CSU213 Exam 2 – Spring 2006

Name:	
Student Id (last 4 digits):	
Homework login name:	
Instructor's Name + Time:	

• Write down the answers in the space provided.
• You may use all forms that you know from Java language. If you want to use a class from Java libraries that is not given to you, you must include the appropriate import statement. Otherwise, you do not need import statements. 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		/14
2		/ 6
3		/14
4		/15
5		/ 7
6		/14
Total		/70

14 Points

Problem 1

The control system for an elevator records each move of the elevator in a list of moves. The data definitions for moves is given by the following class diagram:

```
          +----------+
          | ALoMoves |<-----------+
          +----------+            |
          +----------+            |
               / \                |
               ---                |
                |                 |
       ------------------         |
       |                |         |
+-----------+    +--------------+ |
| MTLoMoves |    | ConsLoMoves  | |
+-----------+    +--------------+ |
+-----------+  +-| Move first   | |
               | | ALoMoves rest |-+
               | +--------------+
               v
           +------+
           | Move |
           +------+
           +------+
             / \
             ---
              |
       ----------------
       |              |
  +----------+   +----------+
  | Up       |   | Down     |
  +----------+   +----------+
  | int diff |   | int diff |
  +----------+   +----------+
```

The value of `diff` represents the number of floors elevator moves during each (up or down) move.

Answer the following questions in this context:

A. Design the method `dest` that produces the destination floor for this move, given the floor where the move originated.

Solution

[POINTS 3: 1 for purpose and header, 1 for two bodies, 1 for examples]

```
/** in the class Move: */
  // compute the destination, given the origin
  abstract int dest(int orig);

/** in the class Up: */
  // compute the destination, given the origin
  int dest(int orig){
    return orig + diff;}

/** in the class Down: */
  // compute the destination, given the origin
  int dest(int orig){
    return orig - diff;}
 \}

/** in the Examples class: */
  Move up3 = new Up(3);
  Move up2 = new Up(2);

  Move down2 = new Down(2);
  Move down3 = new Down(3);

  boolean testDest = up3.dest(4) == 7;
  boolean testDest = down3.dest(2) == 1;
```

B. Make three examples of `ALoMoves` and describe in English the meaning of the information each of them represents.

Solution

[POINTS 2: 1 for examples, 1 for explanation]

```
  Move up3 = new Up(3);
```

```
Move up2 = new Up(2);

Move down2 = new Down(2);
Move down3 = new Down(3);

ALoMoves mtlist = new MTLoMoves();
// elevator did not move

ALoMoves listOk = new ConsLoMoves(this.up3,
    new ConsLoObject(this.down2, this.mtlist));
// elevator went up 3 floors, then down 2 floors

ALoMoves listBad = new ConsLoMoves(this.up2,
    new ConsLoObject(this.down3, this.mtlist));
// elevator went up 2 floors, then down 3 floors

ALoMoves listOkFrom5 = new ConsLoMoves(this.down3,
    new ConsLoObject(this.up2,this.mtlist));
// elevator went down 3 floors, then up 2 floors

ALoMoves listBadfrom4 = new ConsLoMoves(this.down3,
    new ConsLoObject(this.down2,this.mtlist));
// elevator went  3 floors, then down 2 floors
```

C. Show the template for methods in the class `ConsLoMoves`.

Solution

[POINTS 1]

```
/* TEMPLATE:
... this.first ...                  -- Move
... this.first.dest(int) ...        -- int
... this.rest ...                   -- ALoMoves
... this.rest.mmm() ...             -- mmm-return-type
*/
```

D. Develop the method `currentFloor` that determines the current location of the elevator for this list of moves, given the starting floor.

