COMP-202A, Fall 2007, All Sections
Assignment 5

Due: Tuesday, December 4, 2007 (23:55)

You **MUST** do this assignment individually and, unless otherwise specified, you **MUST** follow all the general instructions and regulations for assignments.

Note that for this assignments, graders **will generously** deduct marks for assignments that do not follow instructions and regulations.

Finally, the weight of this assignment in your final grade is equivalent to the combined weight of *two* of the previous assignments.

Failure to respect instructions and regulations:	0 to -20 points
Part 1, Question 1:	0 points
Part 1, Question 2:	0 points
Part 2, Question 1:	4 points
Part 2, Question 2:	5 points
Part 2, Question 3:	5 points
Part 2, Question 4:	6 points
	20 points total

Part 1 (0 points)

Do not hand in this part. It will not be graded. But doing this exercise might help you to do the second part of the assignment (that will be graded). If you have difficulties with the questions of Part 1, then we suggest you go to one of the office hours. The TA can help you and work with you through the warm-up questions.

Warm-up Question 1 (0 points)

Write a Java program which consists of one class called `ReverseFile`. This class should define one method, `main()`. Your program should do the following:

- It should check whether there are exactly two command-line arguments. If the number of command-line arguments is not exactly two, your program should display the following message:
  `Usage: java ReverseFile <input-file> <output-file>`
  It should then terminate.
- It should attempt to open the file whose path is given by the first command-line

argument for reading. If the file does not exist or if it cannot be opened for any other reason, your program should display an error message describing the situation and terminate.

- It should attempt to open the file whose path is given by the second command-line argument for writing. If an error occurs while trying to open the file (for example, the file is read-only, or the file does not exist and a file with that name cannot be created), your program should display an error message describing the situation and terminate.
- It should read the input file line by line and reverse the characters within each line (that is, the `String "abcde"` should become the `String "edcba"`). Also, it should write the reversed lines to the output file, but in the reverse order; that is, the first line in the input file should be the last line in the output file, the second line in the input file should be the second last line in the output file, and so on.
- Once it has read and reversed all the lines from the input file, and written the reversed lines to the output file in the reverse order, it should close the files.

If an error occurs at any point while reading from the input file or writing to the output file, your program should display an appropriate error message, attempt to close the files, and terminate. All error messages should be displayed to the standard error stream (that is, `System.err`) instead of the standard output stream (that is, `System.out`).

Hint: In order to reverse the order in which the lines appear in the output file, your program will most likely have to store them in memory. Note that your program should be able to handle a file containing any number of lines, up to the amount of memory available to the Java Virtual Machine.

Warm-up Question 2 (0 points)

The following program prompts the user to enter the number of integer to be read from the keyboard, reads that many integers from the keyboard, and stores them in an array in the same order as the one in which they were read. It then calls a recursive method called `isIncreasing()` which determines whether the integers in the array occur in increasing order, and displays the result.

The `isIncreasing()` method has a problem which makes the program that uses it crash on almost all arrays. Can you find it, describe what the problem is, and fix it?

Also, the `isIncreasing()` method is inefficient, as in the general case, it checks whether the elements in the rest of the array are in increasing order before checking the numbers in the last two positions. This always results in a number of recursive calls equal to the number of integers in the array, regardless of whether the array is in increasing order or not. Method calls have *overhead*, as they involve saving variables and return addreses on the run-time stack before the method is executed, and removing them from the run-time stack when the method finishes; this overhead should be minimized. Can you rewrite the general case so that it does not make unnecessary recursive calls?

Hint: For this problem, a recursive call is unnecessary when you can determine that the numbers in the array do not occur in increasing order just by looking at the last two

elements of the array.

```
import java.util.Scanner;

public class RecursionExercise {
  public static boolean isIncreasing(int[] array, int size) {
    boolean isIncreasing;

    if (size < 1) {
      isIncreasing = true;
    } else {
      isIncreasing = isIncreasing(array, size - 1);
      if (array[size - 2] > array[size - 1]) {
        isIncreasing = false;
      }
    }
    return isIncreasing;
  }

  public static void main(String[] args) {
    Scanner keyboard;
    boolean result;
    int[] myArray;
    int size;

    keyboard = new Scanner(System.in);

    System.out.print("Enter the number of integers you want your array "
      + "to contain: ");
    size = keyboard.nextInt();

    myArray = new int[size];
    for (int i = 0; i < size; i++) {
      System.out.print("Enter the value to be stored in position " + i +
        " of the array: ");
      myArray[i] = keyboard.nextInt();
    }

    result = isIncreasing(myArray, myArray.length);
    System.out.println("The values in this array are "
      + (result ? "" : "NOT") + " in increasing order.");
  }
}
```

Part 2 (4 + 5 + 5 + 6 = 20 points)

The questions in this part of the assignment will be graded.

