CSU213 Exam 2 – Spring 2007

Name: ______________________________

Student Id (last 4 digits): ______________________________

Homework login name: ______________________________

Instructor's Name: ______________________________

- Write down the answers in the space provided.
- You may use any valid Java code. We provide the APIs for any library classes and their methods that you may need. If you need a method and you don't know whether it is provided, define it.
- Remember that the phrase "develop a class" or "develop a method" means more than just providing a definition. It means to design them according to the design recipe. You are *not* required to provide a method template unless the problem specifically asks for one. However, be prepared to struggle if you choose to skip the template step.
- We will not answer *any* questions during the exam.

Good luck.

Problem	Points	/
1		/13
2		/13
3		/20
4		/19
Total		/65

Reference page

- For all classes assume that the `ISame` interface has been implemented automatically and a `TestHarness` that uses `same` for comparison is available.
- Write all tests as invocation of the `test` method in the `Examples` class:

```
test("name of the test", actual, expected)
```

- If the problem consumes a `Traversal` and it is not known how the `Traversal` is implemented, define the example data as:

```
Traversal<Data> tr = '(data1, data2, ...);
```

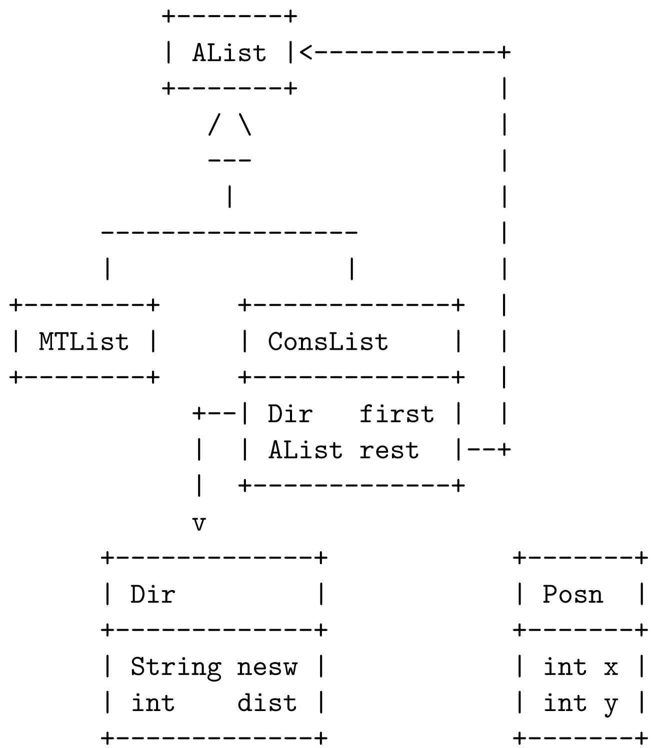
- In all problems you can omit the visibility modifiers, unless it is
  - required by the interface you are implementing
  - required by the the class you are extending
  - the problem explicitly asks you to do so
- Throughout the exam you may use the clasess `AList<T>`, `ConsList<T>` and `MTList<T>` to represent a collection of data. You may also assume that the methods that implement the `Traversal<T>` interface have been defined for this class hierarchy.
- Recall that the `Traversal<T>` interface is defined as follows:

```
public Traversal<T> getRest()
    throws IllegalUseOfTraversalException;
public T getFirst()
    throws IllegalUseOfTraversalException;
public boolean isEmpty();
```

13 Points

Problem 1

The following class diagram shows classes that represent a sequence of routing directions:



Each instance of `Dir` represents a travel in the given direction for the given distance.

The `String nesw` has to be one of the following: `"N"`, `"E"`, `"S"`, `"W"`.

The list of `Dir` has to satisfy the following property:

> Any `"N"` or `"S"` must be followed by `"E"` or `"W"`.
>
> Any `"E"` or `"W"` must be followed by `"N"` or `"S"`.

