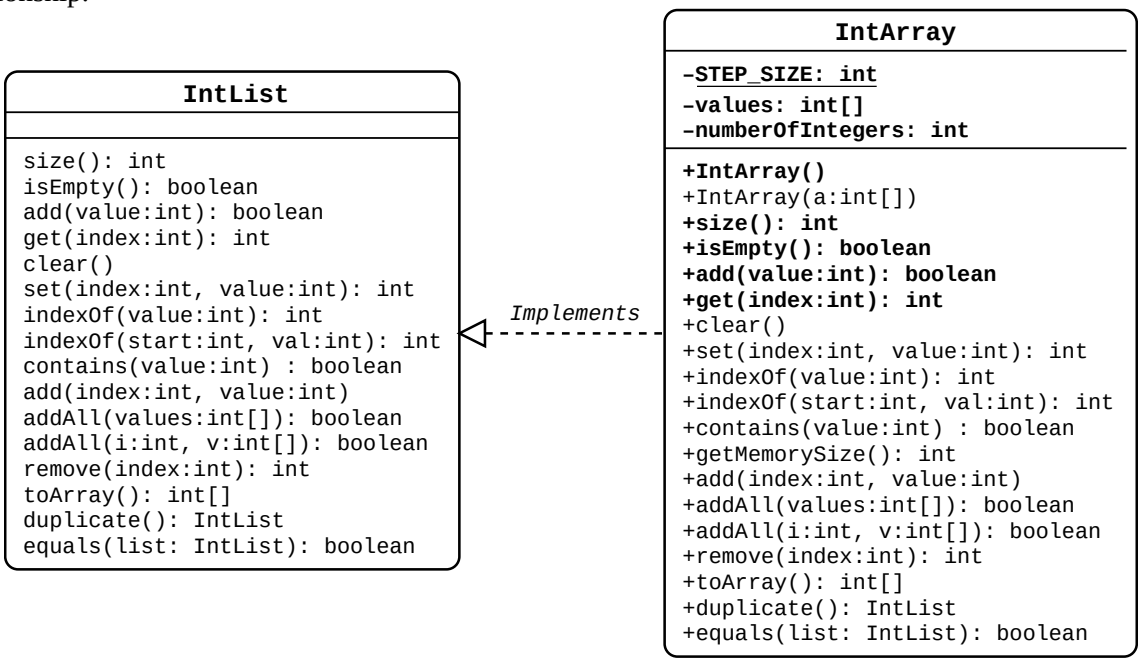
Unit 6 Assignment – IntArray

1. Understand the following UML class diagram representing an ***interface*** and a class that *implements* the given interface. An arrow drawn with a dotted line and an open triangle as the arrow head represents an *implements* relationship.



Create a new *interface* named `IntList` and in it declare all the methods shown in the UML class diagram, above. The first few lines of the interface are shown below. An interface is defined in a way similar to a class, however there are no methods implemented within an interface, only declared. Any class that claims to implement an interface must implement all the methods declared within the interface.

```
 1 /**
 2  * This is the interface for an expandable list of integers
 3  * @author christophernielsen
 4  *
 5  */
 6 public interface IntList {
 7
 8       // Step 1: Basic Functionality
 9       int size();
10       boolean isEmpty();
11       boolean add(int value);
12       int get(int index);
⋮              ⋮
```

Below are the first few lines of the `IntArray` class that you will write as an implementation of integer list objects that have been defined by the interface `IntList`. Note the Java keyword `implements`.

```
 1 /**
 2  * This class is an expandable integer list implemented using arrays
 3  *
 4  * @author christophernielsen
 5  *
 6  */
 7 public class IntArray implements IntList {
 8
 9     private static final int STEP_SIZE = 10;
⋮         ⋮
```

Unit 6 Assignment – IntArray

You will be given a file containing a Java class called `TestIntList`. This class will instantiate objects of your `IntArray` class and test the functionality of the class. You do not need to understand the code of this tester class; hopefully the output messages will be sufficient to help you understand errors your code may have. You may create your additional class to perform other tests on your code and to help debug any problems. Please do not modify the `TestIntList` code unless specifically told to do so, as you should use the original test code to demonstrate that your `IntArray` class is working correctly.

Before we start to implement methods, create "placeholder" methods (i.e.: methods that do not contain the proper functionality) for every method defined the interface `IntList` so that the code may compile. For example, since we have a method in interface `IntList` that has been declared as: "`boolean add(int value);`", we must write a method that conforms to this declaration in the `IntArray` class. Such a method (that does not yet contain the actual functionality, but is just a placeholder) may look like this:

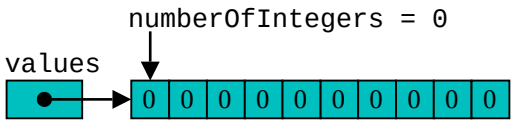
```
1 public boolean add(int value) {
2     return false;
3 }
```

For a method that is to return an array, such as the `toArray` method, or an object, such as the `duplicate` method, you may temporarily use the return value `null`.

