CSE 143, Winter 2011
Programming Assignment #2: SortedIntList (40 points)
Due Thursday, January 20, 2011, 11:30 PM

This program focuses on implementing a collection and inheritance. Turn in two files named `SortedIntList.java` and `SortedIntListTest.java` from the Homework section of the course web site. You will also need the support file `ArrayIntList.java` from the Homework section of the course web site; place it in the same folder as your program.

For this assignment you are to **write a class called `SortedIntList`** that is a variation of the `ArrayIntList` class written in lecture. Your class will be an extension (a subclass) of the original `ArrayIntList` that adds new functionality and modifies some existing functionality. Your class has two primary differences from the original list:

- A `SortedIntList` must maintain its list of integers in **sorted order** (non-decreasing).
- A `SortedIntList` has an option to specify that its elements should be unique (**no duplicates**).

The `ArrayIntList` class already has the following constructors and methods (along with a few others):

`public ArrayIntList()`	constructs an empty list of a default capacity (10)
`public ArrayIntList(int capacity)`	constructs an empty list with given capacity
`public void add(int value)`	appends given value to end of list
`public void add(int index, int value)`	inserts given value at given index, shifting subsequent values right
`public int get(int index)`	returns the element at the given index
`public int indexOf(int value)`	returns index of first occurrence of given value (< 0 if not found)
`public void remove(int index)`	removes value at given index, shifting subsequent values left
`public int size()`	returns current number of elements in the list
`public String toString()`	returns a string version of the list, such as `"[4, 5, 17]"`

The new class should extend `ArrayIntList`, adding a few new methods/constructors, and overriding some existing methods to modify/improve their functionality. Do not modify the contents of `ArrayIntList.java`. You will write:

`public` **`SortedIntList`**`()`	constructs an empty list of a default capacity, allowing duplicates
`public` **`SortedIntList`**`(boolean unique)`	constructs empty list of default capacity and given "unique" setting
`public` **`SortedIntList`**`(int capacity)`	constructs an empty list with given capacity, allowing duplicates
`public` **`SortedIntList`**`(boolean unique, int capacity)`	constructs an empty list with given capacity and "unique" setting
`public void` **`add`**`(int value)`	possibly adds given value to list, maintaining sorted order
`public void` **`add`**`(int index, int value)`	always throws an `UnsupportedOperationException`
`public boolean` **`getUnique`**`()`	returns whether only unique values are allowed in the list
`public int` **`indexOf`**`(int value)`	same behavior as before, but optimized; see next page
`public int` **`max`**`()`	returns the maximum integer value stored in the list (throws a `NoSuchElementException` if the list is empty)
`public int` **`min`**`()`	returns the minimum integer value stored in the list (throws a `NoSuchElementException` if the list is empty)
`public void` **`setUnique`**`(boolean unique)`	sets whether only unique values are allowed in the list; if set to `true`, immediately removes any existing duplicates from the list
`public String` **`toString`**`()`	returns a string version of the list, such as `"S:[4, 5, 17]U"`

Your class will have a field to keep track of whether or not it is limited to unique values. The value can be set to your second constructor or by calling `setUnique`. Think of it as an on/off switch that each list has.

You will override the `add` method to ensure that elements are added in sorted order, and you will override the `indexOf` method to be more efficient by taking advantage of the list's sorted order. You will also override `toString`. All other methods inherited from `ArrayIntList` should use the default inherited behavior and should not be overridden.

Implementation Details:

`public SortedIntList()`
This constructor should initialize a list of default capacity (10) with uniqueness set to `false` (duplicates allowed).

`public SortedIntList(boolean unique)`
This constructor should initialize a list of default capacity (10) with uniqueness set to the given value.

`public SortedIntList(int capacity)`
This constructor should initialize a list with the given capacity and with uniqueness set to `false` (duplicates allowed). If the capacity is negative, an `IllegalArgumentException` should occur. `ArrayIntList`'s constructor does so as well.

`public SortedIntList(boolean unique, int capacity)`
This constructor should initialize a list with the given capacity and with the given setting for whether or not to limit the list to unique values (`true` means no duplicates, `false` means duplicates are allowed). See `get/setUnique` below. If the capacity is negative, an **`IllegalArgumentException`** should occur. `ArrayIntList`'s constructor does so as well.

`public void add(int value)`
Your single-argument `add` method should be overridden to ensure a sorted list. You should no longer add at the end of the list; instead you should add the value at an appropriate place to keep the list **in sorted order**. For example, if the list is [-3, 7, 18, 42] and the user adds 27, afterward the list should contain [-3, 7, 18, **27**, 42] in that order. (See `indexOf`.)

