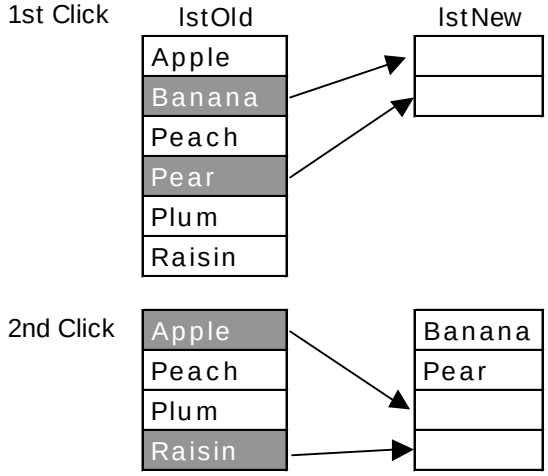
Please use separate paper for your answers. Question point values are shown in parentheses.

1. (18) In class we looked at code that moved selected items from one list to another and deleted the selected items from the original list. The code for this is given below:

```
void cmdMove_actionPerformed(java.awt.event.ActionEvent event)
{
        // first move the selected items to the new list
        int [] selected = lstOld.getSelectedIndices();
        if (selected.length == 0)
           return;
        for (int i=0; i<selected.length; i++)
        {
           int insertIndex = selected[i];
           String value = (String)lstOldModel.getElementAt(insertIndex);
           lstNewModel.addElement(value);
        }
        // now delete the selected items from the old list
        for (int i=selected.length-1; i>=0; i--)
        {
           int delIndex=selected[i];
           lstOldModel.removeElementAt(delIndex);
        }
}
```

This code placed the selected items into the new list in the same order that they were in the original list. Also, when the button was clicked a second or subsequent time, the selected items were placed **after** the items placed in the new list during previous moves. This behavior is summarized below:



The procedure above uses **two** loops – one to move selected items to the new list and another to remove the selected items from the old list. Rewrite the actionPerformed event so that it behaves exactly the same way but completes the actions in **one** loop. You might find the insertElementAt(value, insertPoint) method helpful.

2. (18) When talking about interfaces and inheritance, we looked at the Manageable interface and the KeyVector class. The code for these is shown below:

```
public interface Manageable
{
   String getKey();
}
```

```
import java.util.Vector;
import tManageable.Manageable;

public class KeyVector extends Vector
{
   public KeyVector()
   {
      super();
   }

   public Object getObjByKey(String x)
   {
      for (int i = 0; i < this.size(); i++)
      {
         if (((Manageable)this.elementAt(i)).getKey().equals(x))
            return this.elementAt(i);
      }
      return null;
   }
}
```

When we wanted to store an object in a KeyVector, we needed to be sure that the object's class implemented the Manageable interface as shown below:

```
public abstract class Employee implements Manageable
```

Explain why classes like the Employee class were required to implement the Manageable interface and why the "Manageable" cast was required in getObjByKey method of the KeyVector class.

3. (15) Assume you have a Vector object that has been populated with some String objects. Later, some of the elements are set equal to null (to indicate that they have been deleted). Thus, you may have something like the following:

```
Vector v = new Vector();
v.addElement("One");
v.addElement("Two");
v.addElement("Three");
v.addElement("Four");
v.addElement("Five");
// later the following code occurs
v.setElementAt(null, 1);
v.setElementAt(null, 3);
```

Now you want to go through all elements of the Vector and differentiate between the elements that contain strings and those that are null. You write the following code:

```
for (int i = 0; i < v.size(); i++)
{
        if (v.elementAt(i).equals(null) )
        {
                System.out.println(i + " null");
        }
        else
        {
                System.out.println(i + " " + v.elementAt(i));
        }
}
```

You then discover that the code above fails to do what you want. (It compiles but fails to run to completion.)

Why does the code above fail and how would you fix it?

4. (18) You have an abstract Customer class that has two subclasses called Retail and Wholesale. Both of the subclasses need to store a String fEmpNo and String fEmpName instance variable. In addition, the Wholesale class needs to store a double fDiscountRate instance variable. All instance variables should not be visible outside the class where they are defined.

   The fEmpNo and fEmpName values should be set when an object is instantiated. In addition, there should be "get-type" accessor method for all instance variables and "set-type" accessor method for fEmpName and fDiscountRate.

   There should be an instance method named computeDiscountedPrice(double price). For the Wholesale class, this method should return the discounted price as a double using the formula:

   price * (1 – fDiscountRate)

   For the Retail class, this method should just return the price (basically doing nothing at this time). The method should be able to be called polymorphically from a customer object reference regardless of whether a Retail or Wholesale object is being processed.

   Given this information, create java class definitions for the three classes.

