Lab 5 Array based Bag and List Implementations

Goal

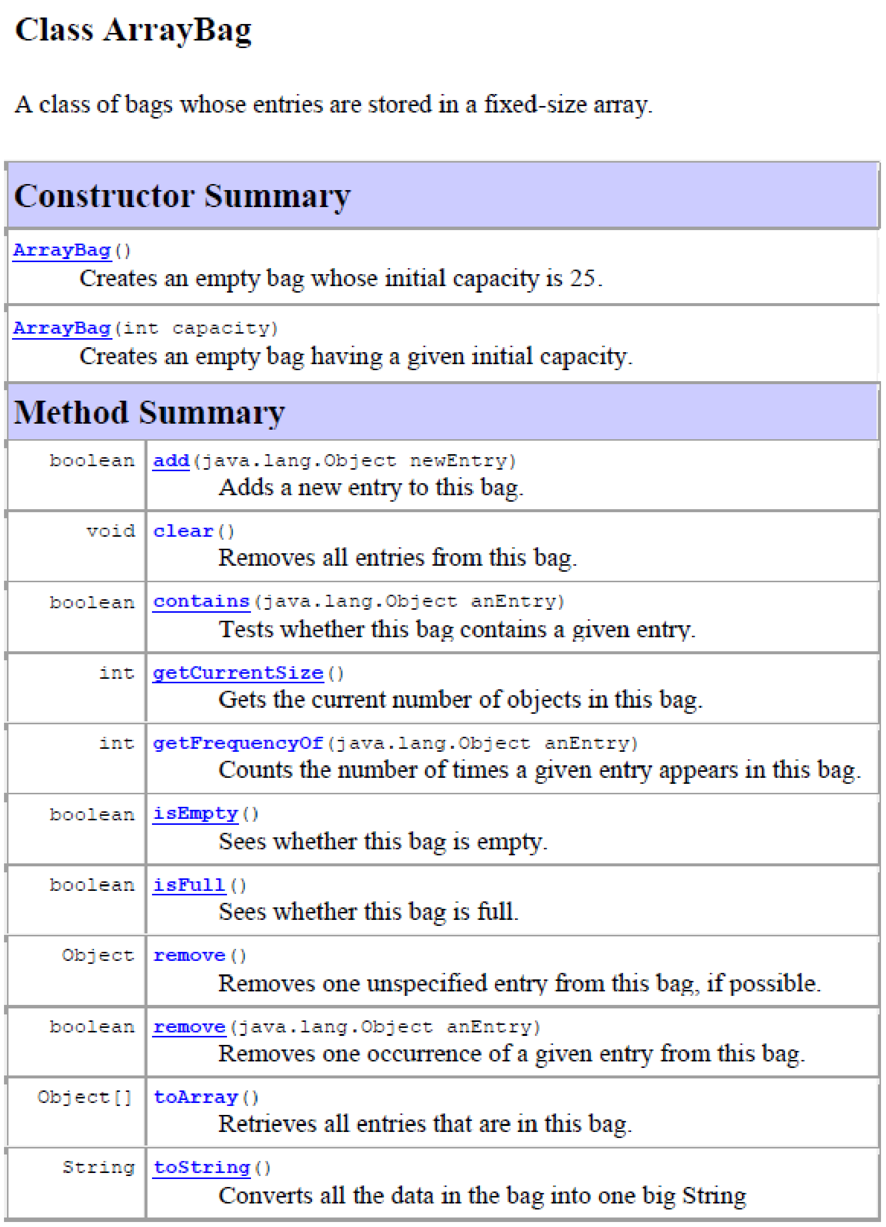
Having worked with Bags and Lists in previous labs, we are now going to pull the covers off them and learn how they are constructed and how they work internally. We will do this by creating our own versions of these classes, using internal arrays to store the data. Next week we will repeat this exercise using linked Nodes to store the data.

Resources

- Chapter 2, and 13

Part A ArrayBag class

The Bag class was introduced in lab 2. From an abstract point of view, a Bag is just a container we can store objects in. The following Javadoc defines many of the methods we tried out in lab. Today we will build a class according to these specifications.



Class ArrayBag

A class of bags whose entries are stored in a fixed-size array.

Constructor Summary
`ArrayBag()` Creates an empty bag whose initial capacity is 25.
`ArrayBag(int capacity)` Creates an empty bag having a given initial capacity.

