ASSIGNMENT 5
Data Structures, Files, Exceptions, and To-Do Lists

COMP-202B, Winter 2009, All Sections

Due: Tuesday, April 14, 2009 (23:55)

You **MUST** do this assignment individually and, unless otherwise specified, you **MUST** follow all the general instructions and regulations for assignments. Graders have the discretion to deduct up to 10% of the value of this assignment for deviations from the general instructions and regulations.

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

Part 1, Question 1:	0 points
Part 2, Question 1:	70 points
Part 2, Question 2:	80 points
Part 2, Question 3:	50 points
	200 points total

Part 1 (0 points): Warm-up

Do **NOT** *submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that consult the TAs during their office hours; they 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 error 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.

Part 2 (70 + 80 + 50 = 200 points)

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

Introduction

In this assignment, you will write classes which can be used in a small application which manages your to-do list. The classes you write are used by the to-do list application to perform three tasks:

1. Loading the tasks on your to-do list from secondary storage when the application starts
2. Keeping track of these tasks while the application is running
3. Saving these tasks to secondary storage when the application is closed.

Tasks managed by this application have a description as well as an optional deadline. The information pertaining to a task is stored in a task record, and each such record will be represented by a `Task` object; the `Task` class has already been written for you.

The classes that you write in this assignment will include a *to-do list*, a *loader*, and a *writer*. A to-do list is a collection of task records which does not define an order on the records it contains, and does not allow repeated elements. Its main use is to be a central repository of information about tasks to be performed. The class provides methods to add a task to the to-do list, remove task from a to-do list, and retrieve all the tasks currently on to-do list.

The information stored in the task records contained in a to-do list can be saved to and loaded from files located on a secondary storage device such as the hard drive. This will enable the to-do list application to save the tasks on the to-do list when the application is closed, and to read them back when the application is restarted. Therefore, the purpose of the loader is to read files containing information about task records, and to recreate the records from the information it reads from those files. Conversely, the purpose of the writer is to save the information stored in task records to a file in such a way that the records can later be recreated by the loader using the data in the file.

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

- `a5-basic.jar`: An archive file containing all the Java classes that you may need to use in order to write the classes required by this assignment and which are not already included in the Java Class Library.
- `a5-docs.zip`: An archive file containing documentation about the classes that you will write for this assignment, as well as the classes included in `a5-basic.jar` that you are likely to use to complete this assignment.

- `a5-tests.zip`: An archive file containing test programs for each of the classes that you will write for this assignment.
- `junit.jar`: An archive file containing all the classes implementing a test framework called JUnit. The test programs in `a5-tests.zip` depend on JUnit in order to work.
- `a5-gui.zip`: An archive file containing the implementation of the user interface for the COMP-202 to-do list application. This application will use the classes that you write for this assignment in order to keep track of the tasks the user needs to complete.

Important note: You **MUST** make sure that **ALL** your classes meet **ALL** the requirements. This includes both the requirements in this specification **AND** the requirements in the documentation included in `a5-docs.zip`.

Question 1: The To-Do List (70 points)

Write a Java class called `ToDoList`, to be saved in a file called `ToDoList.java`. The purpose of a `ToDoList` objects is to store references to `Task` objects.

A `ToDoList` can never contain duplicate elements. That is, if an attempt is made to add a reference to a `Task` object to a `ToDoList`, and a reference to the same `Task` object is already stored in the `ToDoList`, the latter will not change as a result. However, references to different `Task` objects whose attributes contain equal values can be added to a `ToDoList` without restriction.

Furthermore, the `ToDoList` has no concept of element order; that is, the order in which the `Task` references are stored in the `ToDoList` has no importance; the same is true of the order in which they are returned when they are retrieved.

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

- A constructor `ToDoList()`, which initializes a newly-created `ToDoList`
- `boolean addTask(Task)`, which adds a reference to a `Task` to a `ToDoList`
- `void clear()`, which makes a `ToDoList` empty
- `ArrayList<Task> getAllTasks()`, which returns an `ArrayList` containing all `Task` references currently stored in the `ToDoList`
- `int getSize()`, which returns the number of `Task` references that the `ToDoList` currently contains
- `boolean removeTask(Task)`, which removes the reference to the specified `Task` from the `ToDoList` if it is present.

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

- It **MUST** behave exactly as described in the specification given in the documentation for the `ToDoList` class found in `a5-docs.zip`.
- It **MUST** keep track of the `Task` references it contains using a **PLAIN ARRAY** (of `Task`s); you **MUST NOT** use `ArrayList`s (or any other class which is part of the Java Collection Framework such as `LinkedList`, nor any other class you obtain from third-party libraries such as the Apache Commons Collections) **AT ALL** within the `ToDoList` class, with **ONE** exception; you may, of course, use an `ArrayList` of `Task`s in the `getAllTasks()` method (but **ONLY** in that one method).