Solution

[POINTS 3: 1 for purpose and header, 1 for the body 1 for examples]

```
  /** in the class ALoMoves: **/
  // determine where the elevator is now, if it started at the given floor
  abstract int currentFloor(int starting);

  /** in the class MTLoMoves: **/
  // determine where the elevator is now, if it started at the given floor
  int currentFloor(int starting){ return starting; }

  /** in the class ConsLoMoves: **/
  // determine where the elevator is now, if it started at the given floor
  int currentFloor(int starting){
    return this.rest.currentFloor(this.first.dest(starting));
  }

  boolean testCurrentFloor1 = this.mtlist.currentFloor(4) == 4;
  boolean testCurrentFloor2 = this.listOk.currentFloor(0) == 1;
  boolean testCurrentFloor3= this.listOkFrom5.currentFloor(5) == 4;
```

E. Develop the method `validMoves` that consumes an integer that represents the top floor for this elevator and determines whether the given list of moves represents a valid list of moves. The constraint is that the elevator cannot go any lower than the floor 0 or any higher than the given top floor. Assume that the elevator started at the floor zero when the recording started.

Solution

[POINTS 5: 1 for validMoves, 1 for helper validFrom (purpose + header), 1 for the body, 2 for examples]

```
/** in the class ALoMoves: **/
// is this a valid list of elevator moves?
abstract boolean validMoves();

// is this a valid list of elevator moves from the given floor?
abstract boolean validFrom(int floor);

/** in the class MTLoMoves:
// is this a valid list of elevator moves?
boolean validMoves(){ return true; }

// is this a valid list of elevator moves from the given floor?
boolean validFrom(int floor){
  return floor >= 0;
}

/** in the class ConsLoMoves: **/
// is this a valid list of elevator moves?
boolean validMoves(){
  if (this.first.dest(0) >= 0)
    return this.rest.validFrom(this.first.dest(0));
  else
    return false;
}

boolean validFrom(int floor){
  if (this.first.dest(floor) >= 0)
    return this.rest.validFrom(this.first.dest(floor));
  else
```

```
    return false;
}

/** Examples / Tests **/
boolean testValidFrom1 = this.mtlist.validFrom(4) == true;
boolean testValidFrom2 = this.mtlist.validFrom(0) == true;
boolean testValidFrom3 = this.mtlist.validFrom(-4) == false;

boolean testValidFrom4 = listOkFrom5.validFrom(5) == true;
boolean testValidFrom5 = listBadfrom4.validFrom(4) == false;

boolean testCurrentFloor1 = this.mtlist.currentFloor(4) == 4;
boolean testCurrentFloor2 = this.listOk.currentFloor(0) == 1;
boolean testCurrentFloor3= this.listOkFrom5.currentFloor(5) == 4;

boolean testValidList1 = this.mtlist.validMoves() == true;
boolean testValidList2 = this.listOk.validMoves() == true;
boolean testValidList3 = this.listBad.validMoves() == false;
```

6 Points

Problem 2

The Crook Construction Company (CCC) has an elaborate hierarchy of supervisors and laborers. Every supervisor has two subordinates. Each subordinate can be either another supervisor or a laborer.

A. Design the class definition in the form of a class diagram for the data that represents the organizational structure of the CCC and make examples of data.

Solution

[POINTS 2: 1 for the diagram, 1 for the examples]

```
          +-------------+
          | Emp         |<--------------+
          +-------------+               |
          | String name |               |
          +-------------+               |
                / \                     |
                ---                     |
                 |                      |
       ---------------------            |
       |                   |            |
+-------------+    +---------------+    |
| Laborer     |    | Boss          |    |
+-------------+    +---------------+    |
| int payBack |    | Emp helperOne |----+
+-------------+    | Emp helperTwo |----+
                   +---------------+

Emp lab1 = new Laborer("lab1", 200);
Emp lab2 = new Laborer("lab2", 100);
Emp lab3 = new Laborer("lab3", 250);

Emp boss2 = new Boss("b1", this.lab1, this.lab2);
Emp boss1 = new Boss("b1", this.boss2, this.lab3);
```

B. To get a job, each laborer and supervisor must participate in the kick-back scheme as follows:

- each laborer is assigned a fixed sum to pay as kickback for each job they are given.
- each supervisor collects the kickbacks from the subordinates and passes one half of the collected amount to his supervisor.

The chief supervisor passes on his half of the kickbacks as a bribe to make sure the CCC company gets the contract.

In the programs you write you may assume that the kickbacks are so large that half of the amount will always be an integer.

Develop the method `bribe` for this class hierachy that computes the amount of bribe the CCC company pays to secure a contract.