1. Make examples of data for these classes.

Solution

[POINTS 3: 1 point Dir, 1 point lists, 1 point if bad one is there too]

```
 Dir n8 = new Dir("N", 8);
 Dir e5 = new Dir("E", 5);
 Dir s2 = new Dir("S", 2);
 Dir w1 = new Dir("W", 1);
 Dir s1 = new Dir("S", 1);

/** ok route that moves (4, 5) from the start */
 AList route = new ConsList(this.n8,
     new ConsList(this.e5,
         new ConsList(this.s2,
             new ConsList(this.w1,
                 new ConsList(this.s1,
                     new MTList()))))));

 /** bad route - E is followed by W */
 AList badrouteEW = new ConsList(this.n8,
     new ConsList(this.e5,
             new ConsList(this.w1,
                 new ConsList(this.s1,
                     new MTList())))));

 /** bad route - N is followed by S */
 AList badrouteNS = new ConsList(this.n8,
     new ConsList(this.s2,
             new ConsList(this.w1, new MTList())));

 /** bad route - W is followed by E */
 AList badrouteWE = new ConsList(this.n8,
     new ConsList(this.w1,
             new ConsList(this.e5, new MTList())));

 /** bad route - S is followed by N */
 AList badrouteSN = new ConsList(this.e5,
```

```
new ConsList(this.s2,
        new ConsList(this.n8, new MTList())));
```

2. Design the method `okRoute` for the `Alist` of `Dir`. For each method (e.g. helper method as well as the original one) you may either provide a compete test suite or provide at least three examples/tests together with a short explanation of what other test cases are needed to complete the test suite.

Solution

[POINTS 5: 3 points for the methods, 2 points for examples: examples in Dir as well as List classes; methods: purpose statements, bodies, in all classes]

```
// AList class: -------------------------------------------
 /**
   * Is this a good route - making only allowed transitions?
   * @return
   */
  public boolean goodRoute();

  /**
   * Was the move from the previous direction to this routing OK?
   * @param previous the previous direction order
   * @return true if we went from NS to EW, or from EW to NS
   */
  public boolean goodMoveFrom(Dir previous);

// MTList class: -------------------------------------------
 /**
   * Is this a good route - making only allowed transitions?
   * @return
   */
  public boolean goodRoute(){
    return true;
  }

 /**
   * Was the move from the previous direction to this routing OK?
   * @param previous the previous direction order
```

```
   * @return true if we went from NS to EW, or from EW to NS
   */
  public boolean goodMoveFrom(Dir previous){
    return true;
  }

// ConsList class: -------------------------------------------
  /**
   * Is this a good route - making only allowed transitions?
   * @return true is all turns are NS --> EW or EW --> NS
   */
  public boolean goodRoute(){
    return this.rest.goodMoveFrom(this.first) &&
           this.rest.goodRoute();
  }

  /**
   * Was the move from the previous direction to this routing OK?
   * @param previous the previous direction order
   * @return true if we went from NS to EW, or from EW to NS
   */
  public boolean goodMoveFrom(Dir previous){
    return previous.OKnext(this.first);
  }

// Dir class: -------------------------------------------
  /**
   * is the given direction Ok after this one?
   * @param that next direction order
   * @return OK if NS after EW or if EW after NS
   */
  public boolean OKnext(Dir that){
    if (this.nesw.equals("N") || (this.nesw.equals("S")))
      return (that.nesw.equals("E") || (that.nesw.equals("W")));
    else
      return (that.nesw.equals("N") || (that.nesw.equals("S")));
  }

// Examples class: -------------------------------------------
public void testOKnext(){
```

```
  System.out.println("test OK next N --> E: " + n8.OKnext(e5));
  System.out.println("test OK next N --> W: " + n8.OKnext(w1));
  System.out.println("test OK next S --> E: " + s2.OKnext(e5));
  System.out.println("test OK next S --> W: " + s2.OKnext(w1));
  System.out.println("test OK next E --> N: " + e5.OKnext(n8));
  System.out.println("test OK next E --> S: " + e5.OKnext(s2));
  System.out.println("test OK next W --> N: " + w1.OKnext(n8));
  System.out.println("test OK next W --> S: " + w1.OKnext(s2));

  System.out.println("test bad EW route: " + !e5.OKnext(w1));
  System.out.println("test bad NS route: " + !n8.OKnext(s2));
  System.out.println("test bad WE route: " + !w1.OKnext(e5));
  System.out.println("test bad SN route: " + !s2.OKnext(n8));
}

public void testGoodRoute(){
  System.out.println("test OK route: " + route.goodRoute());
  System.out.println("test bad EW route: " + !badrouteEW.goodRoute());
  System.out.println("test bad NS route: " + !badrouteNS.goodRoute());
  System.out.println("test bad WE route: " + !badrouteWE.goodRoute());
  System.out.println("test bad SN route: " + !badrouteSN.goodRoute());
}
```

