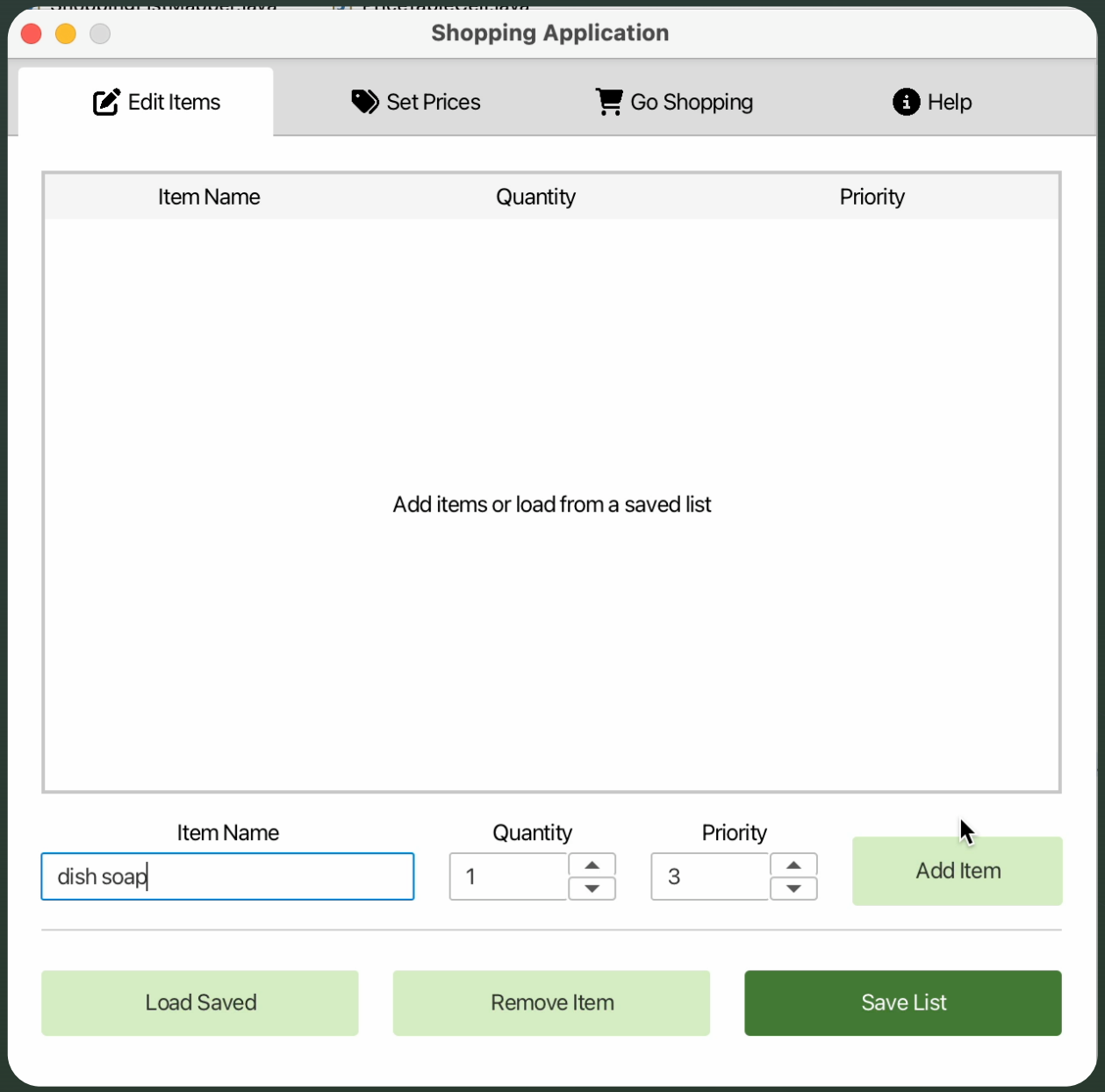
Anshita Khare

# SHOPPING APP V2



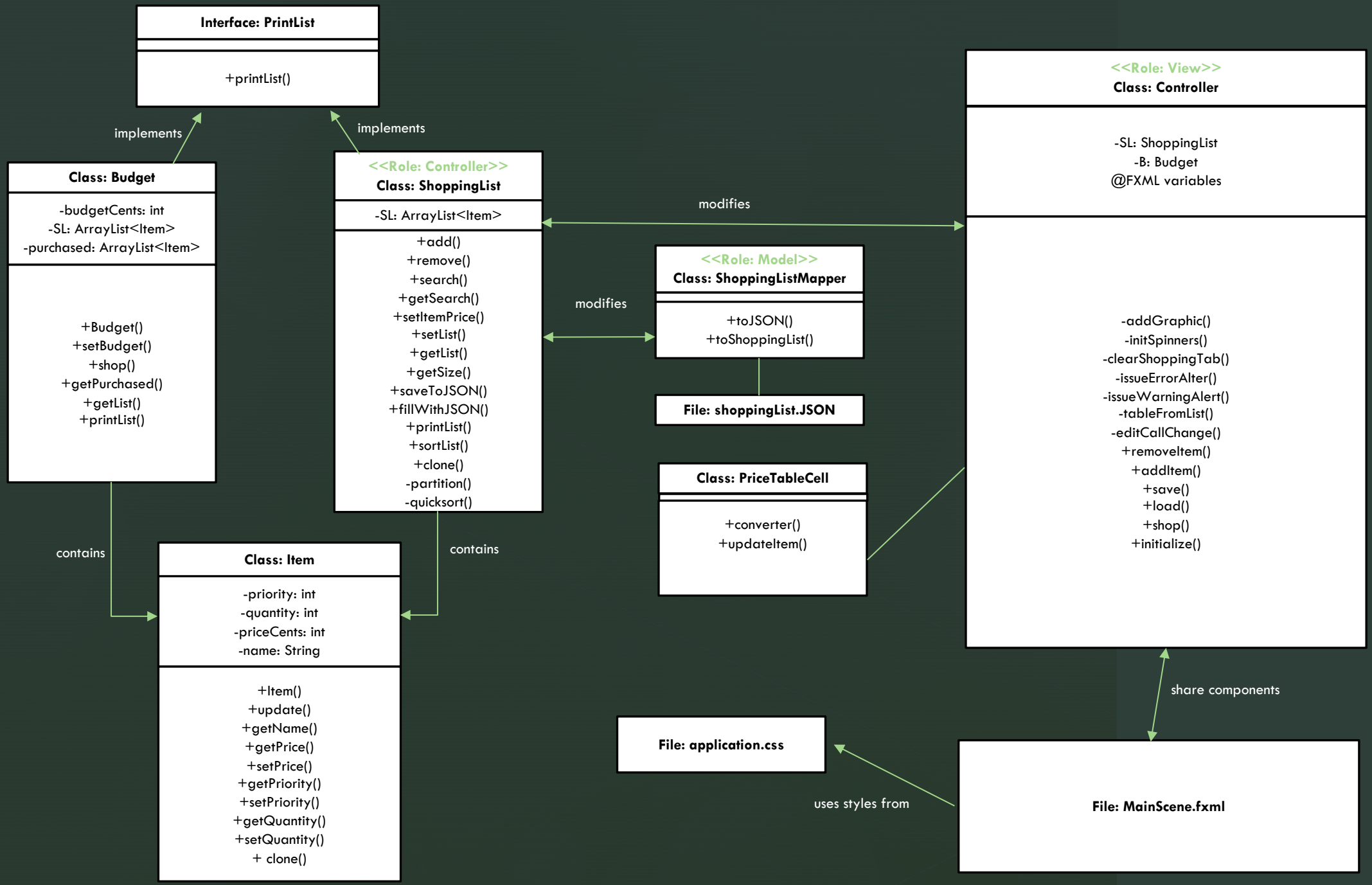

# Demo

- The tabs model an object and altering information in any given tab affects the others
- Each tab contains dynamically created tables to allow users to view and modify data
- Keyboard accessible!