  Because the specification given in the documentation for the `ToDoList` class found in `a5-docs.zip` mentions that the number of `Task` references which can be added to a `ToDoList` is limited only by the amount of memory available to the Java Virtual Machine, the array used to store the `Task` references in a `ToDoList` object will have to grow each time it is full and a `Task` reference is added to the `ToDoList`. In addition, in order to make unused memory available for reuse, this array will have to shrink when the number of actual `Task` references it contains drops below a certain percentage of the number of `Task` references it can contain. Because "growing" and "shrinking" an array are expensive operations (they involve creating a new array with a different capacity, and copying all the elements from the old array to the new), you **MUST** minimize the number of times the array grows and shrinks by implementing the following array resizing scheme:

– When a new `ToDoList` object is created, the array it uses to store `Task` references can contain a maximum of 10 references.

– Every time the array is full and needs to be grown in order to accomodate a new `Task` reference, the new array will have twice the capacity of the old array. You **MUST NOT** grow the array if it is not full.

– Every time the number of `Task` references in the array drops below 25% of the array's capacity, and the latter is greater than 10, the array needs to be shrunk. In such cases, the new array will have half the capacity of the old array. You **MUST NOT** shrink the array if it is more than 25% full. Likewise, when the array's capacity is 10, you **MUST NOT** shrink the array further, regardless of the number of actual elements it contains.

You are also responsible for finding a way to keep track of the actual number of `Task` references stored in the array at any given time (as opposed to the array's capacity; that is, the number of `Task` references it can contain before it needs to grow).

Question 2: Loading the Tasks (80 points)

Write a Java class called `ToDoListLoader`, to be saved in a file called `ToDoListLoader.java`. An object which belongs to this class reads *task files*; these files contains information from which it is possible to create `Task`s. The object then creates `Task`s from this information, adds references to these `Task`s to a new `ToDoList` it creates, and returns this new `ToDoList`. The `ToDoListLoader` class **MUST** provide **ALL** of the following methods:

- A constructor `ToDoListLoader()`, which initializes a newly-created `ToDoListLoader` object
- A `ToDoList load(java.io.File)` method, which reads a task file whose path is given by a `java.io.File` object, creates `Task`s from the information it contains, adds references to these `Task`s to a new `ToDoList` that it creates, and returns this new `ToDoList`

The documentation for the `ToDoListLoader` class included in `a5-docs.zip` describes in detail the format in which the information stored in each `Task` is written in the task file. To summarize, each `Task` is saved to the file as follows:

1. The description attribute of the `Task` is written on a line in the file.
2. If the `Task` has a deadline, the `String "true"` is written on the following line; conversely, if the `Task` has no deadline, then the `String "false"` is written on the following line. There can be zero or more space or tab characters both before and after the `String` on this line. This line is referred to as the *deadline indicator line.*
3. If the `Task` has a deadline (and, consequently, the `String "true"` was written on the deadline indicator line for this `Task`), then the integer values which make up the deadline attribute of the `Task` are all on the line immediately after the deadline indicator line of the `Task`, in the following order: year, month, day of the month, hour, minutes, seconds. Each value is separated by one or more space or tab characters. In addition, there can be zero or more space or tab characters before the year value and after the seconds value; however, between the seconds value and the end of the line, there cannot be any characters other than spaces or tabs.

Note that any number of empty lines can occur before the description of the first `Task` in the task file, after the last line containing information about the last `Task` in the file (that is, the deadline indicator line if the `Task` has no deadline, and the line containing the date if the `Task` has a deadline), and between the last line containing information about a `Task` and the description attribute of the next `Task` in the task file.

Hint 1: It is very possible to write this class without declaring any instance or static variables.

Hint 2: The specification of the

```
ToDoList load(java.io.File)
```

method states that the latter must throw some types of exceptions in certain situations. In some of these situations, an exception of the appropriate type will be thrown automatically by the statements you write in this method; for example, if you create a new `Scanner` object that reads from a file, the constructor of the `Scanner` class will automatically throw a `FileNotFoundException` if the file on disk the `Scanner` object was supposed to read from does not exist.

However, in other situations, you will have to write code to detect that a problem has occurred and explicitly throw an exception of the appropriate type in such cases. You can sometimes detect these problems by explicitly checking for them with an `if` statement. You can explicitly throw an exception with the following statement:

```
throw new ExceptionType();
```

where `ExceptionType` is replaced by the class that the exception you want to throw belongs to. For example, if you wished to write a statement that throws an exception of type `MyBizarreException`, you would write:

```
throw new MyBizarreException();
```

However, for other types of problems, an exception of another type will be thrown by the statements you write when such problems occur; you will then have to catch this exception and explicitly throw an exception of the appropriate type instead.