Method Summary	
`boolean`	`add(java.lang.Object newEntry)` Adds a new entry to this bag.
`void`	`clear()` Removes all entries from this bag.
`boolean`	`contains(java.lang.Object anEntry)` Tests whether this bag contains a given entry.
`int`	`getCurrentSize()` Gets the current number of objects in this bag.
`int`	`getFrequencyOf(java.lang.Object anEntry)` Counts the number of times a given entry appears in this bag.
`boolean`	`isEmpty()` Sees whether this bag is empty.
`boolean`	`isFull()` Sees whether this bag is full.
`Object`	`remove()` Removes one unspecified entry from this bag, if possible.
`boolean`	`remove(java.lang.Object anEntry)` Removes one occurrence of a given entry from this bag.
`Object[]`	`toArray()` Retrieves all entries that are in this bag.
`String`	`toString()` Converts all the data in the bag into one big String

Pre-Lab Visualization

Array-based Bag

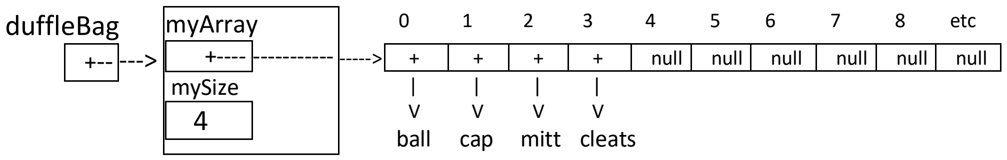
The array based bag will use an internal array to maintain the items in the bag. As we saw in lecture, the items in the bag will be added to the internal array at the location indicated by the the **mySize** instance variable.

Here is some code that creates one of our simple array based bags:

```
ArrayBag duffleBag = new ArrayBag();
duffleBag.add("ball");
duffleBag.add("cap");
duffleBag.add("mitt");
duffleBag.add("cleats");
```

Notice we are not using type specifiers in angle brackets as we did in our lab experiments. This ArrayBag class is somewhat simpler than the one we used before.

A (crude) memory map of the above ArrayBag object might look something like this



In other words, **duffleBag** is a reference to the **ArrayBag** object, which has an internal **array** reference and a **mySize** field. Note that the **mySize** field indicates that the next available cell in the array is at index 4.

A simpler memory notation look be something like **duffleBag → [mySize: 4, myArray:]→**ball,cap,mitt,cleats

Using the above as a guide, draw pictures of the two ArrayBags that are formed by the following statements (note that the remove method removes the last item added to the bag):

```
ArrayBag bag1 = new ArrayBag();
ArrayBag bag2 = new ArrayBag();
bag1.add("a"); bag1.add("b"); bag1.add ("c"); bag1.add ("d"); bag1.remove();
bag2.add(bag1.remove()); bag2.add("e");
```

bag1	bag2

Directed Lab Work

We will now start developing and testing the various methods for the ArrayBag class.The in class demonstration will show how the following methods are implemented:

constructors(2), isFull, isEmpty, getCurrentSize, toString

Your task will now be to complete the following methods:

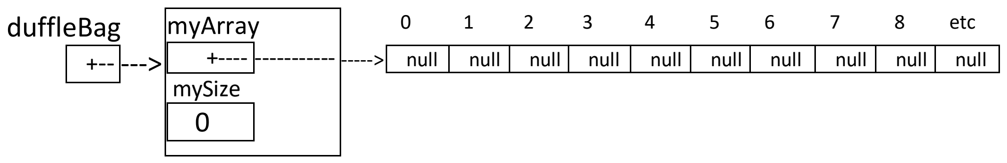
add, getFrequencyOf, contains, remove, clear

And, if time permits, 2 optional methods:

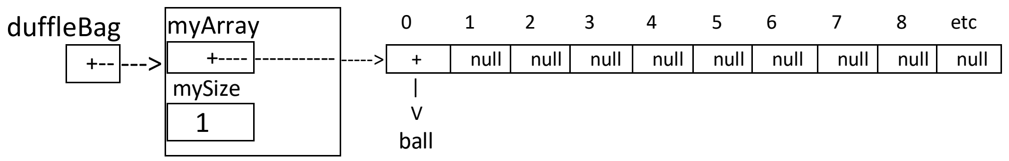
remove(anEntry), toArray

Problem 1: the add method

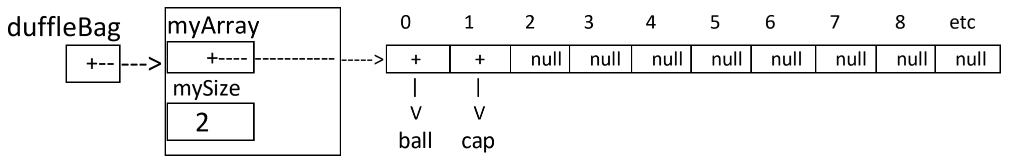
When the ArrayBag is initially created, **myArray** is filled with null values and **mySize** is 0.