Solution

```
[POINTS 4: 1 for method in the Laborer class, 2 for method in the Boss
class, 1 for examples]
  /** in the class Emp: **/
  abstract int bribe();

  /** in the class Boss: **/
  // compute the amount of kickback this boss receives
  // --- and passes on to his boss
  int bribe(){
    return (this.helpOne.bribe()+ this.helpTwo.bribe()) / 2;
  }

  /** in the class Laborer: **/
  //compute the amount of kickback this laborer must pay
   int bribe(){
    return this.payBack;
  }

  /** Examples / Tests **/
  Emp lab1 = new Laborer("lab1", 200);
  Emp lab2 = new Laborer("lab2", 100);
  Emp lab3 = new Laborer("lab3", 250);
  Emp boss2 = new Boss("b1", this.lab1, this.lab2);
  Emp boss1 = new Boss("b1", this.boss2, this.lab3);

  boolean tesbribe1 = lab1.bribe() == 200;
  boolean testbribe2 = lab2.bribe() == 100;
  boolean tesbribe3 = lab3.bribe() == 250;
```

13 Points

Problem 3

Every week the Payroll Processing Company (PPC) produces paychecks for the employees of their contract organizations. For each of the organizations the PPC has a list of their employees with their name, id, and their hourly pay rate. The PPC company receives from each organization a list that for each employee records the number of hours worked during the payroll week. It produces a list of paycheck data that consists of the employee's name and the amount (in cents) the employee earned during this week.

Here are the data definitions for the information that is available to the PPC company:

```
// to represent an employee in a company
abstract class Emp{
  int id;
  String name;

  Emp(int id, String name){
    this.id = id;
    this.name = name; }
}
// to represent employee's permanent information
class EmpInfo extends Emp{
  int hourly;

  EmpInfo(int id, String name, int hourly){
    super(id, name);
    this.hourly = hourly; }
}
// to represent employee's weekly hours worked record
class EmpHours extends Emp{
  int hours;

  EmpInfo(int id, String name, int hours){
    super(id, name);
    this.hours = hours; }
}

// a class that provides traversal
// over a collection of EmpInfo data
Traversal<EmpInfo>

// a class that provides traversal
// over a collection of EmpHours data
Traversal<EmpHours>
```

A. Design the class `EmpCheck` that represents one instance of a payroll check.

Solution

[POINTS 1]

```
class EmpCheck extends Emp{
  int pay;

  EmpInfo(int id, String name, int pay){
    super(id, name);
    this.pay = pay;
  }
}

/** Examples / Tests **/

  Emp kaitCheck = new EmpCheck (123, "Kait", 350);
  Emp janCheck = new EmpCheck (123, "Jan", 480);
```

B. Provide data examples for the three classes that extend `Emp` and the two `Traversal` classes. (Use them in the rest of this problem.)

For the `Traversal` examples it is sufficient to use a Scheme-like notation:

```
  Traversal<X> trx = new Traversal<X>(-----, -----, -----);
```

Solution

[POINTS 2: data examples]

```
  Emp kaitInfo = new EmpInfo (123, "Kait", 10);
  Emp janInfo = new EmpInfo (123, "Jan", 12);

  Emp kaitHours = new EmpHours (123, "Kait", 35);
  Emp janHours = new EmpHours (123, "Jan", 40);

  Emp kaitCheck = new EmpCheck (123, "Kait", 350);
  Emp janCheck = new EmpCheck (123, "Jan", 480);
```

```
Traversal<EmpInfo> infoList =
     new Traversal<EmpInfo>(kaitInfo,janInfo)
Traversal<HoursInfo> infoList =
     new Traversal<HoursInfo>(kaitHours,janHours)
```

C. Develop the method `allChecks` that produces the payroll as an `ArrayList` of `EmpCheck`s, assuming every employee worked during the given week. Assume that both lists are ordered by the employee id-s.

