

## Midterm Exam

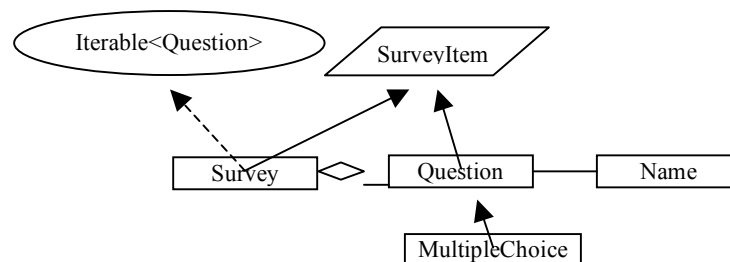
Name: Solutions

Student ID Number: \_\_\_\_\_

Signature: \_\_\_\_\_

**Directions:**

- This exam is closed book and closed notes. No electronic devices are permitted.
- Please check that you have *pages 1 through 7*. If you don't have enough room to answer a question, you can continue onto the back of page 7.
- Do your own work. No discussion or collaboration with other students is permitted.
- If a question seems ambiguous, write an explanation of how you interpreted the question.
- The exam is 83 minutes long (10:07 – 11:30am). Be sure to pace yourself.
- When you finish: *If fewer than 10 minutes remain, please do not turn in your exam early, since getting up may disturb other students who are trying to finish.*
- Use the following example as a guide for drawing class hierarchies. In this example, the **class** Survey and the **class** Question both **extend** the **abstract class** SurveyItem. Also, Survey **implements** the **interface** Iterable<Question> and each survey **has a collection** of Questions. Every Question has a Name. MultipleChoice is a subclass of (i.e., **extends**) Question.



## Background

Recall that the `java.util` package contains support for a variety of abstract data types and their implementations. For example, the **List** interface has **ArrayList** and **LinkedList** implementations, both of which inherit functionality from **AbstractList**.

In addition, the `java.util` package contains a class named **Observable** with the following API. As stated in the Java documentation, “This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed. An observable object can have one or more observers. An observer may be any object that implements interface `Observer`. After an observable instance changes, an application calling the `Observable`'s `notifyObservers` method causes all of its observers to be notified of the change by a call to their `update` method.”

Constructor Summary	
<code>Observable()</code>	Construct an <code>Observable</code> with zero <code>Observers</code> .

Method Summary	
void	<code>addObserver(Observer o)</code> Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
protected void	<code>clearChanged()</code> Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the <code>hasChanged</code> method will now return <code>false</code> .
int	<code>countObservers()</code> Returns the number of observers of this <code>Observable</code> object.
void	<code>deleteObserver(Observer o)</code> Deletes an observer from the set of observers of this object.
void	<code>deleteObservers()</code> Clears the observer list so that this object no longer has any observers.
boolean	<code>hasChanged()</code> Tests if this object has changed.
void	<code>notifyObservers()</code> If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed.
void	<code>notifyObservers(Object arg)</code> If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed.
protected void	<code>setChanged()</code> Marks this <code>Observable</code> object as having been changed; the <code>hasChanged</code> method will now return <code>true</code> .

The Java documentation also states that “a class can implement the **Observer** interface when it wants to be informed of changes in observable objects.” The `Observer` interface has the following method. The first parameter is the `Observable` object that changed. The second parameter is whatever is passed to the `notifyObservers` method above. Throughout this exam, we’ll assume that the value of the second parameter is null.

Method Summary	
void	<code>update(Observable o, Object arg)</code> This method is called whenever the observed object is changed.