After we execute the statement `duffleBag.add("ball");` the duffleBag object should undergo the following changes:



And after we execute the statment `duffleBag.add("cap");` the duffleBag object should undergo the following changes:



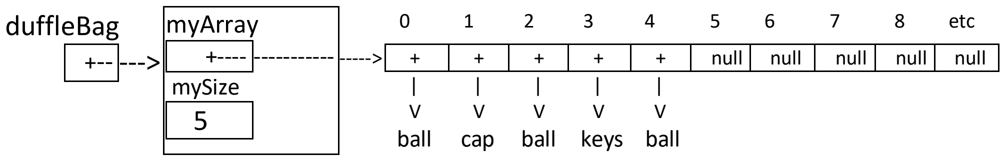
With these observations in mind, we can formulate a basic algorithm for the add method:

1) store **newEntry** at location **mySize** in **myArray**
2) add 1 to **mySize**

But what if the bag is full? **We should add an if statement to our algorithm to only do the above steps if our bag is NOT FULL.** Also, the add method **is supposed to return true if it was successful and false otherwise**. Go ahead and write your modified definition for add now. Then test your method in the Problem1 section of the lab5a test program.

Problem 2: the getFrequencyOf method

When the ArrayBag has a few items in it, we may wish to know how many times an item occurs in the bag. For example, given the following state of our bag,



The statement `duffleBag.getFrequencyOf("ball");` will return the value 3.

In order to determine the number of times **anEntry** occurs in the bag, we'll need to use a loop with an external counter. Here is an approach to solving this problem:

1) declare and set **numTimes = 0**
2) loop over all the items in the array up **until** the loop variable **k** reaches **mySize (a for loop)**
    a. If the object referred to by **myArray** index k "equals" the **anEntry** object
        i. add one to **numTimes**
3) after the loop, return the value in **numTimes**

Write this method now and test it in the lab5a test program.

Problem 3: the contains method

This method returns true if **anEntry** is stored in the bag. It can be written in one of two ways.

Option 1 is to run through the items in myArray using a for loop. If there is ever an object in the array that "equals" anEntry, return true.

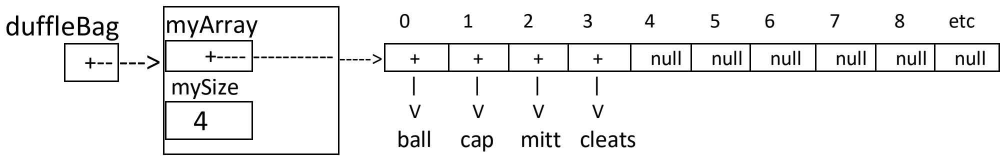
Although option 1 is fine, there is a better way. The advantage of writing numerous methods for a class is that some methods can be written in terms of others. Here is the alternative:

Option 2 is to just use **getFrequencyOf** You could simply return the logical expression that **getFrequencyOf** is greater than 0.

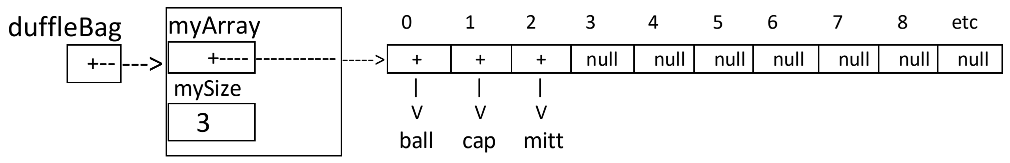
Write this method now and test it in the lab5a test program.

Problem 4: the remove method

Suppose our bag at some point looks like the following:



As we learned in lecture, the remove method is easiest to write if it just removes the last item added to the bag. So, the statement `duffleBag.remove();` would make the bag now look like:



Notice the changes that have taken place on the dufflebag with the single remove statement. Remember, your method has to **return** the entry being removed. Come up with an algorithm to solve this method and test it in problem 4 of the lab5a test program.