Solution

[POINTS 5: 1 for purpose statement, 1 for empty clause, 1 for matched clause, 1 for correct loop/recursion, 1 for examples]

```
// produce payroll checks from the lists of employee info
// and hour worked
ArrayList<EmpCheck> computePay2(Traversal<EmpInfo> eR,
                                Traversal<EmpHours> hR){
  ArrayList<EmpCheck> arrlist = new ArrayList<EmpCheck>();
  if (hR.hasMore())
    if (eR.hasMore())
      if (hR.current().id == eR.current().id){

        arrlist.add(makeCheck(eR.current(),
                              hR.current().hours));
        return computePay2(eR.advance(), hR.advance());
      }
      else
        throw new IllegalArgumentException("input files do not match");
    else
      throw new IllegalArgumentException("input files do not match");
else
  return arrlist;
}

// produce pay checks from a list of employee info
// and a list of hours worked
ArrayList<EmpCheck> allChecks(){
  return computePay2(eR, hR);
}

/** Examples / Tests **/
ArrayList<EmpCheck> makeCheckList(){
  ArrayList<EmpCheck> ilist = new ArrayList<EmpCheck>();

  ilist.add((EmpCheck)kaitCheck);
  ilist.add((EmpCheck)janCheck);

  return ilist;
}

ArrayList<EmpCheck> result =
      pr.computePay2((Traversal<EmpInfo>)infoTr,
                     (Traversal<EmpHours>)hoursTr);

ArrayList<EmpCheck> expected = makeCheckList();
```

D. Develop the method `someChecks` that produces the payroll as an `ArrayList` of `EmpChecks`. This time the list of reported hours contains only those employees that worked at least one hour during the given week. Assume that both lists are ordered by the employee id-s.

Solution

[POINTS 6: 1 for purpose statement, 1 for empty clause, 1 for matched clause, 1 for advance on mismatch, 1 for reporting error, 1 for examples]

```
// produce payroll checks from the lists of employee info
// and hour worked: not every employee reporting hours
ArrayList<EmpCheck> computePay3(Traversal<EmpInfo> eR,
                                Traversal<EmpHours> hR){
  ArrayList<EmpCheck> arrlist = new ArrayList<EmpCheck>();
  if (hR.hasMore())
    if (eR.hasMore())
      if (hR.current().id == eR.current().id){
        arrlist.add(makeCheck(eR.current(), hR.current().hours));
        return computePay3(eR.advance(), hR.advance());
      }
      else
        return computePay3(eR.advance(), hR);
    else
      //return new ArrayList<EmpCheck>();
      throw new IllegalArgumentException("input files do not match");
else
  return arrlist;
}

// produce pay checks from a list of employee info
// and a list of hours worked
ArrayList<EmpCheck> someChecks(){
  return computePay3(eR, hR);
}

/** Examples / Tests **/
ArrayList<EmpCheck> makeCheckList2(){
  ArrayList<EmpCheck> ilist = new ArrayList<EmpCheck>();
  ilist.add((EmpCheck)janCheck);
  return ilist;
}

ArrayList<EmpCheck> result =
    pr.computePay3((Traversal<EmpInfo>)infoTr,
                   (Traversal<EmpHours>)hoursTr);

ArrayList<EmpCheck> expected = makeCheckList2();
```

15 Points

Problem 4

We all know that one bad apple can spoil the whole bunch. So, our goal is to eliminate all bad apples from the given `ArrayList` of `Apple`s. An apple is bad, if it has a worm:

```
class Apple{
  String kind; // granny smith, mackintosh, golden delicious, etc.
  boolean worm; // whether or not there is a worm in this apple
}
```

A. Design the class that implements the `ISelect` interface by selecting all bad apples.

```
interface ISelect<E>{
  boolean select(E e);
}
```

Solution

[POINTS: 1]

```
class BadApple implements ISelect<Apple>{
   public boolean select(Apple apple){
     return apple.worm;
   }
 }
```

B. Design the method `goodApples` in the `Examples` class that produces from the given `ArrayList` a new `ArrayList` that contains only the good apples.