Once you have the `TestIntList` code running with your `IntList` interface and `IntArray` class, demonstrate this to your teacher.

2. For this part of the assignment, you are to implement the most basic functionality of the `IntArray` class. The fields and methods required have been highlighted in the UML diagram in part 1.

The field named `values` will be used to store the list of integers. In the constructor, we will need to create a new array of integers and assign it to the field name `values`. Define the constant `STEP_SIZE` with a value of `10`, and use this as the number of integers in the initial integer array. The field `numberOfIntegers` will keep track of the number of integers that have been added to our list, and should also be initialized to zero. The field values after the constructor is called will are represented diagrammatically here.

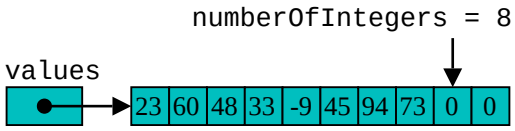


We will later need to write code to expand this array if it fills up, but for this part we will not try to add more than ten integers to our list.

The **`size`** method is to return the count of the number of integers stored in the list.

The **`isEmpty`** method is to return `true` if there are no integers stored in the list.

The **`add`** method is to add the given value to the end of the list and increment the counter for the number of integers stored in the list. The field values after eight integers have been added to the list are represented diagrammatically here.



The **`get`** method is to return the value of the element at the index given by the parameter `index`. For example, if eight elements were added to the list, and `get` is called with the `index` parameter set to `2`, then the method is to return the third element that was added to the list (the `index` is to be zero-based).

Once your code is complete and passes `test1a` from `TestIntList`, demonstrate this the teacher.

3. At present, our `get` method does not verify that the value retrieved from the list was actually stored in the list. If the user tries to request an integer from the list with an index of `10` or greater, then the program will crash because we have only allocated memory for ten integers, with indices `0` through `9`. However, if four elements were added to the list, for example, and the method `get` were called with the `index` parameter set to `4`, the method will return the value zero even though we did not store a zero in this position. Instead of having this silent error, the method is to throw an `IndexOutOfBoundsException` when there is an attempt to access to a position where no element yet saved. To raise an exception, we write code to explicitly do so. Writing of exceptions is not covered in the AP Java Subset, so the code to raise the exception is given below. You will

only need to write the code for the private method `invalidIndex`, which is to return true if the index is out of bounds, and call the given method `checkIndex` from within the `get` method in order to throw an exception if the index is out of bounds.

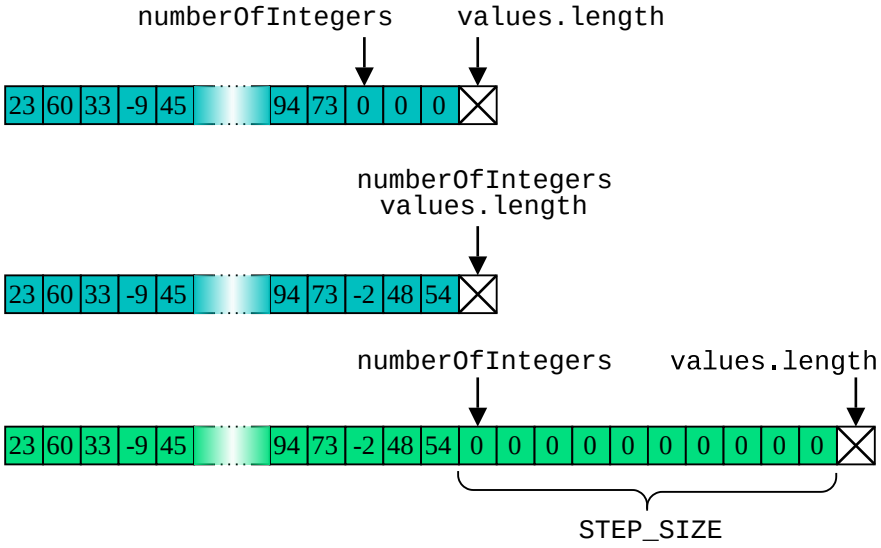
```
 1 private boolean invalidIndex(int index) {
 2     // TODO: replace with correct index check
 3     return true;
 4 }
 5
 6 private boolean checkIndex(int index) {
 7     if(invalidIndex(index)) {
 8         throw new IndexOutOfBoundsException(
 9                 "List index " + index +
10                 " out of bounds for length " +
11                 numberOfIntegers);
12     }
13     return true;
14 }
15
16 public int get(int index) {
⋮         ⋮
```

Once your code passes `test1b`, demonstrate it to your teacher.