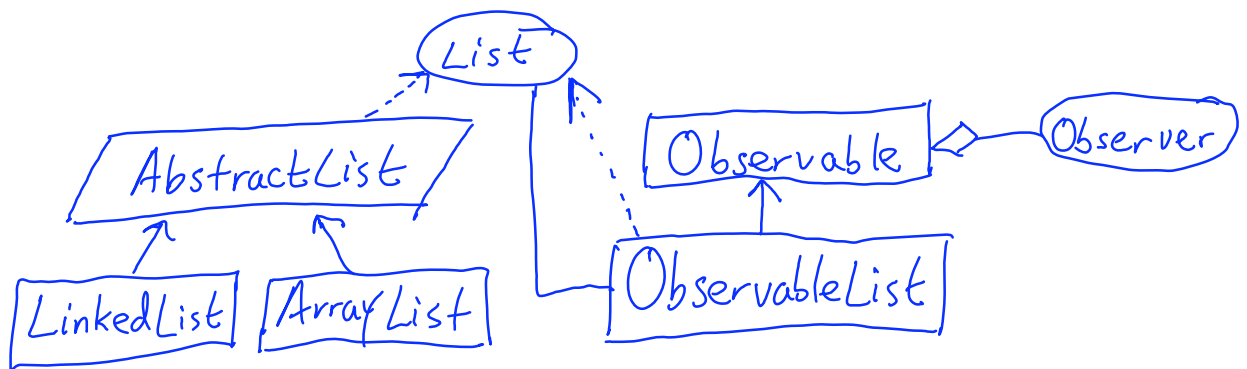
## Design Problem

Consider the following design problem carefully, since it forms the basis of the entire exam.

Suppose you want to create a class called **ObservableList<T>** with the following requirements:

- It has all the functionality of an Observable object.
- It can be used anywhere that a List<T> can be used.
- Its underlying representation can be any List<T> (for example, but not limited to, ArrayList or LinkedList).
- The underlying list exists (and may already contain elements) before the ObservableList constructor is called.
- The ObservableList notifies its observers whenever the list is modified. (Assume that a reference to the underlying list is not retained outside the ObservableList.)

1. (10 points) Using the example notation on the front cover as a guide, draw a class hierarchy showing your ObservableList in the context of the following: List, LinkedList, ArrayList, AbstractList, Observable, and Observer.



Explain why you positioned ObservableList where you did in the diagram.

Since it must have all the functionality of Observable, it extends Observable. Since it can substitute for a List, it implements List. It also has a list as its data.

2. (10 points) Complete the class header and the constructor for your ObservableList class. Also include declarations of any required instance variables.

```
public class ObservableList<T> extends Observable implements List {
    List<T> list;

    public ObservableList ( List<T> list ) {
        this.list = list;
    }
    // methods would be here
}
```

3. (10 points) Recall that every List has a size() method that returns the number of elements in the list. Also recall the requirement that an ObservableList can be used anywhere a List could be used. Provide a complete implementation of the size() method for your ObservableList class.

```
public int size() {
    return list.size();
}
```

4. (10 points) Recall the requirement that an ObservableList must notify its observers after the list is modified. Since every List must have an add method, provide a complete implementation of the add(T) method for your ObservableList class. (Note that the add method for a List always returns true.)

```
public boolean add(T x) {
    list.add(x);
    setChanged();
    notifyObservers();
    return true;
}
```

5. (10 points) Recall that the Iterator interface has the following methods.

Method Summary	
boolean	<u>hasNext()</u> Returns true if the iteration has more elements.
<u>E</u>	<u>next()</u> Returns the next element in the iteration.
void	<u>remove()</u> Removes from the underlying collection the last element returned by the iterator (optional operation).

Suppose that when using your ObservableList, you discover that observers aren't being notified when the list is modified by an iterator's remove method. Describe briefly in **English**, possibly using a diagram, how you would implement the ObservableList's **iterator()** method to solve this problem. Feel free to introduce a new class in your solution.

Create an iterator class that wraps the underlying list's iterator, delegating all methods & performing notification after remove.

6. (15 points) Based on the design you described in Problem 5, provide a complete implementation of the ObservableList `iterator()` method. If you need to define a new class, define it as a local class within this method.

```
public Iterator<T> iterator() {
```

```
    final Iterator<T> it = list.iterator();
```

```
    class NotifyingIterator implements Iterator<T> {
```

```
        public boolean hasNext() {  
            return it.hasNext();
```

```
        }
```

```
        public T next() {  
            return it.next();
```

```
        }
```

```
        public void remove() {  
            it.remove();
```

```
            setChanged();
```

```
            notifyObservers();
```

```
        }
```

```
    }
```

```
    return new NotifyingIterator();
```