3. Design the method `destination` for the `AList` (of `Dir`) that consumes a `Posn origin` and produces a `Posn` that represents the destination after one has followed the given routing. You may again substitute a description of the needed tests for a complete test suite. However, you must include at least three tests for each method, and at least one route in the test suite must consist of at least five segments and cover all four directions.

Solution

[POINTS 5: 3 points for the methods, 2 points for examples: examples in Dir as well as List classes; methods: purpose statements, bodies, in all classes]

```
// AList class: -----------------------------------------------
 /**
   * Compute the coordinate for the destination
   * for this route, given the starting point
   *
   */
  public Posn destination(Posn origin);

// MTList class: -----------------------------------------------
  /**
   * Compute the coordinate for the destination
   * for this route, given the starting point
   *
   */
  public Posn destination(Posn origin){
    return origin;
  }

// ConsList class: -----------------------------------------------
  /**
   * Compute the coordinate for the destination
   * for this route, given the starting point
   *
   * @param origin the location of the satrt of this trip
```

```
   * @return the location at the end of this trip
   */
  public Posn destination(Posn origin){
    return this.rest.destination(this.first.destination(origin));
  }

// Dir class: -------------------------------------------------
 /**
   * produce the poit to which we get following this direction
   * from the origin
   *
   * @param origin the point of origin
   * @return the destination point
   */
  public Posn destination(Posn origin){
    if (this.nesw == "N")
      return (new Posn(origin.x, origin.y + this.dist));
    else if (this.nesw == "S")
      return (new Posn(origin.x, origin.y - this.dist));
    else if (this.nesw == "E")
      return (new Posn(origin.x + this.dist, origin.y));
    else if (this.nesw == "W")
      return (new Posn(origin.x - this.dist, origin.y));
    else return origin;
  }

// class Examples:  -------------------------------------------------
 public void testDestinationDir(){
    Posn origin = new Posn(0, 0);

    System.out.println("test destination: " + n8.destination(origin)
        + " expected: " + new Posn(0, 8));
    System.out.println("test destination: " + e5.destination(origin)
        + " expected: " + new Posn(5, 0));
    System.out.println("test destination: " + s2.destination(origin)
        + " expected: " + new Posn(0, -2));
    System.out.println("test destination: " + w1.destination(origin)
        + " expected: " + new Posn(-1, 0));
  }
```

```
public void testDestination(){
   System.out.println("test OK route: " + route.destination(new Posn(0, 0))
       + "\nexpected: " + (new Posn(4, 5)));
 }
```

13 Points

Problem 2

You are given two Traversal<T> traversals that generate sorted collections of data, with the data sorted according to the given Comparator<T>.

1. Design the method merge in the Algorithms class that consumes the two Traversal<T> objects and an object of the type Comparator<T> and produces an ArrayList<T> of all items from both collections, sorted according to the given Comparator<T>.