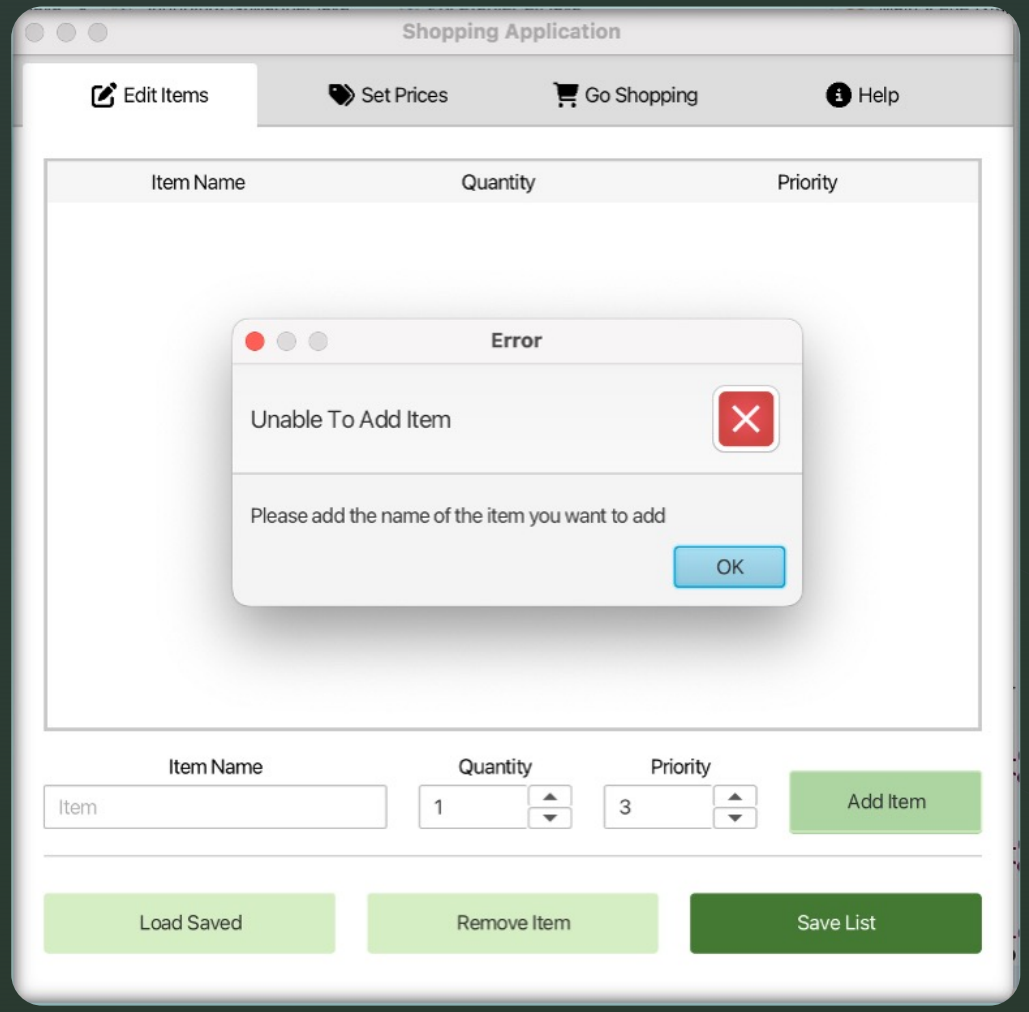
# Functionality

- Edit Items Tab
  - Add item by item name, quantity(1-100), a priority(1-5, with 1 being the highest priority)
  - Modify the quantity and priority of any displayed item by editing the cell
  - Remove selected item
  - Save list to .json file named shoppingList
  - Load saved shopping list from shoppingList.json
- Set Prices Tab
  - Determine the price of each item in the shopping list by dollar and cents by double clicking the price cells for each item in the table
    - The input model rejects non-numeric characters  and appropriately converts excess cents to dollars when needed
  - Save prices to shopping list and write that to .json file
- Go Shopping Tab
  - Determine the shopping budget by dollars and cents and go shopping by priority
    - Try to purchase as many of high priority item(s) as possible if buying all is out of the budget and continue
  - Display purchased items
  - Display remaining items
  - Display remaining budget
  - Allows users option to choose update their shopping list and save the remaining items