4. Write the code for the **`clear`** method.
   This method should put the instance of the class of an empty list. *Hint*: you will not need to allocate a new array, nor reset array values to zero; as long as the list appears to be an empty list when you access the methods in the class, it is effectively an empty list.
   Once your code passes `test1c`, demonstrate it to your teacher.
5. Write the code for the **`set`** method.
   This method is to overwrite the value at the index given by the parameter `index` by the value given in the parameter `value`. Note that this method should not allow you to set an index that does not already have a value in it, so you will need to check if the index is out of bounds, as you have for the `get` method.
   The return value is to be the value that was previously in the array before it was overwritten.
   Once your code passes `test1d`, demonstrate it to your teacher.
6. Write the code for the two versions of method **`indexOf`**, and for the method **`contains`**.
   One version of `indexOf` takes only a single parameter, `value`, that contains the value to search for. This version is to start searching from the start of the list, and continue until it finds the value, or it reaches the end of the list without finding the value. If the integer given in `value` is found, it returns the index where it was first found. If it is not found, `-1` is returned.
   The Second method operates the same way, but rather than starting at the start of the list, it starts from the index given by the parameter `start`.
   The `contains` method returns `true` if the element given by parameter `value` is found in the list.
   Consider how you can save time and energy, and make the code cleaner by not duplicating code. If you are copying and pasting code, it is a bad sign. It suggests that you have duplicate code in your program and there is likely a better way.
   Once your code passes `test1e` and `test1f`, demonstrate it to your teacher.
7. Now we are getting to the more interesting and useful part of this project. We will now implement code that will extend the array beyond the default initial size (arbitrarily decided to be `10`). To do this, we will need to modify the code for the `add` method to include a check to see if adding another integer to the list will go beyond the capacity of the current array, and if it will, we must first extend the size of the array before we attempt to add the integer value to the list. The method `getMemorySize` will need to be implemented for testing purposes (there isn't really a good reason to keep this in the final code). This method is to return the number of elements that can be stored in the currently allocated array, `values`.
   In fact, there is no way in Java to simply extend an array. The memory space immediately after the array may very well be in use to store some other data. Therefore, in order to "extend" the array, we will need to create a new, larger array (yes, using the keyword `new`, similar to how the constructor created the initial array), and then copy the contents of the current array into the newly allocated array. The new array should be the size of

the current array, plus an additional number of elements equal to the constant `STEP_SIZE`. For the test in `TestIntArray` to work properly, this constant must be set to `10`.

The diagram below is intended to help you visualize what happens to the value of `numberOfIntegers` and the length of the `values` array when we use the `add` method to add values to the list, and what should happen when the array of values becomes full. The code will need to copy the values from the original array into the new array. (Note: this diagram does not show adding the new value added to the array after the extension has been made, but only the extending of the array.)



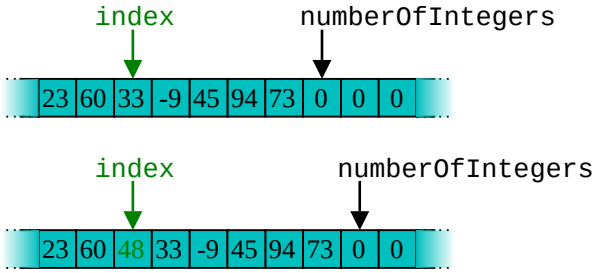
The `add` method should not only extend the array only once, but any time the method is called and the `values` array is found to be full. The code in `TestIntList` will require the array to be extended multiple times, and will verify that the memory size is what it is expected to be by calling the method `getMemorySize`.

Once your code passes `test2a`, demonstrate it to your teacher.

8. Write the code for the version of the method **`add`** that takes two parameters: an `index` where to insert the number, and the `value` to insert.

   Unlike the `set` method, this method is to <u>insert</u> the value before the value that is at index prior to the call, rather than overwrite it. Thus, all the values stored in the array after the index will need to be moved one position down in the `values` array. This method should append to the list (just like the original `add` method with no index does) if the given index is one position beyond the last value in the list. This method requires checking if the index is out of bounds, as well as checking whether the addition of the value will require the array to be extended.

   The diagram below is intended to help visualize the operation of the `add` method when the array `values` does not need to be extended. In this example, the value of `48` is inserted at the index given by `index`.



   Once your code passes `test2b`, demonstrate it to your teacher.