   For example, if:

   tr1 traverses over: "Good Day" "Hello" "So Long"
   tr2 traverses over: "Bye" "Ciao" "Hi"

   the resulting ArrayList<T> will contain "Bye" "Ciao" "Good Day" "Hello" "Hi" "So Long"

   Solution

   [POINTS 8: 1 point purpose statement, 2 points merge body, 2 points helper methods, 1 point throwing exception, 2 points examples of use with lists of Strings]

```
// Algorithms class: ----------------------------------------------
// Note: methods could be static.
/**
   * Merge the two sorted lists generated by two traversals, using the
   * <code>Comparator</code> to determine the order.
   *
   * @param <T> the data type used throughout
   * @param tr1 the traversal for the first dataset
   * @param tr2 the traversal for the second dataset
   * @param comp the <code>Comparator</code> for the data elements
   * @param result the resulting sorted <code>ArrayList</code>
   * @return
   */
  public <T> ArrayList<T> mergeAcc(Traversal<T> tr1,
      Traversal<T> tr2,
```

```
    Comparator<T> comp,
    ArrayList<T> result){
  try{
    if (tr1.isEmpty()){
      this.addAll(result, tr2);
      return result;
    }
    else if (tr2.isEmpty()){
      this.addAll(result, tr1);
      return result;
    }
    else
      if (comp.compare(tr1.getFirst(), tr2.getFirst()) < 0){
        result.add(tr1.getFirst());
        return mergeAcc(tr1.getRest(), tr2, comp, result);
      }
      else{
        result.add(tr2.getFirst());
        return mergeAcc(tr1, tr2.getRest(), comp, result);
      }
  }
  catch(IllegalUseOfTraversalException e){
    System.out.println(e.getMessage());
    return result;
  }

}

/**
 * Add all data elements generated by the given
 * <code>Traversal</code> to the given <code>ArrayList</code>.
 *
 * @param <T> the data type used throughout
 * @param result the resulting <code>ArrayList</code>
 * @param tr the given <code>Traversal</code>
 */
public <T> void addAll(ArrayList<T> result, Traversal<T> tr){
  try{
    while (!tr.isEmpty()){
      result.add(tr.getFirst());
```

```
        tr = tr.getRest();
      }
    }
    catch(IllegalUseOfTraversalException e){
      System.out.println(e.getMessage());
    }
  }

// Examples class: -------------------------------------------
  Algorithms algo = new Algorithms();

  ArrayList<String> arr1 = new ArrayList<String>();
  ArrayList<String> arr2 = new ArrayList<String>();

  public void initArr1Arr2(){
    arr1.add("Good Day");
    arr1.add("Hello");
    arr1.add("Servus");
    arr1.add("So Long");

    arr2.add("Bye");
    arr2.add("Ciao");
    arr2.add("Hi");
  }

  public ArrayList<String> useMerge(){
    initArr1Arr2();
    TraversalALC<String> tr1 = new TraversalALC<String>(arr1, 0);
    TraversalALC<String> tr2 = new TraversalALC<String>(arr2, 0);
    return algo.merge(tr1, tr2, cmp);
  }

  public void printString(ArrayList<String> arr){
    System.out.println("The list of String-s:");
    for(String s: arr)
      System.out.println(s);
    System.out.println("The end of the list of String-s:");
  }

  Comparator<String> cmp = new Comparator<String>(){
```

```
  public int compare(String s1, String s2){
    return s1.compareTo(s2);
  }
};
```

2. Define `tr1` and `tr2` each to represent a list of `Posn`s ordered by their distance to origin. You may assume that the class `Posn` defines the method

```
/** compute the distance of this Posn to the origin */
double distTo0();
```

Use the `merge` method you designed to merge two lists of `Posn`s. Make sure you include examples!

