LAB 3 (part B)

Use the project named **`lab3`** that you made in part A for your work here. Framework code and this lab are available on moodle.

Lab 3, both parts, is due on Friday, March 27th, before the end of the day, via Moodle.

1. Sorting dates

Consider the following method that makes use of a **`Date`** class:

```
public static void dateArrayTest () {
    Date[] dArray = new Date[4];
    dArray[0] = new Date(5, 2);  // May 2nd
    dArray[1] = new Date(2, 9);  // Feb 9th
    dArray[2] = new Date(6, 3);  // June 3rd
    dArray[3] = new Date(1, 11); // Jan 11th
    Arrays.sort(dArray);
    for (int k = 0; k < dArray.length; k++) {
        System.out.println(dArray[k]);
    }
    // should print the dates in chronological order:
    // 1/11, 2/9, 5/2, 6/3
}
```

Write the **`Date`** class. You will need a constructor that takes two integers, the first representing the month and the second the day (don't worry about ensuring that the date is valid), as well as the **`toString`** method. And, as you can see from the Javadoc of **`Arrays.sort(Object[])`**, **`Date`** will need to implement the **`Comparable`** interface.

2. Sorting babynames

Use the **`Runner`** class in the **`babynames`** package to load a long list of baby names from a file. (Don't change the name of the file or package or the **`Runner.readnames()`** method might fail to load the file).

The **`Runner.main`** method uses **`Collections.sort(List<T>)`** to sort the names alphabetically, and lists the last name as "Zulma". I knew a Zulma once… Anyway, recall that **`List`** is the interface that specifies the sequence-like behavior that **`ArrayList`**, among other implementation classes, implements. The "**`<T>`**" in the method signature is Java's way of saying this can work for any Object that can be sorted; **`T`** is a wildcard of sorts.

Use the **`Collections.sort(List<T>, Comparator(<? extends T>))`** method to find the *longest name* (or, one of the names with the longest length). As above, your **`ArrayList<String>`** will satisfy the first argument. The "**`<? extends T>`**" in the second argument is Java's way of asking for a **`Comparator`** that can compare instances of the same **`T`**

class as in the first argument: either that class or a derived class from it. In this case, an instance of a class that implements **`Comparator<String>`** is the right thing.

Use an anonymous inner class to pass in the **`Comparator`** that **`Collections.sort`** needs.

Second, use another anonymous inner class to find the name that comes alphabetically last when considered backwards. Remember, an easy way to reverse a **`String`** in java is

```
new StringBuilder(hi).reverse().toString()
```

3. The observer pattern

The observer pattern tries to loosely couple an information source with a class that does something with that information. In Java, the **`Observable`** class is extended to make the source, and the interface **`Observer`** is used when making the listener.

The framework code contains a **`RandomWalker`** class that will, randomly, "walk" left or right. It has a start method that sets it walking a certain number of times, and waits a little bit of time between each step.

You need to write a class **`RandomWalkerObserver`** that reports the first time that a **`RandomWalker`** deviates 10, 20, and 30 steps to either the left or right, and how many steps it took to shift over that far. If the walker never wiggles that far from center, your observer should report nothing. Eg.,

```
The walker reached 10 spaces to the left after 53 steps.
The walker reached 10 spaces to the right after 221 steps.
```

You need to modify the **`RandomWalker`** class to extend **`Observable`**, and add or modify any methods so that it can accept **`Observer`** objects and notify them each time a step is taken.

Your **`RandomWalkerObserver`** class should implement **`Observer`** and do the reporting specified above.

There is a **`Runner`** class in the framework code with a main method you can use to test your code. Note that it builds a **`RandomWalker`** and a **`RandomWalkerObserver`** but keeps the static types of its variables as **`Observable`** and **`Observer`**; this helps enforce that the pattern is separate from the implementation.