Problem Set 4
Due: 11:59m Friday, October 15

Exam 1 Notice:
On Friday, October 15, the first take-home exam will be posted. It will be due at 11:59pm on Sunday October 24 (giving you two full weekends to work on it). This is a hard deadline. No extensions will be given after this time. The exam will cover the material in lecture through Lecture 10 (Wed. October 13) and the material in problem sets through PS4: iteration, recursion, arrays, vectors, immutable lists, mutable lists, enumerations, text processing, using and implementing abstract data types, and creating and testing Java programs from scratch. Because you should focus on the exam, it is strongly recommended that you submit PS4 on time (11:59pm Friday October 15), although you may use lateness coupons to turn in PS4 late. But since the exam is worth much more than the problem set, it might be best to cut your losses on PS4 if you are not done with it so that you can focus on the exam.

Overview:
The purpose of this assignment is to give you practice with mutable and immutable lists, sorting, and abstract data type implementations.

Working Together:
Reminder: if you worked with a partner on PS1, PS2, or PS3 and want to work with a partner on this assignment, you must choose a different partner. Please let me know if you want a partner but are having trouble finding one.

Submission:
Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn’s office door by 6pm on the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet) indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment;
2. your final version of MergeSortND.java from Problem 1;
3. a transcript of your MergeSortND tests.
4. your final version of MergeSortD.java from Problem 2;
5. a transcript of your MergeSortD tests.
6. your final version of SetCard.java from Problem 3.
7. a transcript of your SetCard tests.

Each team should also submit a single softcopy (consisting of your final ps4 directory) to the drop directory ~cs230/drop/ps4/username, where username is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute the following commands in Linux in the account of the team member being used to store the code.

cd /students/username/cs230

cp -R ps4 ~cs230/drop/ps4/username/
Problem 0: Merge Sort

We begin by describing merge sort, an algorithm used in Problems 1 and 2. We will describe how this algorithm works on lists, though the same idea can be applied to arrays and vectors.

Merge sort is a recursive algorithm, and like almost all such algorithms it can be understood in terms of divide, conquer, and glue.

- **Divide**: In the divide step, the elements of the given list are divvied up into two lists of (nearly) equal length. If the length $n$ of the given list is even, then the resulting lists each has length $n/2$. If the length $n$ of the given list is odd, then the resulting lists have length $(n - 1)/2$ and $((n - 1)/2) + 1$. For this algorithm, it is irrelevant how the elements are divvied up. For example, here are some of the many possible ways to divvy up the elements of \((I, do, not, like, green, eggs, and, ham)\):

  1. \((I, do, not, like)\) and \((green, eggs, and, ham)\)
  2. \((like, not, do, I)\) and \((ham, and, eggs, green)\)
  3. \((I, not, green, and)\) and \((do, like, eggs, ham)\)
  4. \((and, green, not, I)\) and \((ham, eggs, like, do)\)

Although the method of splitting the original list into two nearly equal-length sublists is unimportant for the merge sort algorithm, in this assignment we will focus on an approach to splitting in which the first of the sublists is all the even-indexed elements (where 0 is the first index) in their original relative order, and the second of the sublists is all the odd-indexed elements in their original relative order. So on this assignment, the “correct” way to divvy up the sample list is option (3) above.

- **Conquer**: In the conquer step, each of the two lists resulting from the divide step is recursively sorted via the same algorithm. For example, if the two lists were \((I, not, green, and)\) and \((do, like, eggs, ham)\), then the result of this step is the two lists \((I, and, green, not)\) and \((do, eggs, ham, like)\). (Uppercase letters are lexicographically less than lowercase letters, so I comes before and in sorted order.)

- **Glue**: In the glue step, the two sorted lists resulting from the conquer step are merged into a single sorted result list. In our example, this yields \((I, and, do, eggs, green, ham, like, not)\). The merging process takes any two sorted input lists and returns a result list that has all the elements from both lists in sorted order.