# UML Diagram

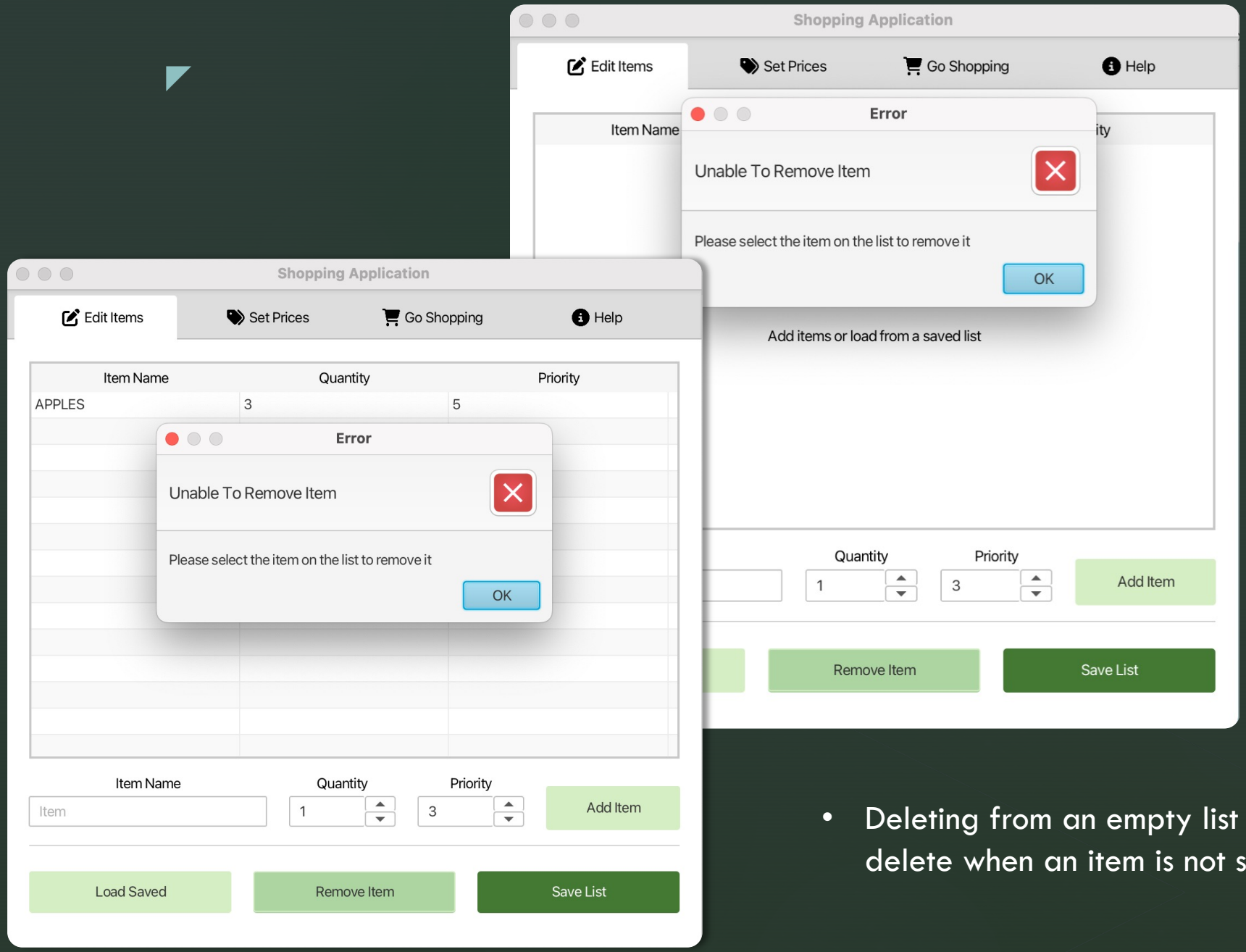

Interface: PrintList
+printList()
implements
implements
Class: Budget
-budgetCents: int
-SL: ArrayList<Item>
-purchased: ArrayList<Item>
+Budget()
+setBudget()
+shop()
+getPurchased()
+getList()
+printList()
<<Role: Controller>>
Class: ShoppingList
-SL: ArrayList<Item>
+add()
+remove()
+search()
+getSearch()
+setItemPrice()
+setList()
+getList()
+getSize()
+saveToJSON()
+fillWithJSON()
+printList()
+sortList()
+clone()
-partition()
-quicksort()
modifies
modifies
<<Role: Model>>
Class: ShoppingListMapper
+toJSON()
+toShoppingList()
File: shoppingList.JSON
Class: PriceTableCell
+converter()
+updateItem()
<<Role: View>>
Class: Controller
-SL: ShoppingList
-B: Budget
@FXML variables
-addGraphic()
-initSpinners()
-clearShoppingTab()
-issueErrorAlter()
-issueWarningAlert()
-tableFromList()
-editCallChange()
+removeItem()
+addItem()
+save()
+load()
+shop()
+initialize()
contains
contains
Class: Item
-priority: int
-quantity: int
-priceCents: int
-name: String
+Item()
+update()
+getName()
+getPrice()
+setPrice()
+getPriority()
+setPriority()
+getQuantity()
+setQuantity()
+ clone()
share components
File: application.css
uses styles from
File: MainScene.fxml