5. (16) Consider the following code:

```
String orig = txtOriginal.getText();
String newString = MyStringUtil.fullTrim(orig);
lblFullTrim.setText(newString);
```

   The class method fullTrim() eliminates all leading and trailing spaces from the string variable orig. It also reduces 2 or more consecutive spaces within the string to a single space. Thus, if orig equaled " a b c ", the fullTrim() method would return "a b c".

   Write the MyStringUtil class and within this class, the fullTrim() class method.

6. (15) In class we looked at an example where our Employee class could store references to a number of Dependent class objects. We also saw a method of the Employee class that gave us access to these Dependent objects. The relevant code follows:

```
public abstract class Employee
{
   // Instance variables
   private String fEmpNo;
   private String fName;
   private Vector fDependent;
     …
   public Vector getAllDependents()
   {
      return fDependent;
   }
```

We later concluded that the getAllDependents() method violated an important principle of object-oriented design.

a. What principle was violated?
b. Explain how this method violated the principle.
c. How could the method be changed so that it does not violate the principle?

Sample Answers:

1. The code below is one solution. Instead of going through the old list's selected items top to bottom and adding them to the end of the new list, this solution goes through the old list's selected items bottom to top and inserts elements at the previous end of the new list.

```
void cmdMove_actionPerformed(java.awt.event.ActionEvent event)
{
        // get the selected list of indices
        int [] selectedIndices = lstOld.getSelectedIndices();
        if (selectedIndices.length == 0)
           return;
        // find where to start inserting in new list
        int insertPoint = lstNewModel.size();
        // starting at bottom of old list, move element to new list
        // and then delele element from old list
        for (int i=selectedIndices.length-1; i>=0; i--)
        {
           int selIndex=selectedIndices[i];
           String value = (String)lstOldModel.getElementAt(selIndex);
           lstNewModel.insertElementAt(value, insertPoint);
           lstOldModel.removeElementAt(selIndex);
        }
}
```

2. Any class that is going to be stored in a KeyVector must implement a getKey() method that returns a String. By implementing the Manageable interface, the class is obligated to provide a non-abstract version of this method. In addition, the "cast" to a Manageable type would fail in the getObjByKey() method if the class of the object being managed did not implement this interface.

   The "cast" in the getObjByKey() method is necessary in order to ensure that the getKey() method is understood by the compiler. The type returned by a vector's elementAt() method is an Object and the Object class does not know anything about a getKey() method.

3. The code fails because the statement:

```
if ( v.elementAt(i).equals(null) )
```

   attempts to call the equals() method on a null pointer (creating a null pointer exception). To solve the problem, compare the object reference in the vector's element against the null pointer using the "= =" operator as follows:

```
if ( v.elementAt(i) == null )
```

4.

```
public abstract class Customer
{
   private String fEmpNo;
   private String fEmpName;
   public Customer(String empNo, String name)
   {
     fEmpNo = empNo;
     fEmpName = name;
   }
   public String getEmpNo()
   {
     return fEmpNo;
   }
   public String getEmpName()
   {
     return fEmpName;
   }
   public void setEmpName(String name)
   {
     fEmpName = name;
   }
   public abstract double computeDiscountedPrice(double price);
}

public class Wholesale extends Customer
{
   private double fDiscountRate;
   public Wholesale(String empNo, String empName)
   {
      super(empNo,empName);
   }
   public void setDiscountRate(double rate)
   {
      fDiscountRate = rate;
   }
   public double getDiscountRate()
   {
      return fDiscountRate;
   }
   public  double computeDiscountedPrice(double price)
   {
      return price * (1 - fDiscountRate);
   }
}
```

```
public class Retail extends Customer
{
   public Retail(String empNo, String empName)
   {
      super(empNo,empName);
   }
   public  double computeDiscountedPrice(double price)
   {
      return price;
   }
}
```

5.

```
public abstract class MyStringUtil
{
   public static String fullTrim(String s)
   {
      // remove leading/trailing spaces with trim() method
      String t = s.trim();
      String answer = "";
      for (int i=0; i<t.length(); i++)
      {
         if (t.substring(i,i+1).equals(" ") &&
            t.substring(i+1,i+2).equals(" "))
         {
            // do nothing with character i – characters i and i+1 are both spaces
         }
         else // concatenate character i to the answer
         {
            answer = answer + t.substring(i,i+1);
         }
      }
      return answer;
   }
}
```

6. (a) This method violated the principle know as encapsulation. Encapsulation states that the internal details of the class should not be seen, known, or accessible outside the class. In particular, instance variables should not be accessible outside the class.

   (b) This method violated encapsulation because it returned an object reference to the private mutable instance variable fDependent outside the class. Note that fDependent is the private instance variable, not the Dependent objects that it points to.

   (c) The way to solve this problem is to return a clone of the fDependent instance variable – not the original variable. The clone may be used to reference the individual Dependent objects (fulfilling the objective of the method). However, the clone does not provide any ability to modify an Employee object's relationship to the dependents associated with it (the Employee class should have its own set of methods to do that).