For example:

```
(I,do,not,like,green,eggs,and,ham)
   /                 /
(I,not,green,and) (do,like,eggs,ham)
   /                 /
(I,green) (not,and) (do,eggs) (like,ham)
   /                 /
(I) (green) (not) (and) (do) (eggs) (like) (ham)
   /                 /
(I,green) (and,not) (do,eggs) (ham,like)
   /                 /
(I,and,green,not) (do,eggs,ham,like)
   /                 /
(I,and,do,eggs,green,ham,like,not)
```
Problem 1 [35]: MergeSort for Immutable Lists

On this problem, you will implement merge sort for immutable lists. Since this version of merge sort is non-destructive (in the sense that it does not change the head or tail components of any existing list node), we will use the name MergeSortND to name the class that holds the methods (where ND stands for “Non-Destructive”).

You should implement the following methods in the MergeSortND class:

**Public Class Methods:**

**public static ObjectList [] split (ObjectList L);**
Splits the elements of the given list L into two sublists, the first of which contains all the even-indexed elements (where indices start at 0) and the second of which contains all the odd-indexed elements. The relative order of elements in the sublists should be the same as in the original list. The two lists are returned as a two-element ObjectList array. For example, if L1 is the list (I, do, not, like, green, eggs, and, ham), then split(L1) should return an array r in which r[0] is the list (I, not, green, and) and in which r[1] is the list (do, like, eggs, ham).

**public static ObjectList merge (ObjectList L1, ObjectList L2);**
Assume that L1 and L2 are lists of Comparable elements, possibly containing duplicates, each of which ordered from least to greatest according to the compareTo function. Returns a list that contains all elements (including duplicates) and in which the elements are ordered from least to greatest according to the compareTo function.

**public static ObjectList sort (ObjectList L);**
Assume that L is a list of Comparable elements. Returns a list of all the elements in L (possibly including duplicates) sorted from least to greatest according to the compareTo function. The sorting should be performed via the merge sort algorithm.

**public static void main (String [] args);**
If args[0] is split, tests the split method on the list denoted by args[1]. If args[0] is merge, tests the merge method on the list denoted by args[1] and args[2]. If args[0] is sort, tests the sort method on the list denoted by args[1]. In each case, if the string denoting the list begins with an open parenthesis, it is assumed to be the string representation of a list. But a string not beginning with an open square bracket is assumed to be the name of a file whose words are the elements of the list. Use FileWords (see Appendix A) to enumerate words from a file. The ../text/ directory contains a number of files that can be uses as inputs. For example:

```
[lyn@w-206-198 ps4] java MergeSortND split "(I,do,not,like,green,eggs,and,ham)"
L = (I,do,not,like,green,eggs,and,ham)
ObjectList [] result = split(L);
result[0] = (I,not,green,and)
result[1] = (do,like,eggs,ham)
```

```
[lyn@w-206-198 ps4] java MergeSortND split "../text/cat-in-hat-4.txt"
L = (The,sun,did,not,shine,It,was,too,wet,to,play,So,we,sat,in,the,house,All,that,
cold,cold,wet,day)
result = split(L);
result[0] = (The,did,shine,was,wet,play,we,in,house,that,cold,day)
result[1] = (sun,not,It,too,to,So,sat,the,All,cold,wet)
```
Java MergeSortND merge "(I,and,green,not)" "(do,eggs,ham,like)"
L1 = (I,and,green,not)
L2 = (do,eggs,ham,like)
merge(L1,L2) = (I,and,do,eggs,green,ham,like,not)

Java MergeSortND sort "(I,do,not,like,green,eggs,and,ham)"
L = (I,do,not,like,green,eggs,and,ham)
sort(L) = (I,and,do,eggs,green,ham,like,not)

Java MergeSortND sort "../text/cat-in-hat-4.txt"
L = (The,sun,did,not,shine,It,was,too,wet,to,play,So,we,sat,in,the,house,All,that,
cold,cold,wet,day)
sort(L) = (All,It,So,The,cold,cold,day,did,house,in,not,play,sat,shine,sun,that,the,
to,too,was,we,wet,wet)

Flesh out the skeleton in MergeSortND.java. You may use the abbreviation OL for ObjectList and ObjectListOps. Feel free to implement any auxiliary methods that you find helpful.