# Test Cases




- Adding unpopulated item

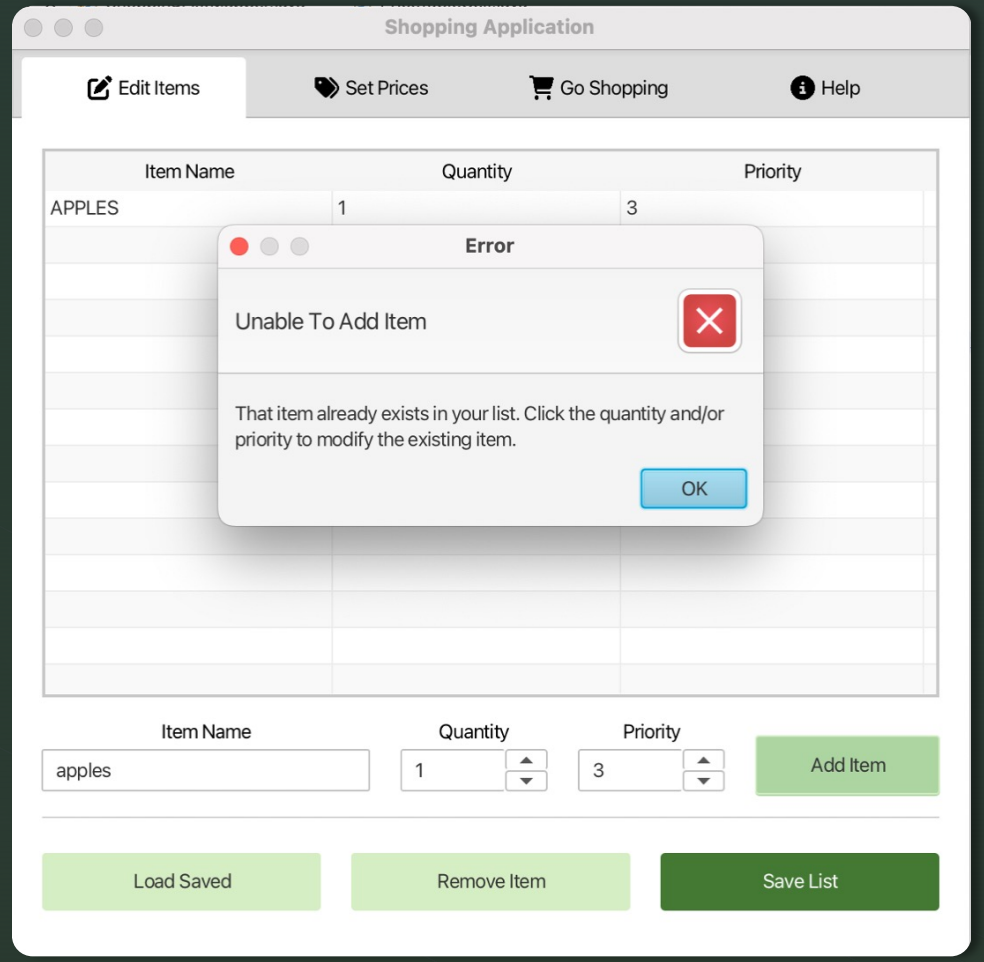
# Test Cases



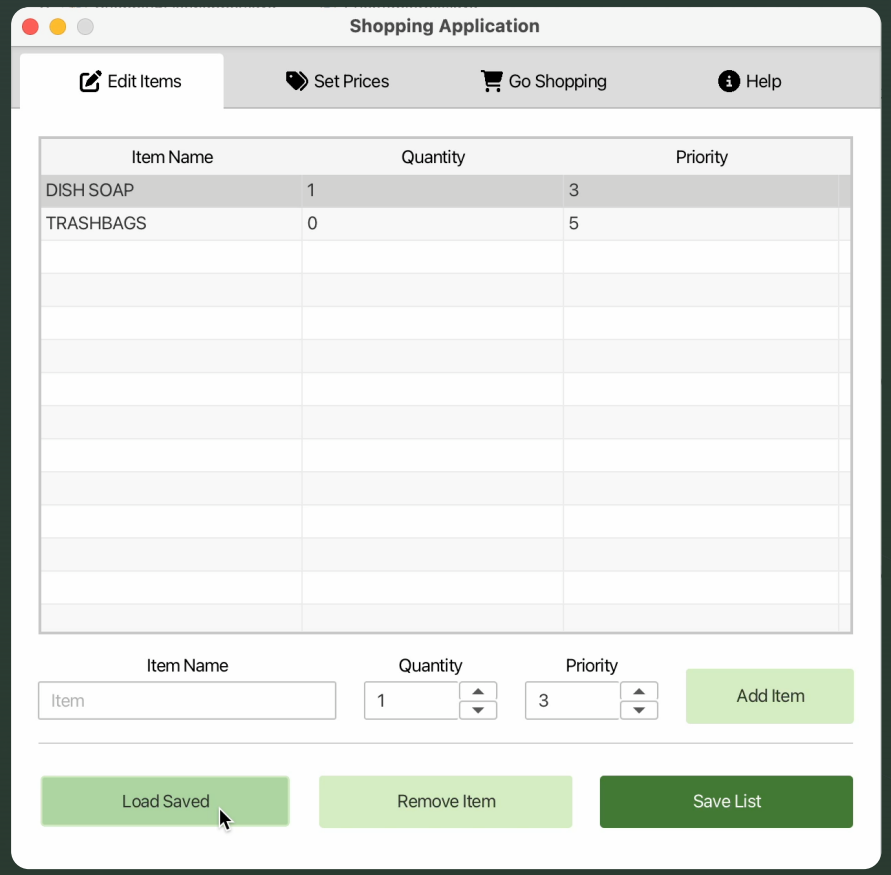

- Deleting from an empty list or trying to delete when an item is not selected

# Test Cases

- Adding Item that already exists