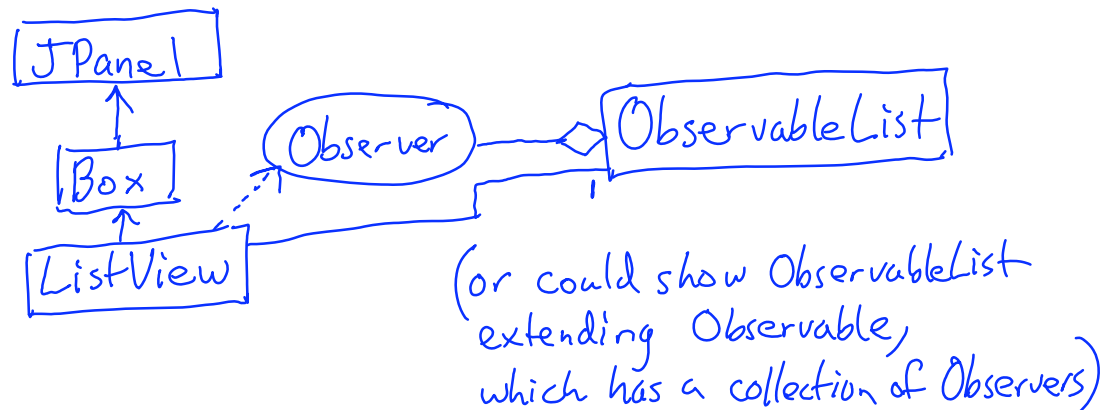
```
}
```

7. (5 points) The **Box** class extends JPanel. When you add things to a box, it displays them in either a single row (horizontally) or a single column (vertically), depending on a single parameter (BoxLayout.X\_AXIS or BoxLayout.Y\_AXIS) to the Box constructor.

Suppose you want to create a subclass of Box called **ListView<T>** with the following requirements:

- its data model is an ObservableList containing elements of type T,
- the data model is passed to the ListView constructor,
- the ListView displays the elements of the data model in a column, and
- the ListView updates whenever the contents of the list is changed, so it always shows exactly what is in the list.

Think about how you would design the ListView class. Then, using the notation on the front of the exam as a guide, draw a class hierarchy diagram showing the relationships among: **Box**, **JPanel**, **ObservableList**, **ListView**, and **Observer**.



8. (10 points) Anything added to a box must be a JComponent. Therefore, anyone using a **ListView<T>** from question 7 must provide a mechanism for creating a JComponent view for each data item of type T in the ObservableList.

One design option would be for the programmer to pass a “factory” object to the **ListView** constructor. The factory would provide a method to create a JComponent for any element of type T. Define a **ViewFactory** interface for this purpose.

```
public interface ViewFactory<T> {
    public JComponent createViewFor(T dataItem);
}
```

9. (5 points) As a concrete example of your interface in question 8, provide an implementation of **ViewFactory<String>** that creates a **JLabel** for each data item.

```
public class LabelMaker implements ViewFactory<String> {
    public JLabel createViewFor(String s) {
        return new JLabel(s);
    }
}
```

10. (5 points) Continuing from the questions 7-9, suppose that, inside your ListView class, the instance variable **factory** refers to the ViewFactory, and the instance variable **list** refers to the data model of type ObservableList. Write the constructor for the class ListView<T>.

```

public ListView( ObservableList<? extends T> list, ViewFactory<? super T> factory ) {
    super (BoxLayout.Y_AXIS);
    this.list = list;
    this.factory = factory;
    list.addObserver (this);
    update(list, null);
}

```

11. (10 points) The purpose of the **update()** method (as shown on the bottom of page 2), is to refresh the view after the list has been modified. Making the same assumptions as in problem 10, complete the implementation of the ListView update method.

```

public void update(Observable o, Object ignored) {
    removeAll(); // remove all the JComponents from this Box
    for (T dataItem : list)
        add (factory.createViewFor (dataItem));

    revalidate(); // forces the layout manager to run
    repaint(); // repaints ListView on the display
}

```

12. (5 points – extra credit) Suppose it turns out that an ObservableList is modified very frequently, so creating new JComponents for all the elements in the list every time would create a lot of garbage. Suggest a way to avoid recreating views for the same data elements every time the list is modified. (That is, how could you reuse the old ones, and only create new views for data items that are actually new to the list?) Answer in English or provide a revised implementation of update() to illustrate your idea.

Use a hash table (ideally WeakHashMap) to keep track of the view for each dataItem.

Inside the loop:

① if the dataItem isn't a key in the map yet, create a view for it & put them into the map

② get the view of the data item from the map & add it to this Box.

If you finish this exam early, check over your work. If fewer than 10 minutes remain, please do not turn in your exam early, since getting up may disturb other students who are trying to finish. Thanks.