Solution

[POINTS 5: 1 point purpose statement, 1 point for usingthe mthod correctly, 1 point for defining the Comparator, 1 point for testing the Comparator, 1 point for examples for the merge method]

```
// Part 2: lists of Posn-s
  ArrayList<Posn> parr1 = new ArrayList<Posn>();
  ArrayList<Posn> parr2 = new ArrayList<Posn>();

  public void initParr1Parr2(){
    parr1.add(new Posn(0, 0));
    parr1.add(new Posn(0, 5));
    parr1.add(new Posn(3, 5));
    parr1.add(new Posn(7, 8));

    parr2.add(new Posn(2, 0));
    parr2.add(new Posn(2, 3));
    parr2.add(new Posn(8, 9));
  }

/**
   * use the merge method to merge two datasets of Posn-s
   * @return the sorted combined ArrayList
   */
  public ArrayList<Posn> useMergePosns(){
    initParr1Parr2();
    TraversalALC<Posn> tr1 = new TraversalALC<Posn>(parr1, 0);
```

```
    TraversalALC<Posn> tr2 = new TraversalALC<Posn>(parr2, 0);
    return algo.merge(tr1, tr2, pcmp);
  }
  /**
   * Print out the contents of the ArrayList of Posn
   * @param arr
   */
  public void printPosn(ArrayList<Posn> arr){
    System.out.println("The list of posn-s:");
    for(Posn p: arr)
      System.out.println("(" + p.x + ", " + p.y + ")");
    System.out.println("The end of the list of Posn-s:");
  }

  /**
   * The <code>Comparator</code> for Posn-s
   */
  Comparator<Posn> pcmp = new Comparator<Posn>(){
    public int compare(Posn p1, Posn p2){
      return (int) Math.floor(p1.distTo0() - p2.distTo0());
    }
  };

 /**
   * Test the <code>Comparator</code> for Posn-s
   */
  public void testComparator(){
    Posn p1 = new Posn(3, 4);
    Posn p2 = new Posn(8, 4);
    Posn p3 = new Posn(3, 2);
    System.out.println("Compare: p1 < p2" + (pcmp.compare(p1, p2) < 0));
    System.out.println("Compare: p1 > p3" + (pcmp.compare(p1, p3) > 0));
    System.out.println("Compare: p2 == p2" + (pcmp.compare(p2, p2) == 0));

  }
```

20 Points

Problem 3

Your library provides the following two interfaces:

```
/** to represent a function that determines the value of the object */
public interface Valuable{

  /** produce the value of this object */
  public int value();
}

/*-------------------------------------------------------*/

/** to represent an inventory of objects that have value */
public interface Inventory<Valuable>{

  /* Add the given item to the inventory and decrease the budget
   * by the value of the item and return the updated inventory */
  public Inventory<Valuable> buy(Valuable item);

  /* Does this inventory contain the given item? */
  public boolean contains(Valuable item);

  /* Produce the available budget amount */
  public int available();

  /* Produce an inventory with the given item removed
   * and the budget increased by its value */
  public Inventory<Valuable> sell(Valuable item);

  /* Is the inventory empty?*/
  public boolean isEmpty();
}
```

Your boss thinks that this interface could represent a portfolio of his stocks. For each stock trade order one needs to record the stock trading code, the number of shares that were bought and the price per share at the time it was bought. The portfolio starts with some available budget. We do not worry about gains or losses after the stocks have been purchased,

nor do we combine stocks with the same trading code bought in different transactions.

Note: You may use whole numbers for prices and the number of shares.

1. Design the class that represent one stock trade order (`class Stock`) — that implements `Valuable`, and then design the class or classes that represent a portfolio of stock trading that implements `Inventory`. The value of each `Stock` order is given by the price multiplied by the number of shares in the order. If we cannot carry out a transaction (either buying or selling) we should throw `TradeNotCompletedException` (abbreviated when handwritten as `TNCE`). Assume that the class `TNCE` has been defined and that it `extends` the Java `Exception` class.

(This page is intentionally left blank.)