The `add` method has to pay attention to whether the client has requested **unique values only**. If so, your method must not allow any element to be added if that value is already in the list. You should not re-sort the entire list every time an element is added. Finding the correct index and inserting the element there is more efficient than re-sorting the whole list.

`public void add(int index, int value)`
For a sorted list, it does not make sense to allow the client to insert an element at any particular index. The elements should be ordered so that the list will remain sorted. Therefore, you do not want this method in your `SortedIntList` class; but you must inherit such a method because you extend `ArrayIntList`. The best we can do is to "disable" this method by overriding it to always throw an `UnsupportedOperationException`. The list should not be modified.

`public boolean getUnique()`
This method should return the current uniqueness setting (`true` means no duplicates, `false` means duplicates allowed).

`public void setUnique(boolean value)`
Allows client to set whether to allow duplicates in the list (`true` means no duplicates, `false` means duplicates allowed).

If the unique switch is set to off (`false`), the list allows any integer to be added, even if that integer is already found in the list (a duplicate). If the unique switch is on (`true`), any call to `add` that passes a value already in the list has no effect. In other words, when the unique switch is `true`, the `add` method should not allow any duplicates to be added to the list. For example, if you start with an empty list that has the unique switch off, adding three 42s will generate the list [42, 42, 42]. But if your list has the unique switch on, adding those same three 42s would generate the single-element list [42].

If the client sets unique to `true` when the list has duplicates, `setUnique` should **remove all duplicates** and ensure that no future duplicates can be added unless the client sets unique back to `false`. If the client changes the unique setting to `false`, the list elements do not immediately change, but it will mean that duplicates could be added in the future.

`public int max()`
`public int min()`
These methods should return the largest and smallest element values contained in the list, respectively. For example, in the list [4, 4, 17, 39, 58], the max is 58 and the min is 4. If the list is empty, it has no largest or smallest element, so you should throw a **`NoSuchElementException`**.

`public String toString()`
This method should return a comma-separated string of the list's elements, **preceded by `"S:"`**. If the list's "unique" switch is on, the string should **end with `"U"`**. For example, if the elements are [4, 4, 17, 39, 58] and unique is off, return `"S:[4, 4, 17, 39, 58]"`. If the elements are [-3, 5, 15] and unique is on, return `"S:[-3, 5, 15]U"`. Note the `U`.

```
public int indexOf(int value)
```

This method should be overridden to take advantage of the fact that the list is sorted. It should use the faster **binary search** algorithm rather than the sequential search used in `ArrayIntList`. If the element is found in the list, your `indexOf` should return its position. The element might occur in the list multiple times; if so, you may return any index at which that element value appears. If the element is not found in the list, your method should return a negative number.

In section we will discuss how binary search is implemented, but you should **not** try to implement binary search yourself. Use the built-in `Arrays.binarySearch` method for all index location searching ("Where is a value currently located?" "Where should I insert a new value?"). You can find its documentation in the Java API from the Links section of the course web site. For example, to search indexes 0-16 of an array called `data` for values 42 and 66, you could write:

```
// index         0  1  2   3   4   5   6   7   8   9  10  11  12  13  14  15   16  17  18
int[] data = {-4, 2, 7, 10, 15, 20, 22, 25, 30, 36, 42, 50, 56, 68, 85, 92, 103,  0,  0};
int index1 = Arrays.binarySearch(data, 0, 17, 42);    // index1 is 10
int index2 = Arrays.binarySearch(data, 0, 17, 66);    // index2 is -14
```

When `Arrays.binarySearch` is unable to find a value in the list, it returns a negative number one less than the index at which the value *would* have been found if it had been in the array (sometimes called the **"insertion point"**). For example, in the example above the value 66 is not found, but if it had been in the list, it would have been at index 13; therefore the search returns -14. You should take advantage of this information when adding new elements to your list.

To get access to the `Arrays` class, you should `import java.util.*;` at the beginning of your class.

Testing Program (`SortedIntListTest.java`):

Along with your `SortedIntList`, turn in a short **JUnit testing file** named `SortedIntListTest.java`. This program should be a JUnit test case that tests your sorted int list by creating at least two lists, adding some elements to them, calling various methods, and checking the expected results. This part of the assignment is worth only a few points, but we want you to practice testing your own code. For full credit, your testing file must contain at least 2 test methods, and you must call at least 3 of the different methods on the list(s) you create.

