

```
1 package adt;
2
3 import entity.Tutor;
4 import java.io.Serializable;
5 import java.util.Comparator;
6 import java.util.Iterator;
7
8 /**
9  *
10  * @author Chew Lip Sin
11  * @author Lim Yi Leong
12  * @author Eugene Teoh
13  * @param <T> type of elements stored in the stack.
14  */
15 public class ArrList<T> implements ListInterface<T>, Serializable {
16
17     private T[] arr;
18     private int numberOfEntries;
19     private static final int DEFAULT_CAPACITY = 20;
20
21     /**
22      * Constructs a new list with the default capacity.
23      */
24     public ArrList() {
25         this(DEFAULT_CAPACITY);
26     }
27
28     /**
29      * Constructs a new list with the specified initial capacity.
30      *
31      * @param initialCapacity The initial capacity of the list.
32      */
33     public ArrList(int initialCapacity) {
34         numberOfEntries = 0;
35         arr = (T[]) new Object[initialCapacity];
36     }
37
38     /**
39      * Adds the specified element to the end of the list.
40      *
41      * @param newEntry The element to add.
42      * @return true if the addition is successful, or false if the list is full
43      * Description: Adds a new entry to the end of the list. Entries currently
44      * in the list are unaffected. The lists size is increased by 1.
45      * Precondition: newEntry is not null. Post-condition:The entry has been
46      * added to the list.
```

```
47     *
48     */
49     @Override
50     public boolean add(T newEntry) {
51         if (isArrayFull()) {
52             //double the array list if array list is full
53             doubleArray();
54         }
55         arr[numberOfEntries] = newEntry;
56         numberOfEntries++;
57         return true;
58     }
59
60     /**
61     * Adds the specified element to the list at the specified position.
62     *
63     * @param newPosition The position to add the element at.
64     * @param newEntry The element to add.
65     * @return true if the element was added successfully, false otherwise. *
66     * Description: Adds a new entry at a specified position within the list.
67     * Entries originally at and above the specified position are at the next
68     * higher position within the list. The list size is increased by 1.
69     * Precondition: newPosition >= 1 and newPosition smaller equal than
70     * getLength()+1newEntry is not null. Post-condition:newEntry is added to
71     * the list in the given position. The old entries have been shifted up one
72     * position.
73     */
74     @Override
75     public boolean add(int newPosition, T newEntry) {
76         boolean isSuccessful = true;
77
78         if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1)) {
79             if (isArrayFull()) {
80                 //double the array list if array list is full
81                 doubleArray();
82             }
83             makeRoom(newPosition);
84             arr[newPosition - 1] = newEntry;
85             numberOfEntries++;
86         } else {
87             isSuccessful = false;
88         }
89
90         return isSuccessful;
91     }
92
```

```
93     /**
94      * Adds all of the elements in the specified array to the end of the list.
95      *
96      * @param newElements The array of elements to add.
97      * @return true if all of the elements were added successfully, false
98      * otherwise. Precondition:newElements must not be null. Post-condition:
99      */
100    @Override
101    public boolean addAll(T... newElements) {
102        if (newElements != null) {
103            if (isElementsValid(newElements)) {
104                for (T element : newElements) {
105                    add(element);
106                }
107                return true;
108            }
109        }
110        return false;
111    }
112
113    /**
114     * Post-condition:The list is empty. Description:Removes all entries from
115     * the list.
116     */
117    @Override
118    public void clear() {
119        numberOfEntries = 0;
120    }
121
122    /**
123     * Checks whether the list contains the specified element.
124     *
125     *
126     * @param anEntry The element to check for.
127     * @return true if the list contains the element, false otherwise.
128     * Description: This method finds whether the new Entry exists or not.
129     * Precondition: The array must exist. Post-condition:The array remains
130     * unchanged
131     *
132     *
133     */
134    @Override
135    public boolean contains(T anEntry) {
136        boolean found = false;
137        //!found = true and index should smaller than the length of array list
138        for (int index = 0; !found && (index < numberOfEntries); index++) {
```

```
139         if (anEntry.equals(arr[index])) {
140             found = true;
141         }
142     }
143     return found;
144 }
145
146 /**
147  * This method is used to retrieve the entry at a given position in the
148  * list.
149  *
150  * @param givenPosition The position of the element to get.
151  * @return a reference to the indicated entry or null, if either the list is
152  * empty, givenPosition smaller 1, or givenPosition bigger getLength()
153  * Precondition:The array must exist. Post-condition:The array remains
154  * unchanged.
155  */
156 @Override
157 public T getEntry(int givenPosition) {
158     T result = null;
159
160     if ((givenPosition >= 1) && (givenPosition <= numberOfEntries)) {
161         result = arr[givenPosition - 1];
162     }
163     return result;
164 }
165
166 /**
167  * Gets the number of entries currently in the list.