Solution

[POINTS 20: 4 points for the class Stock; Portfolio class: 5 points for the buy method, 2 points for the contains method, 2 points for the available method, 5 points for the sell method, 2 points for the isEmpty method]

Note: Buy and sell methods throw exceptions - check that.

```
**
 * Class to represent stock market orders
 *
 * @author Viera K. Proulx
 *
 */

public class Stock implements Valuable{
  /** the trading code for this stock */
  String code;

  /** the number of shares in this order */
  int shares;

  /** the price per share in this order */
  int price;

  /** the full constructor */
  Stock(String code, int shares, int price){
    this.code = code;
    this.shares = shares;
    this.price = price;
  }

  /**
   * Compute the value of this order
   *
   * @return the value of this order
   */
  public int value(){
```

```
    return this.shares * this.price;
  }
}

import java.util.*;
/**
 * Class to represent a stock portfolio
 * and the cash available. Multiple orders of the same stock
 * are recorded separately, as they may have different prices.
 *
 * @author Viera K. Proulx
 *
 */

public class Portfolio implements Inventory<Stock>{

  /** the cash on hand */
  int cash;

  /** the stocks in this portfolio */
  ArrayList<Stock> stocks;

  public Portfolio(int cash){
    this.cash = cash;
    this.stocks = new ArrayList<Stock>();
  }

  public Portfolio(){
    this(0);
  }

  /** Produce an empty inventory with an inital budget */
  public Inventory<Stock> init(int budget){
    return new Portfolio(budget);
  }

  /**
   * Add the given item to the inventory and decrease the budget
   * by the value of the item
   *
```

```
 * @param item the item to buy
 * @return the updated inventory
 */
public Inventory<Stock> buy(Stock item){
  if (this.cash >= item.value()){
    this.stocks.add(item);
    this.cash = this.cash - item.value();
    return this;
  }
  else
    throw new RuntimeException(
        "cannot buy if you do not have enough money");
}

/**
 * Does this inventory contain the given item?
 *
 * @param item the desired item
 * @return true if the inventory contains the item
 */
public boolean contains(Stock item){
  return this.stocks.contains(item);
}

/**
 * Produce the available budget amount
 *
 * @return the current budget amount
 */
public int available(){
  return this.cash;
}

/**
 * Produce an inventory with the given item removed
 * and the budget increased by its value
 *
 * @param item the item to be removed
 * @return the updated inventory
 */
```

```
  public Inventory<Stock> sell(Stock item){
    int index = this.stocks.indexOf(item);
    if (index == -1)
      throw new RuntimeException(
          "cannot sell what you do not have");
    else{
      this.cash = this.cash - item.value();
      this.stocks.remove(index);
      return this;
    }

  }

  /**
   * Is the inventory empty?
   *
   * @return true if there are no items inthe inventory
   */
  public boolean isEmpty(){
    return this.stocks.isEmpty();
  }

  /**
   * Does this inventory contain stock with the given code?
   *
   * @param code the given code
   * @return true if the inventory contains stock with the given code
   */
  public boolean containsCode(String code){
    for (Stock stock: this.stocks){
      if (stock.code.equals(code))
        return true;
    }
    return false;
  }
}

// class Examples:
-----------------------------------------------------
 /** Examples of Stock-s */
```

```
 public Stock msft = new Stock("msft", 30, 10);
 public Stock akam = new Stock("akam", 20, 15);
 public Stock eink = new Stock("eink", 80, 20);
 public Stock disn = new Stock("disn", 30, 20);

 public Inventory<Stock> small = new Portfolio();

 public void trade(){
   this.small = this.small.init(10000);
   this.small = this.small.buy(msft);
   this.small = this.small.buy(eink);
   this.small = this.small.buy(akam);
   ((Portfolio)this.small).printPortfolio();
 }

/**
 * Test the mehtod <code>value</code>
 * ;; in the class <code>Stock</code>.
 * @param argv
 */
 public void testValueStock(){
   System.out.println("The value of msft should be 300: we got "
       + msft.value());
   System.out.println("The value of disn should be 600: we got "
       + disn.value());
 }

 /**
  * Test the method <code>isEmpty</code> and
  * initialize the Portfolio
  */
 public void testIsEmpty(){
   System.out.println("Empty: " + this.small.isEmpty());
   this.trade();
   System.out.println("Not empty: " + !this.small.isEmpty());

   System.out.println("money still available: " +
       this.small.available());
   System.out.println("should be: " + 7800);
```

```
  System.out.println("Contains msft: " +
      this.small.contains(msft));
  System.out.println("Does not contain disn: " +
      !this.small.contains(disn));
  this.small.sell(msft);
  ((Portfolio)this.small).printPortfolio();
}

public static void main(String[] argv){
  Examples e = new Examples();
  e.testValueStock();
  e.testIsEmpty()
}
```