Problem 2 [40]: MergeSort for Mutable Lists

On this problem, you will implement merge sort for mutable lists. Since this version of merge sort is destructive (in the sense that it does changes the tail components of existing list nodes), we will use the name MergeSortD to name the class that holds the methods (where D stands for “Destructive”).

You should implement the following methods in the MergeSortD class:

Public Class Methods:

public static void split (ObjectMList L);
Splits the elements of the given mutable list L into two sublists, the first of which contains all the even-indexed elements (where indices start at 0) and the second of which contains all the odd-indexed elements. The relative order of elements in the sublists should be the same as in the original list. This method should not create any new list nodes, nor should it change the head of any list node. It should perform all its work by modifying the tail components of list nodes. This method has void return type and so does not return a result. However, after the completion of split, the first node of the original list should be the first node of the even-indexed elements, and the second node of the original list should be the first node of the odd-indexed elements.

public static ObjectMList merge (ObjectMList L1, ObjectMList L2);
Assume that L1 and L2 are mutable lists of Comparable elements, possibly containing duplicates, each of which ordered from least to greatest according to the compareTo function. Returns a mutable list that contains all elements (including duplicates) and in which the elements are ordered from least to greatest according to the compareTo function. This method should not create any new list nodes, nor should it change the head of any list node. It should perform all its work by modifying the tail components of list nodes. This method returns the first node of the merged list. It should perform sorting via the merge sort algorithm.