Introduction

To learn more about digital audio tracks, refer to the document titled "Assignmment 5 Background Information"

In this assignment, you will write classes which can be used in a software music player to manage digital audio tracks stored in files on disk, using the information contained in the

metadata tags associated with each track. Each digital audio track will be represented by a `Song` object; the `Song` class has already been written for you.

In the classes you write, digital audio track management will involve a *music collection* and a *playlist*. A playlist is an ordered sequence of digital audio tracks, which can potentially contain repeated elements. Playlists are mainly used by music players to determine which audio tracks should be played and the order in which they should be played. They can be saved to disk so that they can be reloaded later.

A music collection, on the other hand, is a collection of audio tracks. Unlike a playlist, it does not define an order on the audio tracks it contains, and does not allow repeated elements. Its main use is as a central repository of information about audio tracks, so that tracks with certain properties can be easily retrieved.

To do this assignment, you will need the following files:

- `a5-basic-lib.zip`: An archive file containing all the Java libraries that you will need to write the classes required by this assignment, and which are not already included in the Java Platform API
- `a5-docs.zip`: An archive file containing documentation about the classes included in `a5-basic-lib.zip` that you are likely to use in order to write the classes required by this assignment.
- `a5-music.zip`: An archive file containing digital audio tracks in MP3 format which you can use as test data for the classes you write for this assignment.

Question 1 (4 points)

Write a Java class called `Playlist`. `Playlist` objects represent an ordered sequence of `Song` objects, and can potentially contain repeated elements (the same `Song` object can appear in two different positions in the `Playlist`). The `Playlist` class **MUST** provide **ALL** of the following methods:

- Two (2) constructors, which initialize a newly-created `Playlist`
- `add()`, which adds one `Song` to a `Playlist` (2 versions)
- `addAll()`, which adds multiple `Song`s to a `Playlist` (3 versions)
- `clear()`, which makes a `Playlist` empty
- `get()`, which returns a `Song` at a given position in the `Playlist`
- `getSize()`, which returns the number of `Song`s in a `Playlist`
- `isEmpty()`, which returns `true` if a `Playlist` is empty, `false` otherwise.
- `moveDown()`, which moves a `Song` at a given position towards the end of a `Playlist`
- `moveUp()`, which moves a `Song` at a given position towards the beginning of a `Playlist`
- `remove()`, which removes a `Song` at a given position from the `Playlist`

The internal workings of `Playlist` objects can be implemented in whatever way you wish, as long as your implementation of the `Playlist` class follows these instructions:

- It **MUST** behave as described in the specification given in the documentation for the `Playlist` class found in `a5-docs.zip`.
- It **MUST NOT** use Java features, constructs, or Java standard library classes and methods not covered in the lectures by the time the assignment is due, in accordance the general instructions and regulations. In particular, you can store the `Song`s a `Playlist` contains using either arrays or `ArrayList`s, but you **MUST NOT** use any other class that belongs to the Java Collection Framework such as `LinkedList` or any other. If you wish to use advanced data structures, you **MUST** implement them yourself.

Hint: Use an `ArrayList` instead of an array to store the `Song`s a `Playlist` contains.

Question 2 (5 points)

Add the `load()` and `save()` methods to the `Playlist` class you wrote in Question 1.

The `load()` method reads a *playlist file* containing the paths of files on disk, and changes the `Playlist` so it contains only the `Song`s representing the digital audio tracks in the files whose paths are in the playlist file, in the order in which they are found in the playlist file. The `Song` objects are retrieved from a `MusicCollection` (see Question 4); if the `MusicCollection` does not contain a `Song` corresponding to a path in the playlist file, the `Song` is constructed from information found in the metadata tags included with the digital audio track. The order in which the `Song`s appear in the `Playlist` when the method returns is the same as the order in which the paths of the files containing the digital audio tracks corresponding to the `Song`s appear in the playlist file.