14 Points

Problem 4

The US Census Bureau keeps track of all people, their addresses and other demographic information. It also keeps track of all dwellings and who lives there. So, for example, it may record the following information:

> John Doe, age 49, lives at 55 Sunrise Blvd, Sunny, CA
> Jane Deer, age 45, lives at 55 Sunrise Blvd, Sunny, CA
> Ebert Bear, age 78, lives at 44 Rocky Road, Snowy, ME
> Heather Bear, age 75, lives at 44 Rocky Road, Snowy, ME
> Pat Smart, age 23, lives at 33 High Road, Big City, NY

1. Design the classes that represent this data. For each `Person`, there should be a field that represents the name, the age, and the `Address`. For each `Address` there should be the street, the house number, the city and the state, and a list of `Person`s at that address. The `Census` contains both lists of data.

   Write the actual class definitions — class diagrams are not sufficient.

(This page is intentionally left blank.)

Solution

[POINTS 11: 3 points for the class Person, 3 points for the class Address, 2 points for the class Census, 3 points for the method that adds a person to the address's list of dwellers.]

```
/**
 * To represent a person in the US Census
 * @author Viera K. Proulx
 *
 */
public class Person{
  String name;
  int age;
  Address addr;


  public Person(
      String name,
      int age,
      Address addr){
    this.name = name;
    this.age = age;
    this.addr = addr;
    this.addr.addDweller(this);
  }

  /**
   * record this person's move to a new address
   * remove this person from the old address, if relevant
   * @param addr the new address
   */
  public void moveTo(Address addr){
    this.addr.removeDweller(this);
    this.addr = addr;
    this.addr.addDweller(this);
```

```
  }
}

import java.util.*;

/**
 * To represent an address in the US Census
 * @author Viera K. Proulx
 *
 */
public class Address{
  String street;
  int number;
  String city;
  String state;
  ArrayList<Person> dwellers;

  /**
   * The constructor - does not initialize the dwellers
   * @param street
   * @param number
   * @param city
   * @param state
   */
  Address(String street,
      int number,
      String city,
      String state){
    this.street = street;
    this.number = number;
    this.city = city;
    this.state = state;
    this.dwellers = new ArrayList<Person>();
  }

  /**
   * Record that the given person moved to this address
   * @param p --- the given person
   */
  public void addDweller(Person p){
```

```
    if (!this.dwellers.contains(p))
      this.dwellers.add(p);
  }

  /**
   * Record that the given person moved to this address
   * @param p --- the given person
   */
  public void removeDweller(Person p){
    if (this.dwellers.contains(p))
      this.dwellers.remove(p);
  }

  /**
   * Convenience method - to print all dwellers
   * @param dwellers
   * @return
   */
  public String printNames(ArrayList<Person> dwellers){
    String s = "list: ";
    for(Person p: dwellers){
      s = s.concat(p.name + " ");
    }
    return s;
  }

  /** support for pretty printing */
  public String toString(){
    return "new Address(" + this.number + " "
                          + this.street + ", "
                          + this.city + " "
                          + this.state + ")\nDwellers:"
                          + printNames(this.dwellers);
  }
}

import java.util.*;
/** to represent US Census */
public class Census{
  ArrayList<Person> people;
```

```
  ArrayList<Address> places;

  public Census(
      ArrayList<Person> people,
      ArrayList<Address> places){
    this.people = people;
    this.places = places;
  }

  /**
   * move the given person to a new address.
   * adding to the census if not yet there
   * also adding a new address if not yet entered
   *
   * @param p the person with a new address
   * @param a the new address
   */
  public void moveTo(Person p, Address a){
    if (!places.contains(a))
      places.add(a);
    if (!people.contains(p))
      people.add(p);
    p.moveTo(a);
  }
}
```

