CMPT 202
Lab 11 Basic Sorting

Due Wednesday April 22, 2015

In this lab, you will take the sorting algorithms from class and rewrite them in object-oriented classes.

Objectives

The objective of this lab is to

- Further develop object-oriented programming skills using abstract classes.
- Analyze three sorting algorithms.

How Labs are Managed and Graded

1. Select a partner on your own and sign up. The rules are you must pair with a new partner each week.

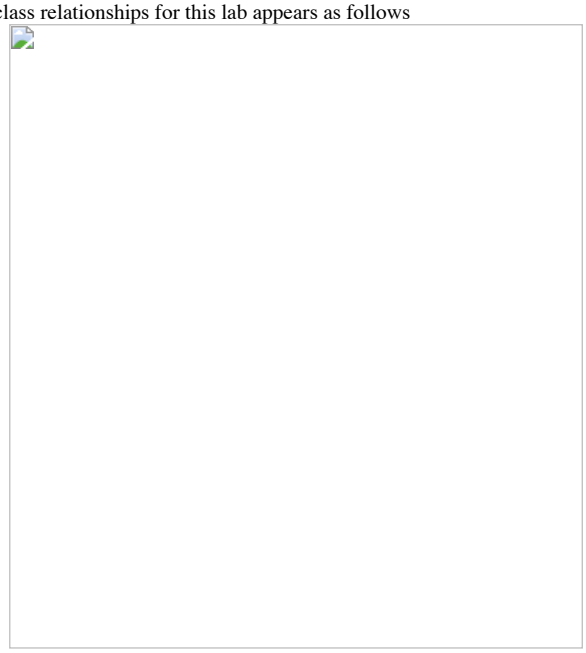
2. Work as a pair to complete the lab. At the end of the lab period, make sure to share all files with each partner.

3. Each partner will submit the lab separately. (Submission instructions are described later in this lab.)

4. If you do not finish the lab during the lab period, you may either get together outside of class to complete it, or complete it on your own. If you choose to complete it on y own, be sure to indicate this in your submission.

UML

The UML class relationships for this lab appears as follows



You may find it helpful to refer back to this UML diagram as you develop this lab.

Abstract Class Review

An abstract class is similar to an interface, except that whereas an interface contains no implemented methods (all methods are considered abstract, and must therefore be defined in the class implementing the interface) an abstract class may implement certain methods. Methods that are not implemented in an abstract class (again, known as abstract methods) must be implemented by a class that extends the abstract class.

> **Big Picture ->** A class that extends an `abstract` class must implement the `abstract` methods defined in the `abstract` parent class. Methods that are defined (non-abstract) are inherited by child classes and are avaialble for re-use.

Step 1: Write an abstract class Algorithm.

The textbook introduces each of the sorting algorithms as `static` methods in separate classes. If we wanted to compare the methods for different sorting algorithms, we would need a lot of repetitive code. This can be avoided by using an object-oriented design.

Begin by using the following abstract `Algorithm` class:

```
public abstract class Algorithm  < T extends Comparable < ? super T > >
```

In this abstract class, include one abstract method named `apply()` that is passed a T array and applies an algorithm to it (in this case it will be some type of sorting algorithm). It should not return anything. Its signature appears as follows:

```
public abstract void apply(T[] array);
```

Next, create a method called `time()`, which also takes a `T` array and returns the number of milliseconds required to invoke the `apply()` method :

```
public long  time(T[] array) {
    long start, end;
      start = System.currentTimeMillis();

      // invoke the apply method
      this.apply(array);

      end = System.currentTimeMillis();

      // returns elapsed time
      return  (end - start);
}
```

Step 2: Add a static method to the Algorithm Class

In this `Algorithm` class, we are going to add a static method to make it easier to compare several sorting algorithms.

Write a method `createRandomArray()` that will take an int (n) as an argument and create and return an array of n random Integers to be sorted. The array should be of type Integer, and the values should be a random integers. Its signature appears as

```
/**
 * Create an array containing n random Integers.
 */
public static Integer[] createRandomArray(int n)
```

Look at the Java API - in particular the `java.util.Random` class for generating random integers.

Only proceed if your `Algorithm` class has no compile errors.

Step 3: Write a SortingAlgorithm Class

This abstract class will be a child class of the `Algorithm` class as it `extends Algorithm` .

```
public abstract class SortingAlgorithm <T extends Comparable <? super T>> extends Algorithm <T>
```

Add the following abstract method to this class:

```
/**
 * sorts the parameter array in ascending order (from smallest to largest)
 */
public abstract void sort (T[] array);
```

This `sort()` method will ultimately invoke the appropriate sorting algorithm (i.e. selection, insertion, merge sort, etc.) This abstract method is an example of polymorphism and abstraction at work - the method defines its behavior that it sorts the array, but does not specify which sorting algorithm is to be used.

Additionally, now define the abstract `apply()` method from the `Algorithm` class so that it invokes the `sort()` method:

```
/**
 * Invoke the appropriate sorting algorithm.
 */
public void apply(T[] array) {
    this.sort(array);
}
```

The `apply()` method implements `apply()` from the parent abstract class `Algorithm`. As you can see, the purpose of `apply()` is to merely invoke the `sort()` method.

Next, write a method `isSorted()`, that will take an array of `Comparable` objects as an argument and return true if it is sorted, and false otherwise. The signature for this method should be:

```
public static <T extends Comparable<? super T>> boolean isSorted(Comparable[] array)
```

An array is considered sorted if every element at position *i* in the array has the following relationship:

```
array[i-1] <= array[i]
```

You should use the compareTo method to check that the array is sorted. You will need this `isSorted()` method to confirm your sorting algorithms are working correctly.