Question 3: Saving the Tasks (50 points)

Write a Java class called `ToDoListWriter`, to be saved in a file called `ToDoListWriter.java`. An object that belongs to this class writes the information contained in `Task`s to task files in such a way that a `ToDoListLoader` can recreate the original `Task`s from this information. The `ToDoListWriter` class **MUST** provide **ALL** of the following methods:

- A constructor `ToDoListWriter()`, which initializes a newly-created `ToDoListWriter` object
- A `void save(ToDoList, java.io.File)` method, which writes the information stored in the `Task`s whose references are contained in the specified `ToDoListWriter` to a task file whose path is given by a `java.io.File` object

The documentation for the `ToDoListWriter` class included in `a5-docs.zip` describes in detail the format in which information about each `Task` is written to the task file. To summarize, it is the same format as the one understood by the `ToDoList load(java.io.File)` method of the `ToDoListLoader` class (see Question 2).

Hint: Like the `ToDoListLoader` class described in Question 2, it is very possible to write this class without declaring any instance or static variables.

Additional Information

Important instructions:

- You **MUST** save the classes you write for the above questions in the following files:
  - Question 1: `ToDoList.java`
  - Question 2: `ToDoListLoader.java`
  - Question 3: `ToDoListWriter.java`

  Submit **ALL THREE** of these files to myCourses. If you implement other classes that are used by the five 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 three required classes (`ToDoList`, `ToDoListLoader`, and `ToDoListWriter`), including 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 the target object (if any), and so on, can be found in the documentation for the `ToDoList`, `ToDoListLoader`, and `ToDoListWriterWriter` classes; this documentation is included in `a5-docs.zip`. The methods you implement **MUST** respect **ALL** the specifications for the `ToDoList`, `ToDoListLoader`, and `ToDoListWriterWriter` 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 which are required by the above specification and the documentation **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 submit 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. Here are a couple approaches you can take to perform this task:
  - You can write a program that creates "dummy" `Task` objects with "dummy" values. You can then use these "dummy" `Task` objects to verify that the classes you wrote for this work properly.
  - You can use the test programs provided in `a5-tests.zip`. Extract the contents of `a5-tests.zip` to the folder in which you have saved the files containing the classes required by this assignment. Then, compile the test files and run them.

    Note that if you use the provided test programs, you need make sure to test your classes in the following order, as the test for each of these classes depends on the correctness of the implementation of the previous class:

    1. `ToDoList`
    2. `ToDoListLoader`
    3. `ToDoListWriter`
- To combine the Java libraries in the provided JAR files with the classes you write so that everything compiles and runs using the JDK, you need to specify when compiling or running your program that the JAR files should be on the *Classpath*. To do so, follow the steps below:
  1. Place the JAR files in the same directory / folder as the one in which the files containing the classes that you wrote are stored.
  2. Open the command-line interface (Command Prompt under Windows, Terminal under Mac OS X, Linux, or other Unix-like operating systems) and navigate normally to the directory / folder containing your files.
  3. To compile a file called `MyFile.java` under Windows, issue the following command:

     ```
     javac -cp a5-basic.jar;.  MyFile.java
     ```

     If file `MyFile.java` is one of the provided test programs and not a file that you wrote, issue the following command instead:

     ```
     javac -cp a5-basic.jar;junit.jar;.  MyFile.java
     ```

To compile a file called `MyFile.java` under Mac OS X, Linux, or another Unix-like operating system, issue the following command:

```
javac -cp a5-basic.jar:.  MyFile.java
```

If file `MyFile.java` is one of the provided test programs and not a file that you wrote, issue the following command instead:

```
javac -cp a5-basic.jar:junit.jar:.  MyFile.java
```

The only difference between the Windows and Unix versions of the above commands is that in the Windows version, a semi-colon (`;`) separates each Classpath entry; in the Unix version, a colon (`:`) separates each Classpath entry instead. Make sure to replace `MyFile.java` with the actual name of the file you wish to compile.

4. To run the `main()` method defined in a class called `MyFile` under Windows, issue the following command.

   ```
   java -cp a5-basic.jar;.  MyFile
   ```

   If file `MyFile.java` is one of the provided test programs and not a file that you wrote, issue the following command instead:

   ```
   java -cp a5-basic.jar;junit.jar;.  MyFile
   ```

   To compile a file called `MyFile.java` under Mac OS X, Linux, or another Unix-like operating system), issue the following command:

   ```
   java -cp a5-basic.jar:.  MyFile
   ```

   If file `MyFile.java` is one of the provided test programs and not a file that you wrote, issue the following command instead:

   ```
   java -cp a5-basic.jar:junit.jar;.  MyFile
   ```

   Again, the only difference between the Windows and Unix versions of the above commands is that in the Windows version, a semi-colon (`;`) separates each Classpath entry; in the Unix version, a colon (`:`) separates each Classpath entry instead. Make sure to replace `MyFile` with the actual name of the class containing the `main()` method you wish to run.