Solution

[POINTS: 3: 1 for the empty clause, 1 for the loop, 1 for examples]

```
// produce a list of good apples from the given list of apples
  ArrayList<Apple> goodApples(ArrayList<Apple> alist){
    ArrayList<Apple> goodlist = new ArrayList<Apple>();
    for (int i = 0; i < alist.size(); i = i + 1){
      if (!alist.get(i).worm)
        goodlist.add(alist.get(i));
    }
    return goodlist;
  }

// solution using the ISelect interface:

Iselect s = new BadApple();

// produce a list of good apples from the given list of apples
  ArrayList<Apple> goodApples(ArrayList<Apple> alist){
    ArrayList<Apple> goodlist = new ArrayList<Apple>();
    for (int i = 0; i < alist.size(); i = i + 1){
      if (!s.select(alist.get(i)))
        goodlist.add(alist.get(i));
    }
    return goodlist;
  }

/** Examples / Tests **/
 ArrayList<Apple> makeList(){
    ArrayList<Apple> source = new ArrayList<Apple>();
    source.add(macok);
    source.add(mac);
    source.add(gd);
    source.add(gdok);
    source.add(gsok);
    source.add(gs);
```

```
   return source;
 }

// checks that the list produced by sortApples is correct
 boolean testGoodApples(ArrayList<Apple> sourcelist){
   ArrayList<Apple> alist = goodApples(sourcelist);
   return alist.get(0).worm == false &&
          alist.get(1).worm == false &&
          alist.get(2).worm == false &&
          alist.size() == 3;
 }
```

C. Design the method `sortApples` that mutates the given `ArrayList` by moving all the bad apples to the beginning of the `ArrayList`.

Solution

[POINTS 5:1 for the purpose-s and headers, 1 for the main method, 1 for the helper, 1 for swap, 1 for examples]

```
    // sort the apples in this list by moving the bad ones
    // to the lower end
    // apples at indices below low are already all bad.
    void sortApples(ArrayList<Apple> alist, int low){
      int badindex = findBad(alist, low);
      if (badindex < alist.size()){
        swap(alist, low, badindex);
        sortApples(alist, low + 1);
      }
    }

    // find the next bad apple in the bunch
    int findBad(ArrayList<Apple> alist, int low){
      int index = low;
      while(index < alist.size() &&
            !alist.get(index).worm)
        index = index + 1;
      return index;
    }

    // swap two elements of an arraylist
    void swap(ArrayList<Apple> alist, int i1, int i2){
      Apple tmp = alist.get(i1);
      alist.set(i1, alist.get(i2));
      alist.set(i2, tmp);
    }

  /** Examples / Tests **/
   boolean testSort(){
      ArrayList<Apple> alist = makeList();
      sortApples(alist, 0);

     return alist.get(0).worm == true &&
             alist.get(1).worm == true &&
             alist.get(2).worm == true &&
             alist.get(3).worm == false &&
             alist.get(4).worm == false &&
             alist.get(5).worm == false;
    }
```

D. Design the method `keepApples` that mutates the given `ArrayList` by removing all the bad apples at the beginning of the `ArrayList`. It consumes an `ArrayList` that has all the bad apples first, then the remainder of the `ArrayList` are the good apples.

Solution

[POINTS 3: 1 for the empty clause, 1 for the loop, 1 for examples]

```
  // remove from the low end of the array all bad apples
  void keepApples(ArrayList<Apple> alist){
    int index = 0;
    while(index < alist.size() &&
          alist.get(index).worm)
      alist.remove(index);
  }

  /** Examples / Tests **/
 boolean testKeep(){
    ArrayList<Apple> alist = makeList();
    sortApples(alist, 0);
    keepApples(alist);

   return alist.get(0).worm == false &&
           alist.get(1).worm == false &&
           alist.get(2).worm == false &&
           alist.size() == 3;
  }
```

E. Design the method `keepApples2` that mutates the given `ArrayList` by removing all the bad apples at the end of the `ArrayList`. It consumes an `ArrayList` that has all the good apples first, then the remainder of the `ArrayList` are the bad apples.