Problem 5: the clear method

The clear method removes all of the items in the array. It could be written one of two ways: 1) using a for loop to set all of the cells in myArray to null (and then set mySize to 0), or 2) use a while loop to continually **remove** one item from **myArray** as long as the bag is NOT **isEmpty**.

Code up your solution to this method and test it.

Problem 6: method remove(anEntry)

There is another method that removes items from the bag. This one requires the user to specify which item to remove. To solve this one, first locate the index in **myArray** where **anEntry** is stored (if at all). Then, copy the last item in **myArray** to the location of **anEntry**, reduce mySize by one, and return **true**. If the item is not found, return **false**.

In order to locate **anEntry**, our text makes use of a private **getIndexOf** method, which returns the index where **anEntry** is located, or -1. This simplifies the logic of **remove(anEntry)**, but it requires writing an extra method. This method is private because we don’t want it being invoked by users of our ArrayBag class. Knowing what index an item is stored at would violate the understanding of what a bag represents at the abstract level: an unordered collection of objects.

Problem 7: Converting the ArrayBag to a generic class

We have been using **Object** as the data type for all of the objects added to the bag, and for the base type of **myArray**. Although this approach works as a means to create a working **ArrayBag** class, it does have some defects. The problem occurs when we accidently insert something unintended into our Bag, such as an array of String rather than a single String object. These mistakes would go unnoticed by the compiler and not appear until run time, where they are usually much harder to find. In order to allow the compiler to flag incorrect insertions into our **ArrayBag**, we will now make a simple modification that allows us to specify the data type our ArrayBag can hold.

The steps to do this are: 1) add **<T>** after the class declaration, and 2) Change all occurences of **Object** to **T** in the remaining code,

The constructor requires a little extra work: Java does not allow declaring arrays of a generic type, so constructor will create an array of Object and cast it as an array of T. This requires suppressing the compiler warnings about an unsafe cast, but there is no danger here, it’s just a protocol that must be followed. Here is how the constructor will look:

```
public ArrayBag(int initialCapacity)
{
       mySize = 0;
       @SuppressWarnings("unchecked")
       T[] tempArray = (T[]) new Object[initialCapacity];
       myArray = tempArray;
} // end constructor
```

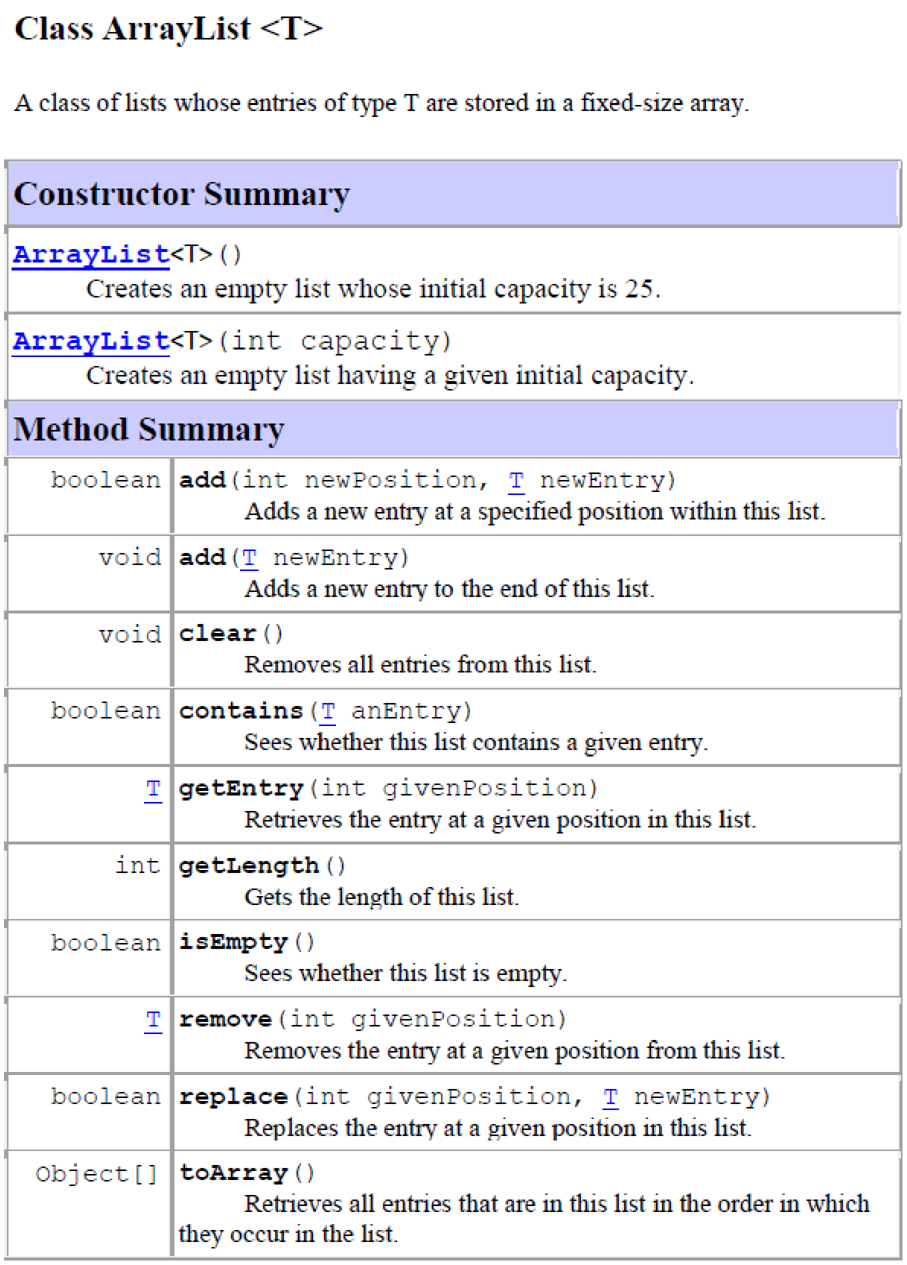
Try this out and then check that you can now declare duffleBag as an ArrayBag<String>().