5. Make sure to add the appropriate `import` statements to the classes you write. For example, a class which uses the `Task` class needs the following `import` statement:

   ```
   import comp202.winter2009.a5.util.Task;
   ```

You should now be able to compile the files containing the classes that you wrote, as well as the provided test programs, without the compiler reporting errors related to missing classes. You should also be able to run provided test programs, as well as any test programs you wrote, without the virtual machine reporting such errors. Note that the `-cp` option is **ONLY** necessary when the code you want to compile / run requires libraries located in another directory or a JAR file.

- To combine the Java libraries in the provided JAR files with the classes you write so that everything compiles and runs using Eclipse, you need to add the JAR files to the Classpath of your project. To do so, follow the steps below:

  1. In the `Package Explorer` or `Navigator` view, right-click on your project. In the context menu that appears, select the `Properties` option.
  2. The window that appears will be split in two main portions. The smaller, left-hand portion contains a number of option categories, and you can select a category from those listed; the larger middle portion contains the options for the selected category. In the left-hand portion of the window, choose `Java Build Path`.

3. The middle portion of the window will now contain four tabs. Click on the `Libraries` tab. The libraries included in your project will now appear, along with a number of buttons; click on the button labelled `Add External JARs...`.
4. Using the file selection window that appears, navigate to the directory / folder where you saved the JAR files, select them, and click on the confirmation button (it might be labelled `OK`, `Open`, or something else, depending on the operating system that you use).
5. Selecting the JAR files as described in the previous step you should bring you back to the `Properties` window, and the list of libraries included in your project will now contain all the JAR files you selected. At the bottom of the `Properties` window are two buttons; click on the one labelled `OK`.
6. Make sure to add the appropriate `import` statements to the classes you write. For example, a class which uses the `Task` class needs the following `import` statement:

```
import comp202.winter2009.a5.util.Task;
```

Once you added both `a5-basic.jar` and `junit.jar` to the Classpath of your project by following the above steps, you should now be able to compile all the classes that you are required to write by the above specification without the compiler reporting errors related to missing classes. Note that it is necessary to perform the above steps **ONLY** when the code you want to compile / run requires libraries not included in the Java Standard Library.

Also, note that Eclipse gives you the option of running the provided test programs either as a Java Application, or as a JUnit test. Both result in similar displays, so you use both and decide which one you prefer.

- To run the COMP-202 to-do list manager using the classes you wrote for this assignment from the command line, follow the steps below:

1. Extract the contents of `a5-gui.zip`; it should contain two files: as a JAR file named `a5-gui.jar`, and a Java source file named `COMP202ToDoListManager.java`. Place both files in the same directory / folder as the one in that contains both the files containing the classes that you wrote and `a5-basic.jar`.
2. Open the command-line interface (Command Prompt under Windows, Terminal under Mac OS X, Linux, or other Unix-like operating systems) and navigate normally to the directory / folder containing your files.
3. Compile the application by issuing one of the following commands, depending on your operating system. Under Windows, issue:

```
javac -cp a5-basic.jar;a5-gui.jar;.  COMP202ToDoListManager.java
```

Under Mac OS X, Linux, or another Unix-like operating system), issue:

```
javac -cp a5-basic.jar:a5-gui.jar:.  COMP202ToDoListManager.java
```

4. Launch the application by issuing one of the following commands, depending on your operating system. Under Windows, issue:

```
java -cp a5-basic.jar;a5-gui.jar;.  COMP202ToDoListManager
```

Under Mac OS X, Linux, or another Unix-like operating system), issue:

```
java -cp a5-basic.jar:a5-gui.jar:.  COMP202ToDoListManager
```

5. Plan!

Note that you should not attempt this before you have thoroughly tested the classes you wrote for this assignment.

- To run the COMP-202 to-do list manager using the classes you wrote for this assignment from Eclipse, follow the steps below:

  1. Extract the contents of `a5-gui.zip`; it should contain two files: as a JAR file named `a5-gui.jar`, and a Java source file named `COMP202ToDoListManager.java`
  2. In Eclipse, add `a5-gui.jar` to the Classpath of your project; to do so, perform the normal steps that enable you to add a JAR file to the Classpath of a project, as described above.
  3. Import the file `COMP202ToDoListManager.java` in Eclipse, and run the file normally.
  4. Plan!

  Again, note that you should not attempt this before you have thoroughly tested the classes you wrote for this assignment.

What To Submit

```
ToDoList.java
ToDoListLoader.java
ToDoListWriter.java
```