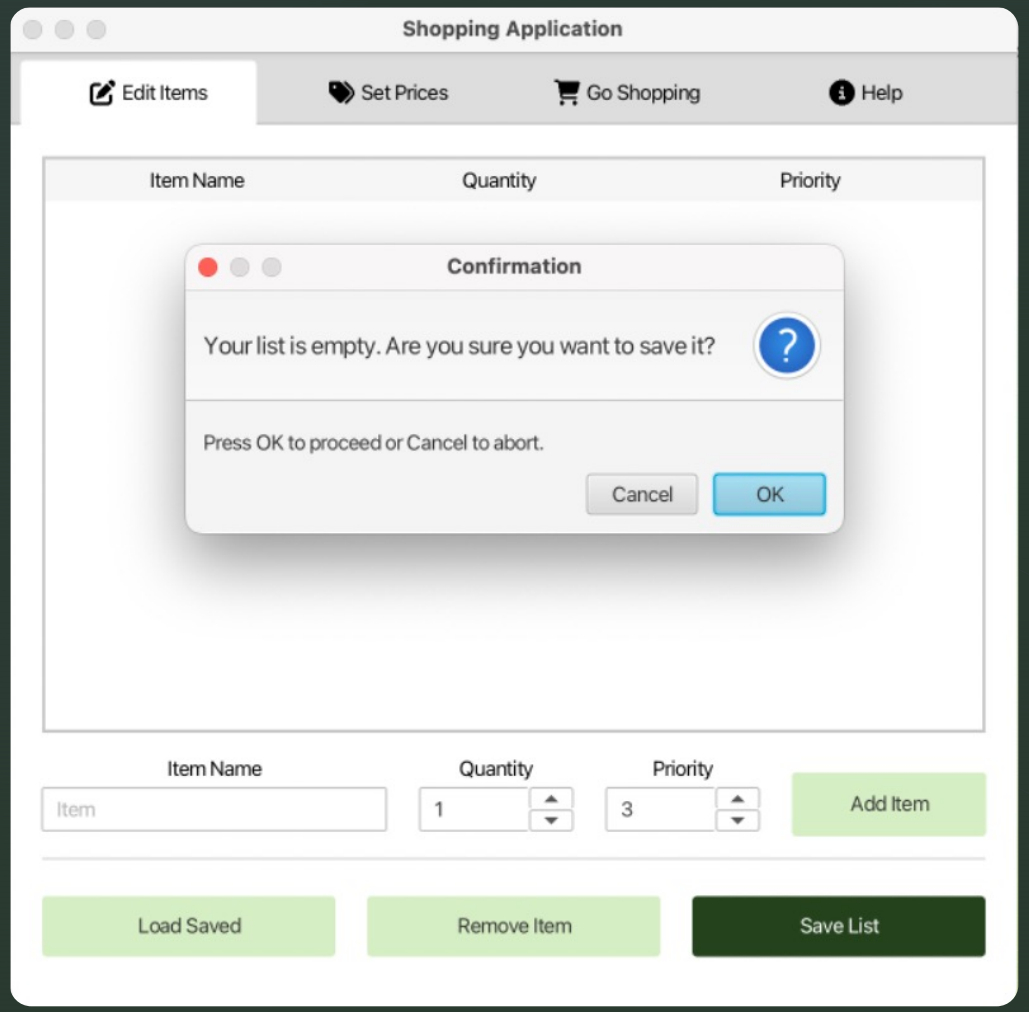


# Test Cases




- When loading multiple times or loading after starting new shopping list, previous information must not be visible after new load & warning must be issued before loading