168  *
169  *
170  * @return The number of entries currently in the list. Precondition:The
171  * array must exist. Post-condition:The array remains unchanged.
172  */
173 @Override
174 public int size() {
175     return numberOfEntries;
176 }
177
178 /**
179  * This method check if the array is empty
180  *
181  * @return true if the list is empty, false otherwise. * Post-condition:The
182  * array remains unchanged.
183  *
184  */
```

```

185     @Override
186     public boolean isEmpty() {
187         return numberOfEntries == 0;
188     }
189
190     /**
191     * Removes the element at the specified position in the list.
192     *
193     * @param givenPosition The position of the element to remove.
194     * @return The element that was removed, or null if the position is invalid.
195     */
196     @Override
197     public T remove(int givenPosition) {
198         T result = null;
199
200         //the number enter by user must between 1 and the length of the array list
201         if ((givenPosition >= 1) && (givenPosition <= numberOfEntries)) {
202             result = arr[givenPosition - 1];
203
204             //shift the existing entries if the entry removed is not located at the last
entry
205             if (givenPosition < numberOfEntries) {
206                 removeGap(givenPosition);
207             }
208
209             //length should minus 1 after removing a entry
210             numberOfEntries--;
211         }
212         return result;
213     }
214
215     /**
216     * Removes all occurrences of the specified elements from the list.
217     *
218     * @param elements The elements to be removed.
219     * @return {@code true} if removal is successful, {@code false} if the list
220     * is empty or elements are invalid.
221     */
222     @Override
223     public boolean removeAll(T... elements) {
224         if (isEmpty() || !isElementsValid(elements)) {
225             return false;
226         } else {
227             for (T element : elements) {
228                 remove(element);
229             }

```

```
230         return true;
231     }
232 }
233
234 /**
235  * Replaces the entry at the specified position with the new entry.
236  *
237  * @param givenPosition The position of the entry to be replaced.
238  * @param newEntry The new entry to replace the existing entry.
239  * @return {@code true} if replacement is successful, {@code false} if the
240  *         list is empty, or position is invalid.
241  */
242 @Override
243 public boolean replace(int givenPosition, T newEntry) {
244     boolean isSuccessful = true;
245
246     if ((givenPosition >= 1) && (givenPosition <= numberOfEntries)) {
247         //-1 because the givenPosition will only start with 1, does not like
248         // index of array list that start with 0.
249         arr[givenPosition - 1] = newEntry;
250     } else {
251         isSuccessful = false;
252     }
253     return isSuccessful;
254 }
255
256 /**
257  * Checks if the array is full.
258  *
259  * @return {@code true} if the array is full, {@code false} otherwise.
260  */
261 @Override
262 public boolean isFull() {
263     return false;
264 }
265
266 /**
267  * Removes the first occurrence of the specified entry from the list.
268  *
269  * @param anEntry The entry to be removed.
270  * @return {@code true} if removal is successful, {@code false} if the entry
271  *         is not found.
272  */
273 @Override
274 public boolean remove(T anEntry) {
275     boolean isSuccessful = false;
```

```
276
277     //if the array list is not empty
278     if (!isEmpty()) {
279         for (int i = 0; i < numberOfEntries; i++) {
280             if (arr[i].equals(anEntry)) { //compare the given entry and
281 //                every entry in the array list,
282 //                // if true then go in and remove the given entry
283                 removeGap(i + 1);
284                 isSuccessful = true;
285                 numberOfEntries--;
286             }
287         }
288     }
289
290     return isSuccessful;
291 }
292
293 /**
294  * Checks if the array is full.
295  *
296  * @return {@code true} if the array is full, {@code false} otherwise.
297  */
298 private boolean isArrayFull() {
299     return arr.length == numberOfEntries;
300 }
301
302
303 /**
304  * Doubles the size of the array.
305  */
306 private void doubleArray() {
307     T[] oldArray = arr;
308     arr = (T[]) new Object[2 * oldArray.length];
309     System.arraycopy(oldArray, 0, arr, 0,
310         numberOfEntries);
311 }
312
313 /**
314  * Returns a string representation of the list.
315  *
316  * @return A string representation of the list.
317  */
318 @Override
319 public String toString() {
320     String outputStr = "";
321     for (int index = 0; index < numberOfEntries; ++index) {
```

```
322         outputStr += arr[index] + "\n";
323     }
324
325     return outputStr;
326 }
327
328 /**
329  * Creates room for a new entry at the specified position.
330  *
331  * @param newPosition The position at which to create room.