Below is a unit test class for testing `isSorted()`. It also has a test for the `SelectionSort` , `InsertionSort` and `MergeSort` classes (for the next step). Download these unit tests, comment out `testSelectionSort()` , `testInsertionSort()` and `testMergeSort()` and make sure your pass the `testIsSorted()` test.

- SortingTest.java

OK, Let's Take a Quick Breather

At this point you have defined an abstract class `Algorithm` that allows you to create a random array, as well as time a sorting algorithm. However, because it is abstract, there are no specifics, and it can be used with any sorting algorithm.

You also defined an abstract class `SortingAlgorithm` that specifies an abstract method `sort()` - this abstract method is called from the `apply()` method. Again, because `SortingAlgorithm` is also abstract, it is not defined in terms of any specific sorting algorithm.

So far your two abstract classes are, well, abstract; they can work with any specific sorting algorithm. That is what you will cover in the next steps.

Step 4: Rewrite Selection Sort as a SortingAlgorithm Class

Create a class called `SelectionSort.java` that is a child class of the `SortingAlgorithm` class. The header of this class will appear as follows:

```
public class SelectionSort <T extends Comparable<? super T>> extends SortingAlgorithm <T>
```

Now take the Selection Sort from class notes on sorting and use code in this class to implement the `SelectionSort` class.

As it extends `SortingAlgorithm`, it must implement the `sort()` method. This `sort()` method can call the `static selectionSort()` method that you adopted from `GenericComparablesSelectionSort` class.

Move the `swap()` method to the `SortingAlgorithm` class (because it may be useful to other `SortingAlgorithm` subclasses). The rest of the class should not require any other modifications.

Test your new `SelectionSort` class by running the unit tests `SortingTest` - be sure to uncomment `testSelectionSort()`

Step 5: Implement Insertion Sort as a SortingAlgorithm Class

Create a class InsertionSort that is a child class of the `SortingAlgorithm` class. The header of this class will appear as follows:

```
public class InsertionSort <T extends Comparable<? super T>> extends SortingAlgorithm <T>
```

Implement the insertion sort algorithm that we covered in class notes on sorting. This implementation will look quite similar to the `SelectionSort` class except that it will have a method named `insertionSort()` .

Test your new `InsertionSort` class by running the unit tests `SortingTest` - be sure to uncomment `testInsertionSort()`

Step 6: Include Merge Sort

The link below contains an implementation of the merge sort algorithm. It has been provided for you and you will use it to contrast results with the selection and insertion sorts.

- MergeSort.java

Even though you are being supplied this code, it is worthwhile ensuring that it works properly. Uncomment `testMergeSort()` and make sure the unit test passes.

Step 6: Compare Different Sorting Algorithms

At this point, you have three different sorting algorithms: (1) Selection Sort, (2) Insertion Sort, and (3) Merge Sort. From class we know that Selection and Insertion are $O(N^2)$ algorithms, while Merge is O(N lg N). In this step you will compare timing results of these three different algorithms.

Write a `compareSorts()` method in the `SortingAlgorithm` class that is passed a value `n`. The signature appears as follows:

```
public static void compareSorts(int n)
```

This method will create an array of `Comparable` objects of size `n`, create a copy of that array, and then sort the three copies (details below), using each of the sorting algorithms.

To simplify the code for sorting the array copies, you should create an array of the sorting algorithms, such as:

```
SortingAlgorithm[] sorts = new SortingAlgorithm[3];

  sorts[0] = new SelectionSort();
  sorts[1] = new InsertionSort();
  sorts[2] = new MergeSort();
```

You should also create a two dimensional array (`sortArray`) of size 3. Create a randomly generated array (using the `createRandomArray()` method in your `Algorithm` class) and store it in `sortArray[0]`. Then, copy it to `sortArray[1]` as well as `sortArray[2]` (The reason behind copying the array is that we want to test each sorting algorithm using identical arrays.) You may find the following code helpful:

```
Integer[][] sortArray = new Integer[3][n];
sortArray[0] = createRandomArray(n);

System.arraycopy(sortArray[0], 0, sortArray[1], 0, sortArray[0].length);
System.arraycopy(sortArray[0], 0, sortArray[2], 0, sortArray[0].length);
```

After you have an array of sorting algorithms and a second two dimensional array to sort, use a for loop to cycle through all your sorts:

```
for (int i=0; i < sorts.length; i++) {
          sorts[i].time(sortArray[i]);
          // anything you need to do to report statistics
  }
```

Step 7: Compare the three sorting algorithms

Create a new class named `TimeSorts` and have the `main()` method compare the elapsed time for sorting random arrays of various sizes (suggestions include 25000, 50000, 75000, and 100000 - although you can certainly go higher than this) by calling `compareSorts()` multiple times. You can either add a for-loop in your main method so it creates several different sized arrays (better), or run your program multiple times and copy and paste the timing results for comparison.

Is there a consistent "winner" and a consistent "loser"?

Go to this Google form and enter your results for 25000, 50000, 75000, and 100000 .

Lab Handin

Upload to Angel:

Zip the following files together:

- Algorithm.java
- SortingAlgorithm.java
- SelectionSort.java
- InsertionSort.java
- TimeSorts.java

As always, use good programming style for your code. At a minimum, this includes

- using meaningful variable names
- using proper indentation
- including a comment block before each class file that describes the class and includes your name and the assignment
- including a comment block before each method that describes what the method does.

Don't forget to rate your pair-programming partner!