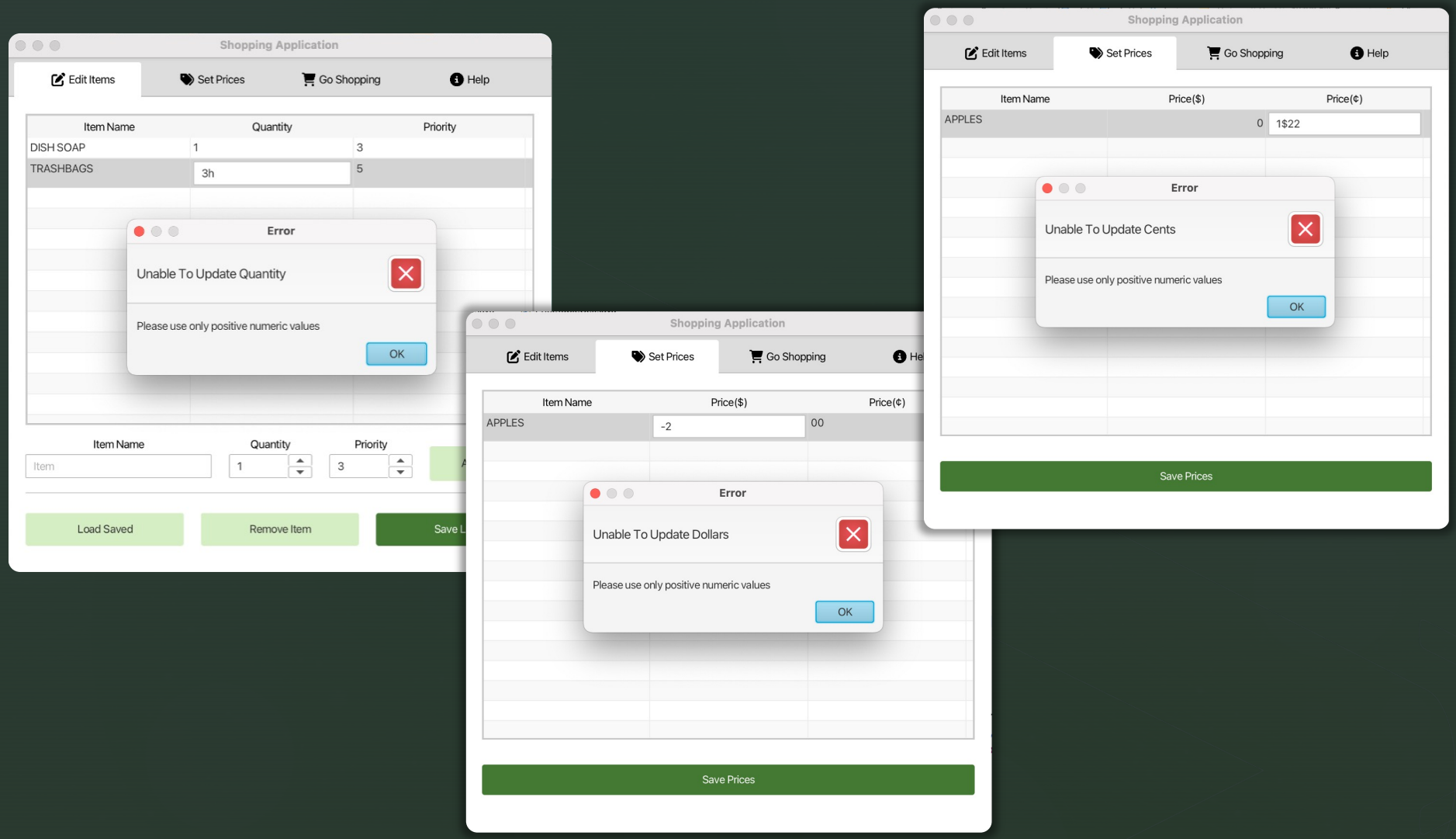
# Test Cases



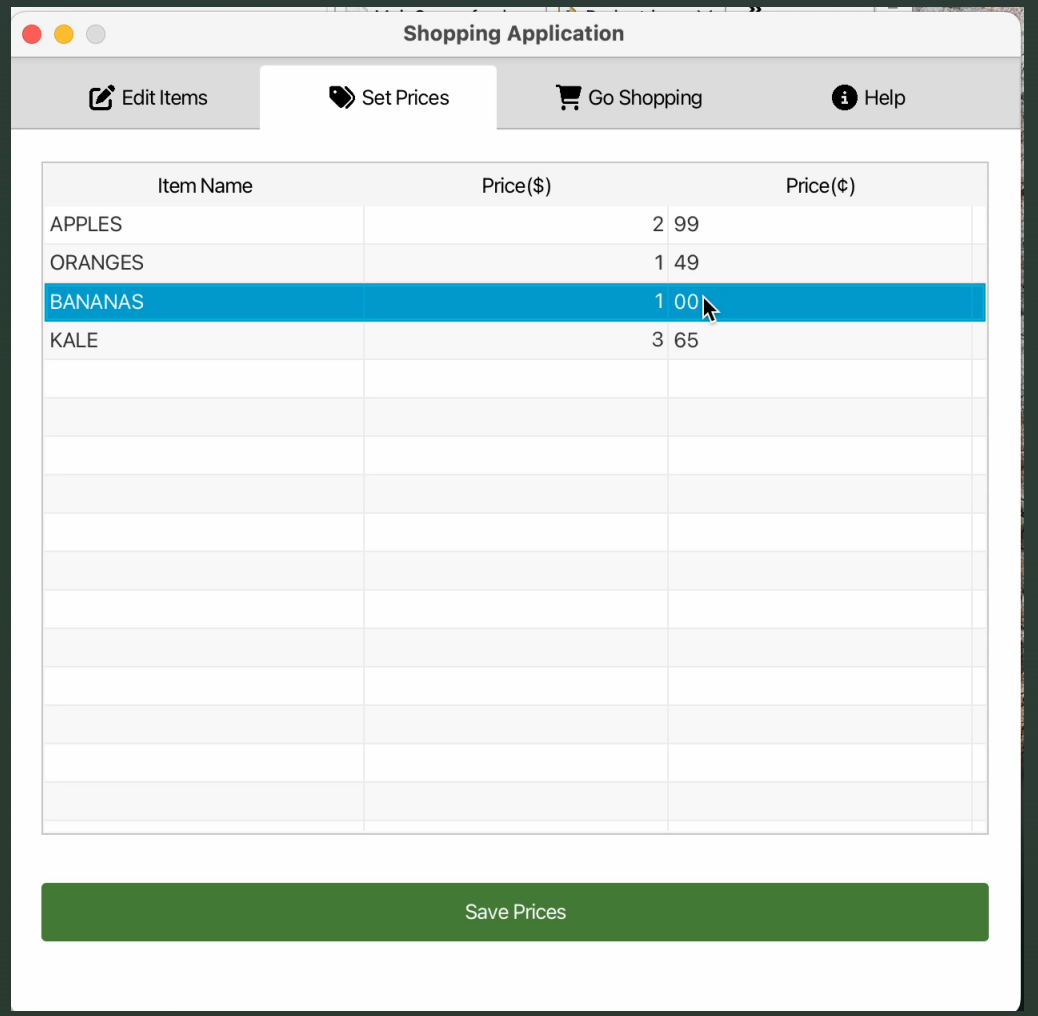

- Saving an empty list warning

# Test Cases

- Set quantity/priority/price/budget with invalid character(abc!#~) or negative number



- Overflow cents input are reformatted as dollars
  - Eg: Cents = 257 → $2 57



# Algorithms

**1.) Sort the items in the list by order of priority**

Assumptions: 1 is highest priority.

The following algorithm will put the items with the highest priority value at the end of the list. This is because the list will be consumed from the bottom up by the Budget's shop function later.

```
public void sortList() {
//calls internal quicksort method
}
/** A function that partitions the given array list for quick sort
* @param currList the sublist that the function works with
* @param left the index of the first item in the sublist
* @param right the index of the last item in the sublist*/
private int parition(ArrayList<Item> currList, int left, int right) {
//The function creates two variables called biggestIndex and smallestIndex and sets them to left and right, respectively.
//Creates a variable called temp to temporarily hold an Item object during the swapping process.

//The function sets the pivot variable to the priority value of the Item object that is at the midpoint of the sublist represented by left and right.

//The function enters a loop that will continue until biggestIndex is greater than smallestIndex.
        //Checks whether the priority value of the Item object at biggestIndex is greater than the pivot value.

                //If it is, the function increments biggestIndex until it reaches an Item object with a priority value less than or equal to pivot.

        //Checks whether the priority value of the Item object at smallestIndex is less than the pivot value.
                //If it is, the function decrements smallestIndex until it reaches an Item object with a priority value greater than or equal to pivot.

        //If biggestIndex is less than or equal to smallestIndex, the function swaps the Item objects at biggestIndex and smallestIndex in currList, updates biggestIndex and
        smallestIndex, and continues the loop.

// By the time the loop is finished, the function has moved all Item objects with priority values less than or equal to pivot to the left side of the sublist and all Item objects with
priority values greater than pivot to the right side of the sublist.