332  */
333 private void makeRoom(int newPosition) {
334     int newIndex = newPosition - 1;
335     int lastIndex = numberOfEntries - 1;
336
337     // move each entry to next higher index, starting at end of
338     // array and continuing until the entry at newIndex is moved
339     for (int index = lastIndex; index >= newIndex; index--) {
340         arr[index + 1] = arr[index];
341     }
342 }
343
344 /**
345  * Removes the gap left by removing an entry at the given position.
346  *
347  * @param givenPosition The position at which to remove the gap.
348  */
349 private void removeGap(int givenPosition) {
350     // move each entry to next lower position starting at entry after the
351     // one removed and continuing until end of array
352     int removedIndex = givenPosition - 1;
353     int lastIndex = numberOfEntries - 1;
354
355     for (int index = removedIndex; index < lastIndex; index++) {
356         arr[index] = arr[index + 1];
357     }
358 }
359
360 public static <T extends Comparable<T>> void insertionSort(
361     ListInterface<T> a, Comparator<T> comparator, String val) {
362     for (int unsorted = 1; unsorted < a.size(); unsorted++) {
363         T firstUnsorted = a.getEntry(unsorted + 1);
364         insertInOrder(firstUnsorted, a, unsorted, comparator, val);
365     }
366 }
367
```



```

368 //inserts element at the correct index within the sorted subarray
369 private static <T extends Comparable<T>> int insertInOrder(T element,
370     ListInterface<T> a, int end, Comparator<T> comparator, String val) {
371     int index = end;
372     if ("asc".equals(val)) {
373         while ((index > 0) && (comparator.compare(element,
374             a.getEntry(index)) < 0)) {
375             a.replace(index + 1, a.getEntry(index));
376             //shifting
377             index--;
378         }
379     } else if (val.equals("des")) {
380         while ((index > 0) && (comparator.compare(
381             element, a.getEntry(index)) > 0)) {
382             a.replace(index + 1, a.getEntry(
383                 index)); //shifting
384             index--;
385         }
386     }
387     a.replace(index + 1, element);
388     return 0;
389 }
390
391 /**
392  * Checks if all provided elements are valid (non-null).
393  *
394  * @param newElements The elements to validate.
395  * @return {@code true} if all elements are valid, {@code false} if at least
396  * one element is null.
397  */
398 private boolean isElementsValid(T... newElements) {
399     boolean valid = true;
400     for (int i = 0; i < newElements.length && valid; i++) {
401         if (newElements[i] == null) {
402             valid = false;
403         }
404     }
405     return valid;
406 }
407
408 /**
409  * Returns an iterator over the elements in the list.
410  *
411  * @return An iterator over the elements in the list.
412  */
413 @Override

```

```

414     public Iterator<T> getIterator() {
415         return new getIterator();
416     }
417
418
419     @Override
420     public <T extends Comparable<T>> void bubbleSort() {
421         boolean sorted = false;
422         for (int pass = 1; pass < this.size() && !sorted; pass++) {
423             sorted = true;
424             for (int index = 1; index <= this.size() - pass; index++) {
425                 // swap adjacent elements if first is greater than second
426                 if (((T) this.getEntry(index)).compareTo((T) (this.getEntry(index + 1)))
427 > 0) {
428                     swap(index, index + 1); // swap adjacent elements
429                     sorted = false; // array not sorted because a swap was performed
430                 }
431             }
432         }
433
434         private void swap(int a, int b) {
435             T temp = this.getEntry(a);
436             this.replace(a, this.getEntry(b));
437             this.replace(b, temp);
438         }
439
440         public int compare(Tutor tutor1, Tutor tutor2) {
441             return tutor1.getSfaculty().compareTo(tutor2.getSfaculty());
442         }
443
444         /**
445          * Inner class to implement the Iterator interface for the ArrayList.
446          *
447          * @param <T>
448          */
449         public class getIterator<T> implements Iterator<T> {
450
451             private int index;
452
453             /**
454              * Constructs a new ListIterator.
455              */
456             public getIterator() {
457                 index = 0;
458             }

```

```
459
460     /**
461      * Checks if there are more elements to iterate over.
462      *
463      * @return {@code true} if there are more elements, {@code false}
464      * otherwise.
465      */
466     @Override
467     public boolean hasNext() {
468         return index < numberOfEntries;
469     }
470
471     /**
472      * Retrieves the next element in the iteration.
473      *
474      * @return The next element in the iteration.
475      */
476     @Override
477     public T next() {
478         if (!hasNext()) {
479             return null;
480         }
481         T nextEntry = (T) arr[index];
482         index++; // advance iterator
483         return nextEntry;
484     }
485 }
486 }
```