2. Show how you would represent the information shown at the beginning of this problem as data in your classes.

Solution

[POINTS 3: Just examples; circularly referential data - cannot be instantiated in the constructor.]

```
  Address ca = new Address("Sunrise Blvd", 55, "Sunny", "CA");
  Address me = new Address("Rocky Road", 44, "Snowy", "ME");
  Address ny = new Address("High Road", 33, "Big City", "NY");

  Person jd1 = new Person("John Doe", 49, ca);
  Person jd2 = new Person("Jane Doe", 45, ca);

  Person eb = new Person("Ebert Bear", 78, me);
  Person hb = new Person("Heather Bear", 75, me);

  Person ps = new Person("Pamela Smart", 23, ny);

  ArrayList<Person> people = new ArrayList<Person>();
  ArrayList<Address> places = new ArrayList<Address>();

  public void initLists(){
    people.add(jd1);
    people.add(jd2);
    people.add(eb);
    people.add(hb);
    people.add(ps);

    places.add(ca);
    places.add(me);
    places.add(ny);
  }

public void printAddr(ArrayList<Address> arr){
    System.out.println("The list of addresses:");
```

```
  for(Address a: arr)
    System.out.println(a.toString());
  System.out.println("The end of the list of Addresses:");
}

Census c = new Census(people, places);
```

3. Design the method `moveTo` that updates the `Census` when a `Person` moves to a new `Address`.

   Note: The address could be a newly built house; the person could be a newborn, or a foreigner who has moved to this country.

   Solution

   [POINTS 5: 1 point purpose, 1 point for checking if person already in the list and for checking whether this is a new address, 1 point body, 2 points tests for all possibilities]

```
//class Census: ------------------------------------
 /**
   * move the given person to a new address.
   * adding to the census if not yet there
   * also adding a new address if not yet entered
   *
   * @param p the person with a new address
   * @param a the new address
   */
  public void moveTo(Person p, Address a){
    if (!places.contains(a))
      places.add(a);
    if (!people.contains(p))
      people.add(p);
    p.moveTo(a);
  }

// class Examples: -----------------------------------
public void testMoveTo(){
    initLists();
    jd1.moveTo(ny);
    System.out.println("jd1.address should be " + ny.toString());
    System.out.println("jd1.address is " + jd1.toString());

    System.out.println("ny should contain " + jd1.toString());
    System.out.println("the answer is " + ny.dwellers.contains(jd1));
  }
```

Methods for ArrayList:

```
ArrayList<E>()    // the constructor

// append the specified element to the end of this list
boolean add(E e)

// inserts the specified element at the specified position in this list
void add(int index, E element)

// returns true if this list contains the specified element
boolean contains(E e)

// returns the element at the specified position in this list
E get(int index)

// tests if this list has no elements
boolean isEmpty()

// removes the element at the specified position in this list
// moves all elements at higher indices one position to the left
E remove(int index)

// replaces the element at the specified position in this list
// with the specified element - returns ths original element
E set(int index, E e)

// returns the number of elements in this list
int size()

// returns the index of the first occurrence of the given element
int indexOf(E e)
```