23 Mutating ArrayList

Introduction

We would like to design the program that will allow us to keep track of our friends' phone numbers. As the time goes on, we may want to add new friends to the list, we may want to change someone's phone number as they find a new job. We will make several such phone lists, for example a family list, school friends, friends from the sports club, the acm friends, etc. Of course, some people may be listed in several of these lists. There are two problems we want to address. The first problem is that if a phone number for a person is changed, then one update operation should change the phone number in all lists. The second problem is that all of these lists are accessed by various programs and devices that have a reference to some, or all of these phone lists. Therefore, adding a new entry to a phone list cannot produce a new list the way we programmed so far. Instead, it needs to modify (mutate) an existing list.

Here is an example of the information we want to represent:

Friends: Pat 2345, Jan 1398, Kerry 6783
Family: Terry 8877, Alex 6655, Randy 5566
SwimClub: Terry 8877, Pat 2345, Jan 1398

PhoneLists: Friends, Family, SwimClub

We define a class *PhoneRec* to represent a phone record, an *ArrayList* for each of the three phone lists and an *ArrayList* to represent the list of phone lists. We omit the definition of the class *PhoneRec* and show the definition for only first three people in our lists, assuming the rest of them are defined in a similar way. The complete examples of data then become:

```
PhoneRec pat   = new PhoneRec("Pat", 2345);
PhoneRec jan   = new PhoneRec("Jan", 1398);
PhoneRec kerry = new PhoneRec("Kerry", 6783);
...

ArrayList friends  = new ArrayList();
ArrayList family   = new ArrayList();
ArrayList swimClub = new ArrayList();

ArrayList phoneLists = new ArrayList();
```

```
public void initLists(){
    friends.add(pat);
    friends.add(jan);
    friends.add(kerry);

    family.add(terry);
    family.add(alex);
    family.add(randy);

    swimClub.add(terry);
    swimClub.add(pat);
    swimClub.add(jan);

    phoneLists.add(friends);
    phoneLists.add(family);
    phoneLists.add(swimClub);
}
```

Until now we have made very little use of Java visibility modifiers. The reason we did not have to be concerned about it was that we never modified an existing instance and so our actions could not compromise the intergrity of the data contained in an instance of a class. In the presence of mutation it is very important to guard against inadvertent or inappropriate changes of the values that comprise an instance of a class. This is typically done by declaring the visibility of fields to be either private of protected (we will not explain the distinction here) and providing methods to access the information (get-ters) and to make changes in the values of some informoation represented by an instance (set-ters). The *get*(int *index*) and *set*(int *index*, *Object value*) methods for the *ArrayList* class are examples of get-ters and set-ters.

For example the class *PhoneRec* may provide a method *newNumber* that records a new phone number for the given person. The advantage is that the client has no idea whether the information about the old number is retained, or whether a person can have more than one phone number. Even though the original implementation may have been designed to hold only one, most recent, number, a later change to the implementation of the method *newNumber* will not 'break' the existing programs.

Here is the code for the method *newNumber* in the class *PhoneRec*:

```
// a class to represent one entry in a phone book
class PhoneRec{
  String name;
  int phone;

  PhoneRec(String name, int phone){
    this.name = name;
    this.phone = phone;
  }

  // change the phone number for this person
  public void newNumber(int phone){
    this.phone = phone;
  }
}

// examples
class Examples{
  PhoneRec pat   = new PhoneRec("Pat", 2345);
  PhoneRec jan   = new PhoneRec("Jan", 1398);

  this.pat.newNumber(3456);

  public void runTests(){
    test("Pat’s phone change: ", 3456, pat.phone);

    test("Pat’s phone in friends list: ",
        pat, friends.get(0));
    test("Pat’s phone in swimClub list: ",
        pat, swimClub.get(1));
  }
}
```