public static ObjectMList sort (ObjectMList L);
Assume that L is a mutable list of Comparable elements. Returns a mutable list of all the elements in L (possibly including duplicates) sorted from least to greatest according to the compareTo function. This method should not create any new list nodes, nor should it change
the head of any list node. It should perform all its work by modifying the tail components of list nodes. This method returns the first node of the sorted list.

```java
public static void main (String [] args);
If args[0] is split, tests the split method on the list denoted by args[1]. If args[0] is merge, tests the merge method on the list denoted by args[1] and args[2]. If args[0] is sort, tests the sort method on the list denoted by args[1]. In each case, the testing method should show the state of the argument lists before the method invocation and after the method invocation. If the string denoting the list begins with an open parenthesis, it is assumed to be the string representation of a list. But a string not beginning with an open square bracket is assumed to be the name of a file whose words are the elements of the list. Use FileWords (see Appendix A) to enumerate words from a file. The ../text/ directory contains a number of files that can be uses as inputs. For example:

[lyn@w-206-198 ps4] java MergeSortD split "(I,do,not,like,green,eggs,and,ham)"
L = (I,do,not,like,green,eggs,and,ham)
ObjectList evens = L;
ObjectList odds = OML.tail(L);
split(L);
Before split: evens = (I,do,not,like,green,eggs,and,ham)
Before split: odds = (do,not,like,green,eggs,and,ham)
After split: evens = (I,not,green,and)
After split: odds = (do,like,eggs,ham)

[lyn@w-206-198 ps4] java MergeSortD split ".../text/cat-in-hat-4.txt"
L = (The,sun,did,not,shine,It,was,too,wet,to,play,So,we,sat,in,the,house,
All,that,cold,cold,wet,day)
ObjectList evens = L;
ObjectList odds = OML.tail(L);
split(L);
Before split: evens = (The,sun,did,not,shine,It,was,too,wet,to,play,So,we,
sat,in,the,house,All,that,cold,cold,wet,day)
Before split: odds = (sun,did,not,shine,It,was,too,wet,to,play,So,we,sat,in,
the,house,All,that,cold,cold,wet,day)
After split: evens = (The,did,shine,was,wet,play,we,in,house,that,cold,day)
After split: odds = (sun,not,It,too,to,So,sat,the,All,cold,wet)

[lyn@w-206-198 ps4] java MergeSortD merge "(I,and,green,not)" "(do,eggs,ham,like)"
Before merge: L1 = (I,and,green,not)
Before merge: L2 = (do,eggs,ham,like)
merge(L1,L2) = (I,and,do,eggs,green,ham,like,not)
After merge: L1 = (I,and,do,eggs,green,ham,like,not)
After merge: L2 = (do,eggs,green,ham,like,not)

[lyn@w-206-198 ps4] java MergeSortD sort "(I,do,not,like,green,eggs,and,ham)"
Before sort: L = (I,do,not,like,green,eggs,and,ham)
sort(L) = (I,and,do,eggs,green,ham,like,not)
After sort: L = (I,and,do,eggs,green,ham,like,not)
Flesh out the skeleton in `MergeSortD.java`. You may use the abbreviation `OML` for `ObjectMList` and `ObjectMListOps`. Feel free to implement any auxiliary methods that you find helpful.
Problem 3 [25]: Implementing SetCards as Integers

In lecture on Wed. October 13, we will see how the same abstract data type (ADT) can be implemented in many different ways. In particular, we will study two different implementations of the SetCard class from your earlier assignments. In this problem, you will define a third implementation of the SetCard class in ps4/SetCard.java.

This implementation is based on the following observation: there are precisely 81 cards in the Game of Set, so we can uniquely specify each card by a number from 0 to 80 that specifies the index the card would have in an array of lexicographically ordered cards. Fig. 1 shows the correspondence between each integer in the range 0–80 and the string representation of the card at that index.

| 0:1ebc | 1:1ebc | 2:1ebt | 3:1egc | 4:1egs | 5:1egt | 6:1erc | 7:1ers | 8:1ert |
| 9:1hbc | 10:1hbs | 11:1hbt | 12:1hgc | 13:1hgs | 14:1hgt | 15:1hrc | 16:1hrs | 17:1hrt |
| 27:2ebc | 28:2ebc | 29:2ebt | 30:2egc | 31:2egs | 32:2egt | 33:2erc | 34:2ers | 35:2ert |
| 36:2hbc | 37:2hbs | 38:2hbt | 40:2hgc | 40:2hgs | 40:2hgt | 42:2hrc | 43:2hrs | 44:2hret |
| 45:2fbc | 46:2fbs | 47:2fbt | 48:2fgc | 49:2fgs | 50:2fgr | 51:2frc | 52:2frs | 53:2fret |
| 54:3ebc | 55:3ebc | 56:3ebt | 57:3egc | 58:3egs | 59:3egt | 60:3erc | 61:3ers | 62:3ert |
| 63:3hbc | 64:3hbs | 65:3hbt | 66:3hgc | 67:3hgs | 68:3hgt | 69:3hrc | 70:3hrs | 71:3hret |
| 72:3fbc | 73:3fbs | 74:3fbt | 75:3fgc | 76:3fgs | 77:3fgt | 78:3frc | 79:3frs | 80:3fret |

Figure 1: Table showing the integer index of each of the 81 cards.

Based on this observation, it is possible to implement a version of the SetCard class in which each instance is has as its single instance variable an integer \( n \) specifying the unique index of the card. The class should obey the same SetCard contract specified in PS1, a copy of which appears in Appendix B for your convenience. For example, `new SetCard(2, 'h', 'r', 'c')` should create a new SetCard instance whose integer instance variable \( n \) is 42.

Your task in this problem is to flesh out the definition of the SetCard class in ps4/SetCard.java based on this idea. The behavior of your version of SetCard should be indistinguishable from that of the other representations we have studied. That is, no one using your SetCard implementation should have any clue that cards are being represented as integers “under the hood”.

**Notes:**

- The main method for your SetCard class should include testing code that shows that your methods work as advertised.
- Strive to make your method implementations as efficient as possible. For instance, one way to implement `toString` is to have a method with a lot of nested if statements, but a much better implementation uses the card index to pick the correct string out of an array of strings. Indeed, arrays are helpful for implementing many of the methods.
- Your method implementations (except for `toString` and `fromString`) should not convert the integer to some other representation (such as a string) and then use that other representation – that will be considered “cheating” (in a non-General-Judiciary sense ;-)).
- **Hint:** If \( n \) is the integer index variable, then the numbers \( n \mod 3 \), \( (n/3) \mod 3 \), \( (n/9) \mod 3 \), and \( n/27 \) are all interesting quantities.
Appendix A: FileWords Contract

The FileWords class is an implementation of the Enumeration interface that yields the words of a given file one by one. We shall assume that a “word” is any contiguous sequence of alphanumeric characters (i.e., letters and digits), as well as certain “special characters” – namely, ‘,’ ‘-’, ‘\’, and ‘\’ (i.e., the single quote character) – that occur between two alphanumeric characters. Any other characters, including special characters touching at least one non-alphanumeric character, are not part of any word. For example, the following sentences

Mary-Sue -- she wouldn’t use "foo_bar", 'baz!quux', or @this*that% to name 3.141 & $17.42. Would she?

contain the following 16 words:

Mary-Sue she wouldn’t use foo_bar baz quux or this that to name 3.141 17.42 Would she

In addition to the hasMoreElements and nextElement instance method required by implementations of Enumeration, FileWords supports the following constructor method and main testing method:

```java
public FileWords (String filename):
Create an enumeration that enumerates the words of the file named by filename.

public static void main (String [] args):
Invoke EnumTest.test on new FileWords(args[0]).
```

Appendix B: SetCard Contract

The SetCard class models a single card in the Game of Set. Such a card has four attributes, and three possible values for each attribute: (1) number: 1, 2, or 3; (2) shading: empty, filled, or hatched; (3) color: blue, green, or red; and (4) shape: circle, square, or triangle.

Public Constructor Methods:

```java
public SetCard (int number, char shading, char color, char shape):
Creates a card with the specified attributes. number should be either 1, 2 or 3; shading should be either 'e' (empty), 'f' (filled) or 'h' (hatched); color should be either 'b' (blue), 'g' (green) or 'r' (red); and shape should be either 'c' (circle), 's' (square) or 't' (triangle). For example, new SetCard(2,'f','g','c') creates a card with two filled green circles. If one or more parameters is not one of the allowed values for an attribute, a RuntimeException is thrown.
```

Public Instance Methods:

```java
public int number ();
Returns the number attribute of this card (either 1, 2, or 3).

public char shading ():
Returns the shading attribute of this card (either 'e' (empty), 'f' (filled), or 'h' (hatched)).
```
public char color ();
Returns the color attribute of this card (either 'b' (blue), 'g' (green), or 'r' (red)).

public char shape ();
Returns the shape attribute of this card (either 'c' (circle), 's' (square), or 't' (triangle)).

public String toString ();
Returns a string representation of this card. A string representation of a card has four characters, each of which specifies one of the four attributes of the card: the first is a digit specifying the number, the second specifies the shading, the third specifies the color, and the fourth specifies the shape. For instance "2ghs" is the string for the card that has two green hatched squares. Fig. 1 shows the string representations for all 81 cards in the Game of Set.

public boolean equals (Object x);
Returns true if x is a SetCard with the same four attributes as this card, and false otherwise.

public int compareTo (Object x);
If x is a SetCard instance, returns a negative number if this card comes before x in the card ordering (see the definition of card ordering below), 0 if this card is equal to x in the card ordering, and a positive number if this card comes after x in the card ordering. If x is not a SetCard instance, throws a ClassCastException.

The ordering on cards is defined as follows. Consider the following ordering on attributes: for numbers, 1 < 2 < 3; for shading, empty < filled < hatched; for colors, blue < green < red; and for shapes, circle < squares < triangle. Viewing the four attributes in the order number, shading, color, and shape induces a lexicographic ordering (i.e., dictionary ordering) on cards in which cards are first compared by number, then by shading, then by color, and finally by shape. Fig. 1 shows the ordering of the string representation all 81 cards in the Game of Set ordered from least to greatest according to the card ordering. The ordering has been chosen so that it coincides with the lexicographic ordering of the strings representing the cards. For example, two green filled triangles is “less than” two green hatched squares because "2gft" is less than "2ghs" in dictionary order.

Public Class Methods:

public static SetCard fromString (String s);
Returns a card whose string representation is s. E.g., SetCard.fromString("2ert") returns a card with two empty red triangles. Throws a RuntimeException if the string s is not a valid string representation of a card.
Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS230 Problem Set 4
Due 11:59 Friday October 15

Names of Team Members:

Date & Time Submitted:

Collaborators (anyone you or your team collaborated with):

By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.
Signature(s):

In the Time column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the Score column when grading your problem set.

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Reading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1 [35]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2 [40]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 3 [25]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>