Problem 8: the toArray method

The toArray method is defined incorrectly in our text—it does not need to be cast into an array of T since Java does not remember what T is for arrays—although the basic idea is correct. The steps are as follows: 1) make a new array of Object called result that has space for mySize items; 2) write a for loop that copies all the items from myArray over to the result array; 3) return the result array. Write your toArray method and then test it in the lab5a program.

Part B ArrayList class

The List class was introduced in lab 4. A List is a container that stores objects in a particular order, and allows deleting and adding items at any point in the list. The following Javadoc defines many of the methods we tried out in lab. Today we will build a class according to these specifications.



Class ArrayList <T>

A class of lists whose entries of type T are stored in a fixed-size array.

Constructor Summary
`ArrayList<T>()` Creates an empty list whose initial capacity is 25.
`ArrayList<T>(int capacity)` Creates an empty list having a given initial capacity.

Method Summary	
`boolean`	**`add`**`(int newPosition, T newEntry)` Adds a new entry at a specified position within this list.
`void`	**`add`**`(T newEntry)` Adds a new entry to the end of this list.
`void`	**`clear`**`()` Removes all entries from this list.
`boolean`	**`contains`**`(T anEntry)` Sees whether this list contains a given entry.
`T`	**`getEntry`**`(int givenPosition)` Retrieves the entry at a given position in this list.
`int`	**`getLength`**`()` Gets the length of this list.
`boolean`	**`isEmpty`**`()` Sees whether this list is empty.
`T`	**`remove`**`(int givenPosition)` Removes the entry at a given position from this list.
`boolean`	**`replace`**`(int givenPosition, T newEntry)` Replaces the entry at a given position in this list.
`Object[]`	**`toArray`**`()` Retrieves all entries that are in this list in the order in which they occur in the list.

Pre-Lab Visualization

1) To make sure you understand how a List works, show the contents of the sample List at each step while performing the following operations on **ArrayList<String> myList**. Just list the contents from left to right:

Operation	Resulting List items (item 1 on left)
myList.add("A");	
myList.add("B");	
myList.add("C");	
myList.add("D");	
myList.add(1, "one");	
myList.add(1, "two");	
myList.add(1, "three");	
myList.add(1, "four");	

2) Show the contents of the List at each step while performing the following operations on List myList. Just list the contents from left to right:

Operation	Resulting List items (item 1 on left)
myList.add("alpha");	
myList.add(1, "beta");	
myList.add("gamma");	
myList.add(2, "delta");	
myList.add(4, "alpha");	
myList.remove(2);	
myList.remove(2);	
myList.replace(3, "delta");	

The array based list will use an internal array to maintain the items in the list. As we saw in lecture, the items in the list can be added at the end or at other locations in the list. In this lab we will be using the generic type specifier **<T>** in our code to help make our data structures type safe. Note that our list will be able to expand as needed to accommodate lists that grow beyond their initial capacity. Therefore a list can never become full, so the **isFull** method is not included in our class declaration. Also, the **add** method will always be successful, so it no longer needs to return a boolean (it's now a void method).