There are also several provided JUnit testing programs on the course web site. You may ask, "Why do I need to write my own testing program when these testing programs are already provided?" Our testing programs are large and complex. It can be good to have a smaller test that runs just a few methods that you are working on. We encourage you to add to this program as you develop your `SortedIntList`. If you want to write a more complex test, that is fine, but not required.

Development Strategy:

We suggest that you develop the program in the following three stages:

1. Write a first version that always allows duplicates and doesn't worry at all about the issue of unique values. Just keep your list in sorted order and use binary search to speed up searching. This stage involves the following:
    a. Create your class file and the overall class header.
    b. Write the constructors.
    c. Modify the first `add` method so that it preserves sorted order. You will need to use binary search here if your solution involves two steps (first locate, then insert).
    d. Modify `indexOf` so that it uses the binary search method; also write `max` and `min`.
2. Modify your code so it keeps track of whether the client wants only unique values. Add any state necessary and modify constructor(s) as needed. Next modify `add` so that it doesn't add duplicates if the unique setting is `true`.
3. Write the `getUnique` and `setUnique` methods. Remember that if the client calls `setUnique` and sets the value to `true`, you must remove any duplicates currently in the list.

We will provide JUnit testing code for each of these 3 stages. For this program only, you are allowed to discuss how to write testing code with other students. Keep in mind that our tests are not guaranteed to be exhaustive. In particular, you will want to do a lot of simpler testing before you try running any of these high-powered tests that really push the limits.

The provided testing code does not test every possible situation, and we do not guarantee full credit for passing the provided tests. But it will give you some good examples of the kind of testing code we want you to write.

References:

Textbook Chapter 15 covers the implementation of `ArrayIntList` in detail. You may want to read it to make sure you understand the class you are extending. Making good use of `ArrayIntList`'s behavior is important for this assignment.

Textbook sections 13.1 and 13.3 discuss the binary search algorithm in more detail. 13.3 discusses how it is implemented and its return values in detail. Seeing how the search is implemented may help you understand how to solve the problem.

Style Guidelines and Grading:

You may not use any features from Java's collection framework on this assignment, such as `ArrayList` or other pre-existing collection classes. You also may not use the `Arrays.sort` method to sort your list.

A major focus of our style grading is **redundancy**. As much as possible you should avoid redundancy and repeated logic within your code, such as by factoring out common code from `if/else` statements, creating "helper" methods to capture repeated code, or having some of your methods/constructors call others if their behaviors are related. Any additional methods you add to `SortedIntList` beyond those specified should be `private` so that outside code cannot call them. Note that even constructors can be redundant; if two or more constructors in your class (or the superclass) contain similar or identical code, you should find a way to reduce this redundancy by making them call each other as appropriate.

Another way you should avoid redundancy is by **utilizing the behavior you inherit from the `ArrayIntList` superclass**. You should not re-implement any `ArrayIntList` behavior from that is not modified in your class. Also, if part of your class's new behavior can make use of the superclass's behavior, you should call constructors/methods from the superclass. For example, the `ArrayIntList` already contains code for adding and removing elements to the list at a specific index, so if your `SortedIntList` code needs to do those things, you should not re-implement that functionality. Recall that you can access overridden behavior from a superclass using the **`super` keyword** as shown in lecture.

Properly **encapsulate** your objects by making any data fields in your class `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used within a single call to one method. Fields should always be initialized inside a constructor or method, never at declaration.

You should follow good **general style** guidelines such as: appropriately using control structures like loops and `if/else` statements; avoiding redundancy using techniques such as methods, loops, and `if/else` factoring; properly using indentation, good variable names, and proper types; and not having any lines of code longer than 100 characters.

Commenting will be more of a style focus on this program (and future programs) than it was on the last assignment. You should comment your code with a heading at the top of your class with your name, section, and a description of the overall program. Also place a comment heading atop each method, and a comment on any complex sections of your code.

Not only will we look for the existence of comments on this assignment, but we also have stricter expectations about the quality of their content. Comment headings should use descriptive complete sentences and should be written in your own words, explaining each method's behavior, parameters, return values, and pre/post-conditions as appropriate. If the method potentially throws any exceptions, comment this and explain what exceptions it throws and under what conditions it will throw them. Be concise but specific in your comments. The `ArrayIntList` class, along with other programs from lecture and section this week, are good examples of appropriate commenting style for this assignment. (Your JUnit test program should have an overall descriptive comment header atop the class, but it does not need commenting on each testing method or otherwise throughout the code.)

For reference, our `SortedIntList.java` is around **120 lines long** including comments (and has **45 "substantive" lines** according to our Indenter tool on the course web site). But you do not have to match this; it's just listed as a sanity check.