//The function returns the index of the last item that has a priority value less than or equal to pivot that can be used to further sort the list
}
/**A recursive quick sort method
* @param currList the list or sublist that is being sorted @param lowIndex the index of the first item in the sublist  @param highIndex the index of the last item in the sublist*/
private void quickSort(ArrayList<Item> currList, int lowIndex, int highIndex) {
//Calls a partition function to select a pivot element around which the list is divided
//Recursively sorts the two resulting sub-lists, the one to the left of the pivot and the one to the right, by calling itself again for each side }
```

# Algorithms

1) Go shopping, purchase as many items on the list by priority with the given budget; Purchase the highest priority items first then go to the lower priority items and purchase them as needed, minimizing the amount of money left over.
    a. Assumptions: The list given to the below function has already been sorted and the items with the greatest priority are at the bottom of the list

```
/**A method that allows for items within the budget to be purchased*/
public void shop(){
//create an empty list called purchased

//starting from the last item in a list called SL and working backwards so that deletions do not change the index of subsequent items
    //if the budget is ever 0, break since we cannot buy any more items
    //get the details of the item at the current index i in SL(name, quantity, priority, price)

//calculate the total cost of buying the entire quantity of the item at its price.
    //if the budget is greater than or equal to the total cost, buy all of the item's quantity,
    //subtract the total cost from the budget, and remove the item from SL.

//if the budget is less than the total cost but greater than or equal to the item's price
    //buy as much of the item as possible while the budget is not exhausted by repeatedly subtracting the item's price from the budget until the budget can no longer afford the item and updating the quantity of the item purchased accordingly

//if you ended up purchasing something, create a new item with the same name, priority,
and price as the original item but with the quantity set to the quantity purchased,
and add it to the purchased list.
}
```

# Future Additions

- Display total price per item (taking quantity into account)
- Login functionality
- Allow user option to manually sort display sorted