Directed Lab Work

We will now start developing and testing the various methods for the ArrayList class. Since the ArrayList shares many of the same features as the ArrayBag class, some of the method definitions are the same for both. In these cases (the **constructors(2), isEmpty, getLength, contains, clear, and toString**) definitions have already been provided.

Your task will now be to complete and test the following methods:

getEntry(position), add(position, newEntry), remove(position), replace(position, newEntry)

And, if time permits, 2 optional methods: **getPosition(anEntry), moveToEnd(position)**

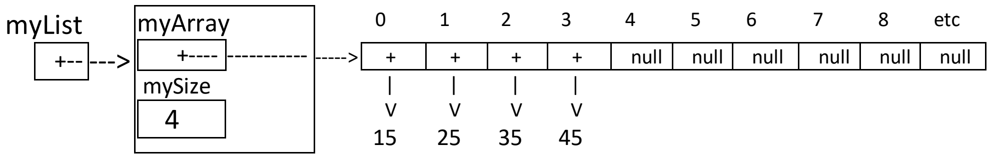
Problem 1: the getEntry(position) method

The **getEntry** method returns the item in **myArray** that sits at **position**. However, there is a twist: our ArrayList class defines position 1 as the start of the list. But our array storage begins at index 0. This can get confusing! However, there is an easy way to translate from the position in the list to the index in the array. Simply define the variable **index** inside any methods that use **position**, as

```
        int index = position-1;
```

and then **index** will refer to the desired array cell number, starting from 0, that we are already familiar with.

As an example, for the following situation with **myList**, if we print the expression **myList.getEntry(3)** we should see a "35" on the screen, NOT "45"!!



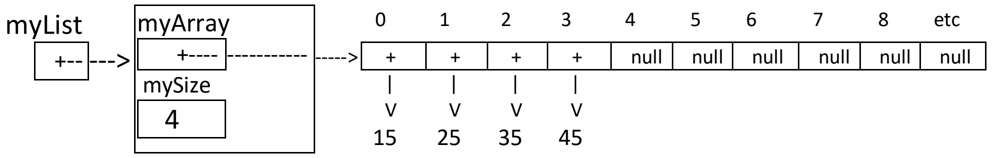
Another task that **getEntry** must perform is to ensure that only legal index values are used to access the array (a user might accidently execute **myList.getEntry(-4)** or **myList.getEntry(5)**, both of which are outside the range of allowable values for **index**. Should this happen **getEntry** should return a value of null.

Write this method now and verify that your program prints the correct output. It's a simple method that just returns the desired array cell from myArray, or null if the desired cell is not valid.

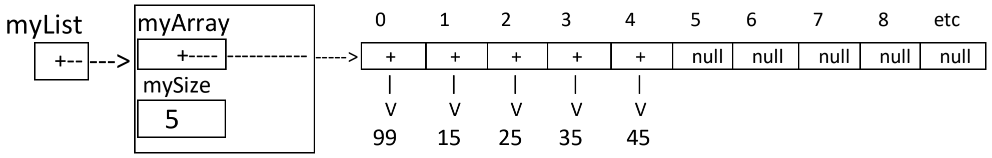
Problem 2: the add(position, newEntry) method

There is an add method for ArrayList which is identical to the add method for ArrayBag. It adds newEntry at the end of filled cells in **myArray**. With a list, however, we often want to add items to different locations besides the end. That is what the **add** method does that we are looking at now. This method takes a **position** and places **newEntry** at the desired location (remember, list positions start at 1 and map to an array index of 0). This method also returns **true** if the add was successful (the position was a valid location) and **false** if the add could not be performed (the position requested was outside the allowable range of cells in the array—it would've created a gap in the array or gone past the array bounds).