The `save()` method writes the paths containing the digital audio tracks corresponding to the `Song`s the `Playlist` contains into a playlist file. The order in which the paths are written to the playlist file is the same as the order in which the corresponding `Songs` appear in the the `Playlist`.

Hint: Use an instance of the `SongFactory` class to create `Song` objects from the metadata tags found in the file containing digital audio tracks. The `SongFactory` class provides a method that reads the information stored in the metadata tags for you, so you do not have to do this yourself.

Question 3 (5 points)

Add the `sort()` method to the `Playlist` class you wrote in Question 1.

The `sort()` method sorts the `Song`s a `Playlist` contains using an order defined by the `SongComparator` object it takes as parameter; the `SongComparator` class has already been written for you.

You are free to use any sorting strategy to wish to implement the `sort()` method, with one caveat: you **MUST NOT** use any class or method from the Java Platform API which does automatic sorting for you.

One sorting strategy that you may use is called *selection sort*. It is quite simple and

works as follows:

- Find the *minimum* `Song` in the `Playlist`, that is, the `Song` that appears before all the others in the order defined by the `SongComparator` object.
- Swap the *first* `Song` in the `Playlist` with the minimum `Song` found in the previous step; that is, move the minimum `Song` to the first position in the `Playlist`, and move the `Song` previously at the first position into the position where the minimum `Song` previously was.
- Repeat the above steps for the remainder of the list, but excluding the previous positions from the search for the minimum `Song`; that is, when looking for the `Song` to move to a given position, do not consider the `Song`s in positions lower than this given position.

You can get up to 2 bonus points for this question if you implement the `sort()` method recursively instead of using loops. The recursive version of the selection sort strategy is the following:

- Find the *maximum* `Song` in the `Playlist`, that is, the `Song` that appears after all the others in the order defined by the `SongComparator` object. You can implement this step using a loop, or even define an helper method which does this step (this helper method can be either iterative or recursive, the choice is yours).
- Swap the *last* `Song` in the `Playlist` with the maximum `Song` found in the previous step; that is, move the maximum `Song` to the last position in the `Playlist`, and move the `Song` previously at the last position into the position where the maximum `Song` previously was.
- Recursively sort the rest of the `Playlist`.

Note that your total grade for this assignment cannot be higher than 20 / 20, even if you implement the `sort()` method recursively. However, bonus points you get for implementing the `sort()` method recursively may (perhaps partially) compensate for any mistakes you make in other questions.

Hint: In order to implement the `sort()` method recursively while respecting the specification, write a helper method which does the real work and is recursive. This additional method may take extra parameters, such as the size of the section of the list that needs to be sorted. Then, simply have the `sort()` method call your recursive helper method.

Question 4 (6 points)

Write a Java class called `MusicCollection`. Unlike a `Playlist`, a `MusicCollection` does not allow duplicate elements and the `Song`s it contains are not ordered. The path of the file associated with each `Song` (that is, the path of the file containing the digital audio track associated with a `Song`) is used as a primary key to uniquely identify each `Song` in the `MusicCollection`; if an attempt is made to add a `Song` to a `MusicCollection`, and the path of this `Song` is the same as the path of a `Song` already included in the `MusicCollection`, the `Song` **MUST NOT** be added to the `MusicCollection`.

The `MusicCollection` class **MUST** provide **ALL** of the following methods:

- A constructor, which initialize a newly-created `MusicCollection`
- `add()`, which adds one `Song` to a `MusicCollection`
- `addAll()`, which adds multiple `Song`s to a `MusicCollection` (2 versions)
- `clear()`, which makes a `MusicCollection` empty
- `getAllAlbums()`, which returns the list of all albums which include at least one of the `Song`s contained in a `MusicCollection`
- `getAllArtists()`, which returns the list of all artists which have produced at least one of the `Song`s contained in a `MusicCollection`
- `getContributedAlbums()`, which returns the list of all albums to which an artist has contributed at least one `Song` that is contained in a `MusicCollection`
- `getSong()`, which returns the `Song` object stored in a `MusicCollection` which corresponds to a given file path, if such a `Song` exists.
- `getSongs()`, which returns a `Playlist` containing all the `Song`s included in a `MusicCollection` which satisfy certain conditions (3 versions)

