Homework 5, CS161, Spring 2012, Due Monday, March 5, 5:00 p.m.

The last thing you should do before you turn your program in is to run it from the command line on the department Linux machines.

We have provided starter files called `DateInterface.java`, `Date.java`, `MomentInterface.java`, `MySort.java`, as well as input files `dates.txt` and `moments.txt`.

Study `DateInterface.java` and `Date.java` to see how `Date.java` implements `DateInterface.java`. One of your goals is to use `DateInterface.java` and `Date.java` as a model for how to implement `MomentInterface.java` as `Moment.java`.

When you compile `DateInterface.java`, you may get some "Lint" warnings about `MySort.java` having a loophole. In later classes, you will learn ways of cleaning this up through the use of *generics*. It won't create problems on this assignment.

- At the top of `Date.java`, you will see that it actually implements two interfaces, `DateInterface` and `Comparable`. We have supplied `DateInterface`, but `Comparable` is already provided in Java.
- There are three integers that give the month, day, year of the date. These are private, and accessed only through getters and setters.
- There are also some private static variables that are useful when performing date calculations. The reason for declaring them static is so that all Date objects in a program will share these variables, instead of each carrying their own copy. This reduces the memory requirement of a set of dates in a program. They are also declared `final`, which means that these variables can't be changed. (The variables are references to arrays, so they can't be made to point to a new array location.)
- Next, compare the methods specified in `DateInterface.java` to those in the next section of `Date.java`. Notice that these methods call other methods that are not listed in the `DateInterface.java` file. The compiler will complain if you don't implement all methods in the interface file, but doesn't mind if you put in extra public or private methods.
- Notice the `try` and `catch` blocks in `daysUntil`. Inside the `try` block, if one of the two dates is not a legal date, then the method creates an `Exception` object. The `Exception` class is a class that is available on the system. One of its constructors takes a String error message as a parameter, and this is the constructor we use to create the `Exception` object. The object carries this message and information about the state of the program when it was created.

  It then `throws` the exception. Execution will jump to the `catch (Exception e)`. This receives a reference to the thrown `Exception` object as a parameter, e. It then calls a method from the `Exception` class called `printStackTrace()` it, which results in a printout of some of the information contained in e. This includes the message we passed to the constructor, and which methods were active at the time the exception was created.

  Finally, the statement `System.exit(0)` causes the program to halt. You don't have to halt your program when you catch an exception if you can think of a graceful way

to fix the problem. We chose to halt it. You should handle exceptions the way we did on your program.

Run the `Date` program and give an invalid date to this method to see the output of these steps.

- In the `DateInterface.java` file, you will see that we want to maintain the *invariant* that a date is either a legal date, or 0/0/0, to indicate an illegal date. Look at `setMonth(), setDay()`, and `setYear()`. These are declared as `protected` rather than `private` or `public`. We don't want want to declare them as public, since then a user could mess up our invariant. We wouldn't be able to count on any sanity in our dates. On the other hand, if we make them private, then a class that extends the `Date` class, such as the `Moment` class that you are going to write, would not be able to change their own month, day, and year. By making them `protected`, we give access to them only to methods in the class and to methods in classes that extend the class.

- Notice that the interface file does not specify the constructors. That's because the interface file does not dictate the name of the class that implements it, so it can't know what the constructors will be called. The comments in the Interface file specify the constructors, however.

- Any objects that have a natural ordering can implement the `Comparable` interface. Examples are objects of type `String`, since dictionary order, also known as "lexicographic order" is a natural way to order a set of strings. Ordering dates from earliest to latest is a natural way to order dates.

  Google "java Comparable" to read more about it. The only method in this interface is "compareTo(Object o)". Since the `Object` class is an ancestor of all others, including ones you define, any object can be passed in as the parameter. When you implement this method, the idea is that the object that is passed in is the same type as 'this'.

  Notice that this object is `typecast` to an object of type `DateInterface` with the following statement:

  ```
  DateInterface d2 = (DateInterface) o2;
  ```

  This will cause a runtime error if somebody passes in an object that is from a class that doesn't implement `DateInterface`. Otherwise, you can apply `DateInterface` methods to it to compute the result of `compareTo()`.

