Lab 11: Interfaces, arrays, arrays, arrays…

Let's get started!

Today we'd like you to:

1. **Open Eclipse**. Remember to keep all your projects in the same workspace. Think of a workspace as a shelf, and each project is a binder of stuff on the shelf. If you need some help organizing your workspace, we'd be happy to help!
2. **Create a new Java project**. Call it something like COMP 1510 Lab 11.
3. **Complete the following tasks**. Remember you can right-click the src package in the Eclipse package explorer project to quickly create a new Java class or interface.
4. When you have completed the exercises, **show them to your lab instructor**. Be prepared to answer some questions about your code and the choices you made.

What will you DO in this lab?

In this lab, you will:

1. create and implement an interface that 'locks' an object
2. declare, instantiate, and use an array of objects
3. reverse the contents of an array, possibly using a helper swap method
4. process command line arguments
5. explore method design and overloading

Table of Contents

1. More Java interfaces

We've looked at a few interfaces these last few weeks: EventHandler, Iterator, Iterable, and Comparable. We've learned that an interface is a great way to specify a collection of public behaviours that defines how to interact with some kind of object. Let's create one of our own interfaces and play with it:

1. **Create a Java interface called Lockable**. The interface should contain the following public methods:
   i. setKey(int key) establishes the key used to unlock
   ii. boolean lock(int key) locks the implementing object if the correct key is provided and returns true if the object was locked
   iii. boolean unlock(int key) unlocks the implementing object if the correct key is provided and returns true if the object was unlocked
   iv. boolean locked() returns true if the implementing object is locked.
2. A class that implements Lockable can be used to instantiate objects whose regular methods are protected. If the object is locked, the methods cannot be invoked. If the object is unlocked, the methods can be invoked.
3. Copy the Coin class from lab 9 to lab 11. **Modify the Coin class so it implements Lockable**. A locked coin cannot be flipped.
4. **Write a Driver file that contains a main method**. The Driver file should demonstrate how a locked Coin cannot be used.

2. Reversing an array

Write a program called **ReverseArray** that prompts the user for an integer, then asks the user to enter that many values. Store these values in an array and print the array. Then reverse the array elements so that the first element becomes the last element, the second element becomes the second to last element, and so on, with the old last element now first. **Do not just reverse the order in which they are printed; you must change the way they are stored in the array**. Do not create a second array; just rearrange the elements within the array you have. (Hint: Swap elements that need to change places using a helper method called swap.) When the elements have been reversed, print the array again.

3. A shopping cart

Let's write a program that simulates part of a point of sale (cash register) system:

1. Create a class called **Item**:
   i. Item stores three pieces of data
      1. name of the item (a String)
      2. price of the item (a double)
      3. the quantity of the item being purchases (an int).
   ii. Item contains an **overloaded constructor**:
      1. The first version accepts a parameter for each instance variable
      2. The second version accepts a parameter for the name and price, and sets the quantity being purchased to 1

iii. Item contains accessors for the values. Since there are no mutators, we can treat Item as **immutable**. What keyword should we add to the instance variable declarations that reminds us that our class remains immutable?
iv. Item requires a toString method.

2. Create a class called **Transaction**:
    v. Transaction stores three pieces of data:
        1. an array of Item (perhaps called cart)
        2. totalPrice (a double representing the total price of the Items in the array)
        3. itemCount (an integer that represents the number of Item objects in the array)
    vi. Transaction requires a **constructor** which accepts one parameter:
        1. an integer used to initialize the array of Item to the specified size
        2. The constructor should also set the other two instance variables to zero.
    vii. Create a method called **addToCart** which accepts three parameters, creates an Item with them, and adds the Item to the cart:
        1. String for an Item name
        2. double for an Item price
        3. integer for the quanitity of the item
    viii. Create an **overloaded version of the addToCart** method which accepts an Item object and adds the Item to the Cart.
    ix. Both version of addToCard should ensure that itemCount and totalPrice are correctly updated whenever an Item is added to the Transaction.
    x. Sometimes we buy too much and it won't fit in the cart. We need a way to increase the size of the cart in the Transaction. Create a method called public void **increaseSize()**:
        1. Before we add an Item to the cart, we need to check if the cart is full.
        2. If it is, we need to:
            a. Declare and instantiate a new local array of Item, copy the old Items to the new array, and then assign the new array to the cart instance variable. Ensure the new local array can carry THREE more items.
    xi. Create a method called **getTotalPrice()** which returns the total price of all the Items in the cart.
    xii. Create a method called **getCount()** which returns the total number of all the Items in the cart. (Don't confuse this with itemCount which returns the number of different kinds of things. We want the total number of things).
    xiii. Create a toString method that returns a String containing the list of Items in the cart: name, price, and quantity, followed by the total price of the transaction.
3. Create a driver class called **Shopping**. In this driver, use a loop to ask users to buy things. Show us that both overloaded addToCart methods work.

4. Averaging numbers (command line arguments)

As discussed in Section 8.4 of the text book, when you run a Java program called Foo, anything typed on the command line after “java Foo” is passed to the main method in the args parameter as an array of Strings.

Write a program called Average that just prints the Strings passed to the program via the command line. Print the command line arguments one per line. If no command line arguments are provided, print “No arguments”.

Modify your program:

1. Assume the arguments given at the command line are integers.
2. If there are no arguments, print the same "No arguments" message.
3. If at least one command line argument is provided, compute and print the average of the arguments as a double. Note that you will need to use the static parseInt method of the Integer class to extract integer values from the Strings that are passed in.
4. If any non-integer values are passed in, your program should produce an error, which is unavoidable at this point. (Can you prevent your program from crashing if this happens?)

Test your program thoroughly using different numbers of command line arguments

5. You’re done! Show your lab instructor your work.