```
 1
 2 import java.util.*;
 3 /**
 4  * An Integer Specific data type which sort itself.
 5  *
 6  * <ul>
 7  * <li> Name: SortedIntList.java
 8  * <li> Description: Sorted Int List
 9  * <li> Class: Java 145
10  * <li> Instructor: Ken Hang
11  * <li> Date: Jan 28 2015
12  * </ul>
13  *
14  * @author  Hai H Nguyen (Bill)
15  * @version Winter 2015
16  */
17 public class SortedIntList {
18
19     public static final int DEFAULT_CAPACITY = 99;
20
21     private int[] elementData;  // list of integers
22
23     private int size;           // current number of elements in the list
24
25     private boolean unique;     // indicate whether there should be duplications or not
26
27     /**
28      * Full Constructor, take a capacity and an indicator, then call Main Constructor.
29      * @param unique        Indicate whether there should be duplications or not
30      * @param capacity      Replacement for the DEFAULT_CAPACITY
31      */
32     public SortedIntList(boolean unique, int capacity){
33         this(capacity);
34
35         this.unique = unique;
36     }
37
38     /**
39      * Main Constructor, take in a capacity and instantiate an array based on it.
40      * @param capacity      Replacement for the DEFAULT_CAPACITY
41      */
42     public SortedIntList(int capacity){
43         if (capacity < 0) {
44             throw new IllegalArgumentException("Invalid capacity: " + capacity);
45         }
46
47         elementData = new int[capacity];
48
49         size = 0;
50     }
51
52     /**
53      * Sub Constructor, take an indicator, call Full Constructor with DEFAULT_CAPACITY
54      * @param unique        Indicate whether there should be duplicates or not
55      */
56     public SortedIntList(boolean unique) {
57         this(unique, DEFAULT_CAPACITY);
58     }
59
60     /**
61      * Default Constructor, Call the Main Constructor with DEFAULT_CAPACITY
62      */
63     public SortedIntList(){
64         this(DEFAULT_CAPACITY);
65     }
66
67     /**
68      * @return          The maximum value from the List
```

```
 69      */
 70     public int max(){
 71         if (size == 0){
 72             throw new NoSuchElementException("Size: " + size);
 73         }
 74 
 75         return elementData[size-1];
 76     }
 77 
 78     /**
 79      * @return          The minimum value from the List
 80      */
 81     public int min(){
 82         if (size == 0){
 83             throw new NoSuchElementException("Size: " + size);
 84         }
 85 
 86         return elementData[0];
 87     }
 88 
 89     /**
 90      * @return          The value of the Unique Flag
 91      */
 92     public boolean getUnique(){
 93         return unique;
 94     }
 95 
 96     /**
 97      * Set the Value of the Unique Flag
 98      * @param value     The Flag to switch to
 99      */
100     public void setUnique (boolean value){
101         if ((unique = value) && size > 1){
102             removeDuplicates();
103         }
104     }
105 
106     /**
107      * @param index     Index to get Value
108      * @return          Value at Index
109      */
110     public int get(int index) {
111         checkIndex(index);
112 
113         return elementData[index];
114     }
115 
116     /**
117      * @return          The Size Field
118      */
119     public int size() {
120         return size;
121     }
122 
123     /**
124      * Add the passed value into the elementData.
125      * @param value     Element to be added
126      */
127     public void add(int value){
128         int index = indexOf(value);
129 
130         if (!(index >= 0 && unique)){
131             ensureCapacity(size + 1);
132 
133             if (index == -1){
134                 index = getInsertIndex (value);
135             }
136 
```

```
137             insert(index,value);
138         }
139     }
140 
141     /**
142      * @param value     Value looking for index
143      * @return          Return the index to Insert the passed Value
144      */
145     private int getInsertIndex(int value){
146         return -(Arrays.binarySearch(elementData, 0, size, value)+1);
147     }
148 
149     /**
150      * Insert the value at the index indicated
151      * @param index     Index to insert the value
152      * @param value     Value to be inserted
153      */
154     private void insert(int index, int value){
155         for (int i = size; i > index; --i) {
156             elementData[i] = elementData[i - 1];
157         }
158 
159         elementData[index] = value;
160 
161         ++size;
162     }
163 
164     /**
165      * Remove all Duplicates
166      */
167     public void removeDuplicates(){
168         int previousSize = size + 1;    // Ensure Space
169 
170         int [] tempData = new int[previousSize];
171 
172         size = 1;
173 
174         tempData[0] = elementData[0];
175 
176         for (int i = 0 ; i < previousSize-1; ++i){
177             if (tempData[size-1] != elementData[i]){
178                 tempData[(++size)-1] = elementData[i];
179             }
180         }
181 
182         elementData = tempData;
183     }
184 
185     /**
186      * Remove the element at passed index
187      * @param index     Index of Element to be Deleted
188      */
189     public void remove(int index) {
190         checkIndex(index);
191 
192         --size;
193 
194         for (int i = index; i < size; i++) {
195             elementData[i] = elementData[i + 1];
196         }
197     }
198 
199     /**
200      * Set the Size to 0, thus negate the use of other functions.
201      */
202     public void clear() {
203         size = 0;
204     }
```

```
205
206     /**
207      * @param value     The value to check for index
208      * @return          Index of the value given. Else return -1
209      */
210     public int indexOf(int value) {
211         int index = Arrays.binarySearch(elementData, 0, size, value);
212
213         if (index >= 0){
214             return index;
215         } else {
216             return -1;
217         }
218     }
219
220     /**
221      * @param value     The value to check for existence
222      * @return          True if elementData contains it, False otherwise
223      */
224     public boolean contains(int value) {
225         return indexOf(value) >= 0;
226     }
227
228     /**
229      * Check if index is within the size
230      * @param index     Index to be checked
231      */
232     private void checkIndex(int index) {
233         if (index < 0 || index >= size) {
234             throw new IndexOutOfBoundsException("index: " + index);
235         }
236     }
237
238     /**
239      * @return      True if size is 0, False otherwise.
240      */
241     public boolean isEmpty() {
242         return size == 0;
243     }
244
245     /**
246      * Check the capacity of the current list. If needed, double it.
247      * @param capacity      Passed size to check
248      */
249     public void ensureCapacity(int capacity) {
250         if (capacity > elementData.length) {
251             int newCapacity = elementData.length * 2 + 1;
252
253             if (capacity > newCapacity) {
254                 newCapacity = capacity;
255             }
256
257             elementData = Arrays.copyOf(elementData, newCapacity);
258         }
259     }
260
261     /**
262      * @return          Comma-separated, bracketed version of the list
263      */
264     public String toString() {
265         if (size == 0) {
266             return "[]";
267         } else {
268             String out = "[" + elementData[0];
269
270             for (int i = 1; i < size; i++) {
271                 out += ", " + elementData[i];
272             }
```

```
273
274             out += "]";
275
276             return out;
277         }
278     }
279 }
```