  The `Comparable` interface says that `compareTo(Object o)` should return -1 if `this` should go before `o` in the ordering, 0 if they are the same, and 1 if `o` should go before `this` in the ordering.

  Why would you want this? Look in `DateList.java`, and run it on the supplied `dates.txt` file. You can see that it sorts the dates from earliest to latest. We wrote the sorting algorithm that it uses in `MySort.java`. If you look in this file, you see that it uses `Mergesort`. However, instead of operating on integers, it operates on an array of objects that implement the `Comparable` interface, and uses the `compareTo` method in place of the $\leq$ operator.

  However, when `DateList.java` passes an array of `Date` objects to this class, the sorting algorithm doesn't even know what kind of objects it's sorting. It uses the

`compareTo` method for guidance about how to sort them, and there is no mention to the `Date` class anywhere in the file. Therefore, the same program can be used to sort an array of `String` objects, or an array of objects from the `Moment` class that you are going to implement.

1. Your `Moment` class should extend the `Date` class and implement the `MomentInterface` and `Comparable` interfaces. It should start out this way:

   ```
   public class Moment extends Date implements MomentInterface, Comparable
   ```

   It implements a date and a time of day down to the second.

2. In your constructors, you can call the `Date` constructors through `super` to set the date part of the object. Inside your `isLegal()` you can call `super.isLegal()` to check whether the `Date` part of the object is a legal date.

3. The constructors and the `toString()` treat the date as though it is specified in the American AM/PM format. That does not constrain you to represent it that way in the private data for the class.

   Here are two alternatives:

   - Create private variables for hour, minute, second and a boolean that is `true` if the time is P.M. and false if it's A.M.
   - Create private variables for hour, minute, second in what Americans sometimes call "military time." (It's used by civilians in much of the world.) In military time, the hour goes from 0 at midnight through 23 right before midnight.

   Does it matter much which one you use? It doesn't if you always use getters and setters to access the private data. If you use the American representation, then `getPM` and `getHour()` are true getters, and `getHourMilitary()` calculates its returned value. If you use the military format in your private variables, `getPM` and `getHour()` calculate their returned values, and `getHourMilitary()` is a true getter.

   Once you've set this up, with matching setters, it is no longer obvious which are the actual getters and setters, and which ones do some calculations to change one format to the other. You never have to think again about what actual representation you've used; you've hidden it behind your getters and setters, simplifying your thinking as you write the remainder of the methods.

   Pick your format; we only care that you've faithfully implemented the interface.

4. You will find that it is easiest to implement `secondsUntil` before `minutesUntil` and `hoursUntil`. It's pretty easy, given that you have access to `daysUntil` from the `Date` class and that you have the ability to work in military time. You can multiply the days by the number of seconds in a day, and then adjust your answer using the two times of day.

   Since `minutesUntil` needs to round, rather than truncating, as integer arithmetic does, you can add thirty seconds to the result of `secondsUntil` before dividing by 60 to get `minutesUntil`.

Since integer arithmetic always truncates toward zero, you have to subtract thirty seconds instead of adding them if the result of `secondsUntil` is negative.

Apply a similar trick to round to the nearest hour in `hoursUntil`.

5. Create a test harness so that you can thoroughly test all of your methods. You will almost certainly lose points for bugs if you don't give yourself a way to generate a lot of tests for special cases. For example, your `compareTo` might work if the dates are different, but fail when the dates are the same. It might only fail when one of the hours is midnight or noon, or when only the seconds differ.

6. When you are done, create a program called `MomentList` by making a copy of `DateList`, and modify it so that it reads in a file of moments, such as the supplied `moments.txt` file, and then calls `MySort.Sort` to sort them chronologically.

Submit the following:

- Your `Moment.java` file;
- Your `MomentList.java` file;

Don't change anything about the `Date.java` file the interface files, or the `MySort.java` files, since they will be the ones we will work with when testing your programs.