9. Implement the two versions of method **`addAll`**. Each of these methods takes an integer array as its parameter. One version will append the elements, and the other version will insert the elements at the position given by `index`.

If this were a proper library method, we would probably want to optimize this method. We wouldn't call the `add` multiple times to insert each element. Recall that each call to the `add` method to insert an element, we will execute a loop to copy all the elements one space down so the element may be inserted. It would be much more efficient to copy all the elements down one time by the number of spaces that is required to insert all the elements given in the parameter `values`, then copy all the elements into the space created.

Another way calling the `add` method is inefficient (this time it is inefficient, even if appending) is if we end up extending the array multiple times. It would be better to extend the array enough to insert all the elements, rather than extending multiple times, which requires copying the old array into the newly allocated array multiples times.

That being said… at times, (such as this case) we may wish to minimize the amount of time we will spend coding and testing, rather than try to minimize execution time. In this case, we're not dealing with huge lists or a lot of processing, so the difference in the amount of time it will take our code to execute will not even be noticeable.

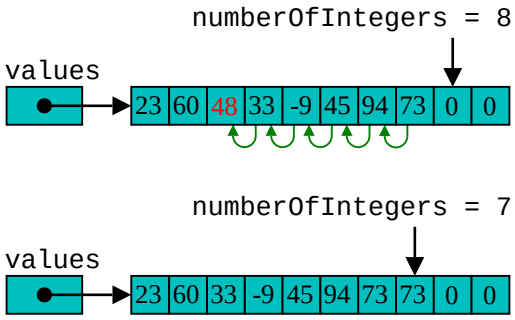
So, unless you wish to challenge yourself, just call one of the `add` methods in each of your `addAll` methods to save yourself some time. Your methods should return `true` if the list has changed as a result of the call.

Once your code passes `test2c1` and `test2c2`, demonstrate it to your teacher.

10. Implement the method **`remove`**.

    This method will need to check whether the index is valid (raise an out of bounds exception if it isn't), copy all the elements after the removed element one space earlier in the array, and appropriately modify `numberOfIntegers`.

    Below is a diagram representing an example where the element at index `2` is removed from a list of eight elements. Note that you do not need to reset the element at index `7` to zero because this array element is not visible outside of the instance – any attempt to read the element at this index should cause an index out of bounds exception.



    Once your code passes `test2d`, demonstrate it to your teacher.

11. Implement the method **`toArray`**.

    This method is to return an array filled with the integers stored in the list. *Hint*: You will need to allocate an array that will exactly fit the number of integers that is in the list.

    Once your code passes `test2e`, demonstrate it to your teacher.

12. Implement the method **`duplicate`**, and the constructor for `IntArray` that takes an array of integers as a parameter.

    Because the original object that we are going to duplicate is of type `IntArray`, the `duplicate` method is to return a new object that is also of type `IntArray`. Although the return type of the `duplicate` method is shown to be `IntList`, because an `IntArray` is a sub-type of the type `IntList`, you may return an `IntArray` object from this method without any type casting or other conversion.

    We will need to make a new object of type `IntArray` to return from the `duplicate` method. This means calling a constructor. Presently, we've only created a constructor that takes no parameters. Perhaps a constructor that will take an array of integers to start the list with would be useful for users of the `IntArray` class. Then we can use that constructor in our `duplicate` method.

    Start by implementing the `IntArray` constructor that takes an array of integers as a parameter. You may wish to *refactor* (meaning modify or re-write) the code for the original `IntArray` method so that you will not have duplicate code in your class. The version of the constructor that takes an array of integers as a parameter should copy all the values from that array into the newly created list. Consider what size of array you will need

to start with. In contrast to the `addAll` method, this constructor method should be easy to write without any calls to other methods in the class.

Once you have completed the new version of the `IntArray` constructor, use it to implement the `duplicate` method. With the constructor implemented, the `duplicate` method can be written in just one or two lines of code.

Once your code passes `test2f1` and `test2f2`, demonstrate it to your teacher.

13. Implement the method **`equals`**.

We have used the `equals` method to test the equality of strings, and you were taught that the "proper" way to test if two objects are equivalent is to call the `equals` method. Implementing your own `equals` method for a class will hopefully help you better understand why we may often wish to implement and use `equals` methods for our own classes.

This method is to return true if all the elements of the list are equal to all the elements of the list provided as a parameter (i.e.: all the same values, in the same order). Note that when called as: `list1.equals(list2)`, the `list1` array elements are in the array `values`, while the values for the elements in `list2` may be accessed by using `list.get(index)` (because the parameter name is `list`, and it is of type `IntList`, which has a `get` method).

Once your code passes `test2g`, demonstrate it to your teacher.