Example 1: if we have the following list defined:

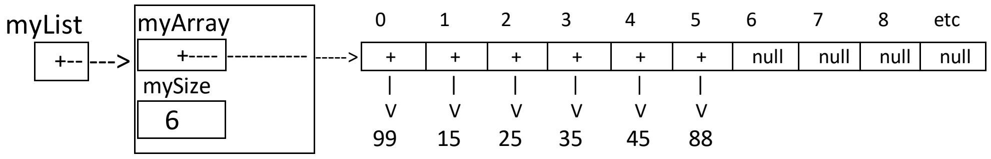


and we execute **myList.add(1,"99");** the result on **myList** will be to change it as follows:



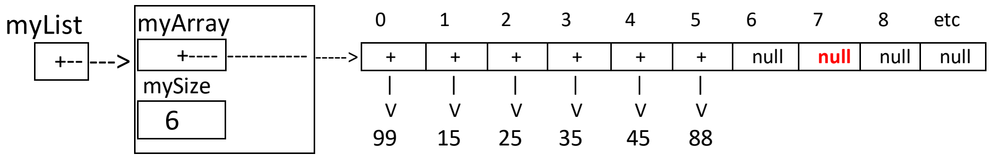
and the **add** method would return **true**, meaning the add was successful.
Notice that the contents of cells 0 to 3 had to shift upward (rightward) one cell to make room for the new entry.

Example 2: If we now execute the statement: **myList.add(6,"88");** the result on **myList** will be to change it as follows:



And again, the **add** method would return **true** (success).

Example 3: However, if we now execute the statement: **myList.add(8,"77");** **myList** will remain **<u>unchanged</u>** and a result of false will be returned since this command attempts to write at cell 7 in the array, leaving a gap at cell 6.



Our algorithm for this add method will then look something like the following:

Add **newEntry** at **position**:

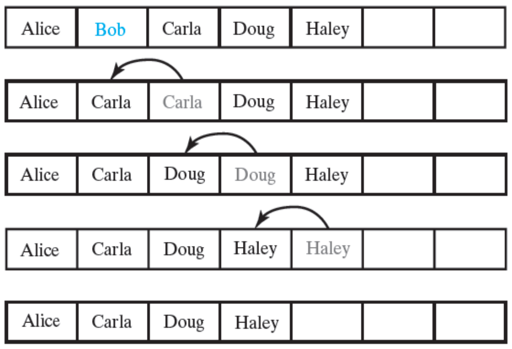
1) if **position** is within the valid range of locations (for the above list that would be positions 1 thru 7)
    a. call **ensureCapacity** to increase the size of the array if needed
    b. define **index** as **position -1** (the array cell number we will place **newEntry** at)
    c. starting with the cell at **mySize**, copy all the values in **myArray** one cell to the right, stopping when cell **index** is reached. (for loop)
    d. Assign **myArray[index] = newEntry** (stores newEntry in correct location)
    e. return **true**;
2) otherwise, return **false**;

Try this now and test out your code in problem 2 of Lab5b

Problem 3) remove, replace

Remove:

Remove saves a reference to the item at **givenPosition** before copying all elements above **givenPosition** to the left one cell, and then returns the reference to the removed item. This method will need to write a loop in remove that copies the appropriate items left one cell as shown in the following diagram showing Bob being removed from a list containing Alice,Bob,Carla,Doug and Haley:



Replace:

If the **givenPosition** is legal, this method copies over the item in `myArray` at **givenPosition** with a reference to **newEntry**, and returns **true**. Otherwise it returns **false**.

Problem 4) getPosition, moveToEnd

Note: there are no stubs for these methods so you'll have to add them competely, with javadocs, to the ArrayList file.

Read the example test code in the Lab5b file to see how these methods are used. The method getPosition should return the position (remember, we start counting at 1) of an item in the list. If it is not found it should return the flag -1.

the method moveToEnd should move the item at givenPosition to the end of the list.

Problem 5) Replace without replace at the client level

The list operations we've been creating have a lot of redundancy. To illustrate this, write some code at the client level under Problem 5 in Lab5b that modifies myList, replacing the "53" item with a "35", only do this **without** using the **replace** method. Use **remove** and **add** to do the replacement. Also, imagine that you will not be able to visualize the list, so you'll have to use **getPosition** to tell you where the item is located, in other words,

1) Find the position of the item to change
2) remove that position
3) add the new value at the same position