We would like to re-arrange the order in which the phone numbers are listed in the *family* list. We would like the entries to be in lexicographical order, with *alex* first and *terry* in the last position. The method

```
Object set(int index, Object element)
```

replaces the element at the specified position in this list with the specified element and returns the element previously at the specified position. We wish to make a three-way swap as follows:
Before:

```
+---------+---------+---------+
| 0: terry | 1: alex  | 2: randy |
+---------+---------+---------+
```

After:

```
+---------+---------+---------+
| 0: alex  | 1: randy | 2: terry |
+---------+---------+---------+
```

We need to save the reference to one item while we change the others as follows:

```
// three-way swap
public void swapThree(){
  Object temp = family.get(0);
  family.set(0, family.get(1));
  family.set(1, family.get(2);
  family.set(2, temp);
}
```

We intentionally did not take the full advantage of the fact that the *set* method returns the object that was at the specified location previously, just to make the logic clear.

Because Java is ill-equipped to deal with natural recursion, we are often forced to rewrite our algorithm into this mutating style. It is much harder to design tests for methods that include such side effects:

```
    test("Terry's's phone in family list: ", terry, family.get(0));
    test("Alex's phone in family list: ", alex, family.get(1));
    test("Randy's phone in family list: ", randy, family.get(2));

    swapThree();

    test("Alex's phone in family list: ", alex, family.get(0));
    test("Randy's phone in family list: ", randy, family.get(1));
  test("Terry's's phone in family list: ", terry, family.get(2));
```

We illustrate the use of the in-place mutation of an ArrayList by studying a mutating version of the selection sort algorithm. The ArrayList is divided into sorted lower part and unsorted upper part. During each iteration we find the location of the smallest element in the unsorted part, swap that element with the leftmost element of the unsorted part, thus increasing the size of the sorted part and decreasing the size of the unsorted part. At the beginning, the size of the sorted part is zero, at the end, the size of the unsorted part is zero:

```
  0   1   2    3   4   5   6   7
+---+---+---++---+---+---+---+---+
| a | c | f || t | k | m | h | w |
+---+---+---++---+---+---+---+---+
...sorted...   ... unsorted ...
```

Looking at the locations 3 through 7 we find the smallest element at location 6 and swap the elements at locations 6 and 3, increasing the sorted part by one and decreasing the unsorted part. After the swap we will have the following:

```
  0   1   2   3    4   5   6   7
+---+---+---+---++---+---+---+---+
| a | c | f | h || k | m | t | w |
+---+---+---+---++---+---+---+---+
  ...sorted...    ... unsorted ...
```

We need to design the following three methods:

```
// find the location of the smallest element
// in the upper part of this list
// using the given Comparator
int findMinLoc(ArrayList alist, int low, Comparator comp){
  ...
}

// swap the elements at the two given locations in this list
void swap(ArrayList alist, int loc1, int loc2){
  ...
}

// sort the list
void sort(ArrayList alist){
  int index = 0;
  while (index < alist.size() − 1){
    swap(alist, index, findMinLoc(alist, index + 1);
    index = index + 1;
  }
}
```

Follow the design recipe when completing these methods!!!

Example of an *ArrayList* Traversal

We add the tests from the previous lectures that included the method *contains*:

```
// does the structure traversed with the given iterator
// contain the given object
boolean contains(IRange it, Object obj);
```

We modify examples shown there to use an ArrayList for the data container:

```
ArrayList mtlist = new ArrayLlist();
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList();

public void runTests(){

    list1.add("Hello");
    list1.add("Hello");
    list2.add("Bye");

    IRange     itmt = new ArrayListRange(mtlist, 0);
    IListRange itl1 = new ArrayListRange(list1, 0);
    IRange     itl2 = new ArrayListRange(list2, 0);

    test("Test contains:", false, contains(itmt, "Hello"));
    test("Test contains:", true, contains(itl1, "Hello"));
    test("Test contains:", false, contains(itl1, "Bye"));
    test("Test contains:", true, contains(itl2, "Hello"));
    test("Test contains:", false, contains(itl2, "Hi"));
}
```