Solution

[POINTS 3: 1 for the empty clause, 1 for the loop, 1 for examples]

```
    // remove from the high end of the array all bad apples
    void keepApples2(ArrayList<Apple> alist){
      int index = alist.size() - 1;
      while(index >= 0 &&
            alist.get(index).worm){
        alist.remove(index);
        index = index - 1;
      }
    }

    /** Examples / Tests **/
   boolean testKeep2(){
      ArrayList<Apple> alist = makeList();
      sortApples2(alist, alist.size() - 1);
      keepApples2(alist);

     return alist.get(0).worm == false &&
             alist.get(1).worm == false &&
             alist.get(2).worm == false &&
             alist.size() == 3;
    }
```

8 Points

Problem 5

A. Consider the two solutions for removing the bad apples:

- moving all the bad apples to the beginning of the `ArrayList` then removing them
- moving all the bad apples to the end of the `ArrayList` then removing them

Which of the two methods would you choose to keep the good apples in a bunch of 10000 apples. Explain the reason why — provide estimates on the number of apples that need to be moved using each method to produce the result.

Solution

[POINTS: 3: 1 for correct choice, 1 for estimate for each choice]

Moving the bad apples to the end of the ArrayList. Removing from the end of the ArrayList does not involve any shifting of the other elements of the ArrayList, it is done in constant time.

The first method involes moving all of the rest of the apples one position to the left for each bad apple removed. If half of the apples are bad, this would be 999 + 998 + ... + 500 apples, for a total of 187,375 moves.

B. Imagine a data set of 1,000,000 items, that needs to be sorted by some criterion, and where the time required to move one piece of data from one place to another is about 2 seconds. For example, you may be moving sound records within a huge file on a disk.

Which of the following sorting algorithms would you choose to do this job?

(a) Mutating version of **quicksort**
(b) Mutating **binary tree sort**
(c) Mutating **insertion sort**
(d) Mutating **selection sort**

Justify your answer in one sentence.

Solution

[POINTS: 2: 1 for the correct choice, 1 for the justification]

Selection sort is the best, as it minimizes the number of moves - only n moves for data set of size n, and moves are the most time consuming tasks in this case.

C. Your next problem involves looking up repeatedly items in a large data set (1,000,000 elements). You can choose how the data should be stored in the computer, but once it is entered, we do not expect the data to change.

Which of the following strategies would you use:

(a) Data is organized as a `Cons` list, and we look up the item using sequential search.

(b) Data is organized as a `binary search tree` and we look up the item there.

(c) Data is stored in an `ArrayList` and we search for the item there.

(d) Data is stored in an `ArrayList` in a sorted order and we can choose our search strategy.

Justify your answer in one sentence.

Solution

[POINTS: 2: 1 for the correct choice, 1 for the justification; if BST is chosen, 1 point total]

Use binary search on sorted ArrayList - it is guaranteed to perform in log n time – Binary search tree is a second candidate, but could exhibit worst case behavior - or close to it, if the data is nearly sorted when received.

14 Points

Problem 6

A little child is playing with a string of colored beads, the beads come in several different shapes. You want to teach her about colors by putting all beads of the same color together. Being a computer geek, you decide to write a program to help you.

```
class Bead{
  Color color; // red, blue ochre, teal, peach, ...
  String shape; // ball, cube, tetrahedron, cylinder, ...
}

interface IMapping<K, V>{
  IMapping makeEmpty();
  boolean isEmpty();
  void add(K key, V value);
  boolean contains(K key);
  V getKey(K key);
}
```

A. Design the method `mixBeads` that consumes a `Traversal` of `Bead`s and an instance of an (originally empty) `IMapping` and produces an `IMapping` that uses colors as the keys and whose values are `ArrayLists` of beads.