The internal workings of `MusicCollection` objects can be implemented in whatever way you wish, as long as your implementation of the `MusicCollection` class follows these instructions:

- It **MUST** behave as described in the specification given in the documentation for the `MusicCollection` class found in `a5-docs.zip`.
- It **MUST NOT** use Java features, constructs, or Java standard library classes and methods not covered in the lectures by the time the assignment is due, in accordance the general instructions and regulations. In particular, you can store the `Song`s a `Collection` contains using either arrays or `ArrayList`s, but you **MUST NOT** use any other class that belongs to the Java Collection Framework such as `LinkedList`, `Hashtable`, `HashMap`, `HashSet`, `TreeSet`, or any other. If you wish to use advanced data structures, you **MUST** implement them yourself.

Hint: You can use the `getSongs()` method which takes a `SongFilter` object in combination to the `sort()` method in the `Playlist` class to implement the other `getSongs()` methods, and even the `getAllArtists()`, `getAllAlbums()`, `getContributedAlbums()`, and `getSong()` methods. Look at the documentation for the `FilterFactory` class to find a way to create `SongFilter`s which would be useful to write those methods; this documentation is included in `a5-docs.zip`.

Additional Information

Important instructions:

- You **MUST** save the class you write for questions 1, 2, and 3 in **ONE** file called `Playlist.java`. Likewise, you **MUST** save the class you write for question 4 in a file called `MusicCollection.java`. Submit **BOTH** these files to WebCT. If you

implement other classes that are used by the two classes required by the assignment, you **MUST** submit the files containing the source code (files with extension `.java` for these classes as well. In accordance to the general instructions and regulations, submit to WebCT **ONLY** source code files relevant to the assignment.

- Full specifications for the all the methods you have to write in the `Playlist` and `MusicCollection` classes (the parameters these methods take, what these parameters represent, the return types of these methods, the values these methods return in various circumstances, the effects that calling each method has on the state of a `Playlist` or `MusicCollection` object, and so on) can be found in the documentation for the `Playlist` and `MusicCollection` classes, included in `a5-docs.zip`. The methods you implement **MUST** respect **ALL** the specifications for the `Playlist` and `MusicCollection` classes that are found in the documentation.
- Remember that, in accordance with the general instructions and regulations for assignments, all attributes of the classes you write **MUST** be `private`, and all the methods you write **MUST** be `public`. If you define helper methods not listed in the above specification or in the documentation, these methods **MUST** be `private`. Also remember that not specifying an access modifier results in an access modifier which is neither `public` nor `private`.

Other tips and information:

- The assignment does not require that you hand in any test programs. However, it will be necesary for you to write / use such programs in order to verify that the classes required by this assignment work properly. There are a couple of ways you can use to test the classes you write for this assignment:
  - You can write a program that creates "dummy" `Song` objects with "dummy" values, and use these "dummy" `Song` objects to verify that your `Playlist` and `MusicCollection` classes work properly.
  - A test program template is included in `a5-basic-lib.zip`; this program is stored in the file called `A5TestTemplate.java`. Test programs built using this template prompts the user to enter a directory where MP3 files are located, and loads all files in this directory (as well as any subdirectories) into a `MusicCollection`. You can add code to this template that tests whether or not the methods you have implemented work as you expect. The file `SimpleMusicCollectionTest.java` contains a simple test program which tests some of the functionality of the `MusicCollection` class.

    When you first test your classes using test programs built from this template, do not use your entire MP3 collection as a test set. Instead, use a directory containing only a few files. If your methods work, progressively add files to this directory, or progressively use directories which contain more files.
  - A more exhaustive test program will be made available as the due date for this assignment gets closer.
- To combine the Java libraries in `a5-basic-lib.zip` with the classes you write, and

have everything compile and run under DrJava, do the following:

- Extract the contents of `a5-basic-lib.zip` in a directory on your hard drive. This will create two new subdirectories in that directory; one will be called `a5-code` and the other will be called `a5-licenses`.
- Save the `.java` files that you write (`Playlist.java`, `MusicCollection.java`, and any test programs that you write) in the `code` directory (that is, the directory where `A5testTemplate.java` and `SimpleMusicCollectionTest.java` are located).
- Compile and run your files normally in DrJava.