Solution

[POINTS: 5:1 pt purpose statement, 1 pt correct loop through the input traversal, 2 pts for correct add to the list of beads, 1 pt for examples]

```
// produce an IMapping of ArrayLists of beads,
// coded by the color
IMapping<Color, ArrayList<Bead>> mixBeads(
    Traversal<Bead> tr,
    IMapping<Color, ArrayList<Bead>> result){

  while (tr.hasMore()){
    addBead(tr.current(), result);
    tr = tr.advance();
  }
  return result;
}

// add a bead to the color-based mapping of beads
void addBead(Bead b, IMapping<Color, ArrayList<Bead>> result){
  if (result.contains(b.color)){
    ArrayList<Bead> alb = result.getKey(b.color);
    alb.add(b);
    result.add(b.color, alb);
  }
  else {
    ArrayList<Bead> alb = new ArrayList<Bead>();
    alb.add(b);
    result.add(b.color, alb);
  }
}

/** Examples / Tests **/
// Examples of data
Bead red1 = new Bead(Color.red, "square");
Bead red2 = new Bead(Color.red, "ball");
Bead red3 = new Bead(Color.red, "cylinder");
Bead blue1 = new Bead(Color.blue, "ball");
Bead blue2 = new Bead(Color.blue, "square");

ArrayList<Bead> makeMixed(){
  ArrayList<Bead> beads = new ArrayList<Bead>();
  beads.add(red1);
  beads.add(blue1);
  beads.add(blue2);
  beads.add(red2);
  beads.add(red3);
  return beads;
}

  test("mixedBeads blue: ",
       sortedBeads.getKey(Color.blue).size(), 2);
  test("mixedBeads 1red: ",
       sortedBeads.getKey(Color.red).size(), 3);
```

B. Design the method `redBlue` that consumes the resulting `IMapping` and determines whether there are more red beads than blue beads.

Solution

[POINTS: 3: 1 pt purpose, 1 pt method, 1 pt examples]

```
// are there more red beads than blue beads in the given mapping?
boolean redBlue(IMapping<Color, ArrayList<Bead>> source){
  int blues = source.getKey(Color.blue).size();
  int reds = source.getKey(Color.red).size();
  return reds > blues;
}

/** Examples / Tests **/
IMapping<Color, ArrayList<Bead>> sortedBeads =
  mixBeads(new TraversalALC<Bead>(makeMixed()),
    new ArrayListMapping<Color, ArrayList<Bead>>());

test("red-blue", redBlue(sortedBeads), true);
```

C. Design the class `ArrayListMapping` that implements the `IMapping` interface by keeping an `ArrayList` of keys and an `ArrayList` of corersponding values.

Solution

[POINTS: 6: 1 pt correct two ArrayList fields, 1 pt for each method - half point off if no test - round up]

```
class ArrayListMapping<K, V> implements IMapping<K, V>{
  ArrayList<K> keys;
  ArrayList<V> values;

  ArrayListMapping(){
    keys = new ArrayList<K>();
    values = new ArrayList<V>(); }

  public IMapping<K, V> makeEmpty(){
    return new Beads<K, V>(); }

  public boolean isEmpty(){
    return keys.size() == 0; }

  public void add(K key, V value){
    if (keys.contains(key))
      values.set(keys.indexOf(key), value);
    else{
      keys.add(key);
      values.add(value);
    }
  }

  public boolean contains(K key){
    return keys.contains(key); }

  public V getKey(K key){
    return values.get(keys.indexOf(key)); }
}

  /** Examples **/
  void testArrayMapping(){
```

```
  ArrayListMapping<Color, Bead> mybeads =
    new ArrayListMapping<Color, Bead>();

  ArrayListMapping<Color, Bead> newBeads = mybeads.makeEmpty();

  TraversalTestHarness tth = new TraversalTestHarness();
  tth.test("makeEmpty", newBeads.keys.size(), 0);

  tth.test("isEmpty", mybeads.isEmpty(), true);

  mybeads.add(Color.red, red1);
  mybeads.add(Color.blue, blue2);

  tth.test("add", mybeads.keys.size(), 2);
  tth.test("add", mybeads.values.size(), 2);
  tth.test("add", mybeads.values.get(1), blue2);

  tth.test("contains", mybeads.contains(Color.blue), true);
  tth.test("contains", mybeads.contains(Color.green), false);

  tth.test("getKey", mybeads.getKey(Color.blue), blue2);

  mybeads.add(Color.blue, blue1);

  tth.test("getKey", mybeads.getKey(Color.blue), blue1);

  tth.fullTestReport();
}
```

```
interface Traversal<E>{
  // any more elements in the collection?
  boolean hasMore();
  // produce the current element in the collection
  E current();
  // produce a traversal for the rest of the collection
  Traversal<E> advance();
}

/** class 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)
```