Arrays

As you know from COMP 202 (or equivalent), an array is a data structure that holds a set of elements that are of the same type. Each element in the array can be accessed or indexed by a unique number which is its position in the array. An array has a capacity or `length`, which is the maximum number of elements can be stored in the array. In Java, we can have arrays of primitive types or reference type. For example,

```
int[]   myInts = new int[15];
```

creates an array of type `int` which has 15 slots. Each slot is initialized to value 0. We can also define an array of objects of some class, say `Shape`.

```
Shape[]  shapes = new Shape[428];
```

This array can reference up to 428 `Shape` objects. At the time we construct the array, there will be no `Shape` objects, and each slot of the array will hold a reference value of `null` which simply means there is nothing at that slot.

We are free to assign an element to any slot of an arrays. We can write

```
myInts[12] = -3;
shape[239] =  new Shape("triangle", "blue");
```

So if each of these arrays was empty before the above instructions (i.e. after construction of the array), then after the instruction each array would have one element in it.

Arrays have constant time access

A central property of arrays is that the time it takes to access an element does not depend on the number of slots (`length`) in the array. This constant access time property holds, whether we are writing to a slot in an array,

```
a[i] = ....;
```

or reading from a slot in an array

```
...  =   a[i];
```

You will understand this property better once you have taken COMP 206 and especially COMP 273, but the basic idea can be explained now. The array is located somewhere in the computer memory and that location can specified by a number called an *address*. When I say that the array has some address, I mean that the first slot of the array has some address. Think of this as like an apartment number in a building, or an number address of a house on a street. Each array slot then sits at some location relative to that starting address and this slot location can be easily computed.

[ASIDE: To find out where `a[k]` is, you just add the address of `a[0]` to `k` times some contant, which is the amount of memory used by each slot. This works since each slot in the array has constant size. The may be different for arrays of `int` versus arrays of `double` versus arrays of `Shape` but that's not a problem for the constant access time property.]

Lists

In the next few lectures, we look at data structures for representing and manipulating *lists*. We are all familiar with the concept of a list. You have a TODO list, a grocery list, etc. A list is different from a "set". The term "list" implies that there is a positional ordering. It is meaningful to talk about the first element, the second element, the i-th element of the list, for example. For a set, we don't necessarily have an order.

What operations are performed on lists? Examples are getting or setting the i-th element of the list, or adding or removing elements from the list.

```
get(i)          //  Returns the i-th element (but doesn't remove it)
set(i,e)        //  Replaces the i-th element with e
add(i,e)        //  Inserts element e into the i-th position
add(e)          //  Inserts element e (e.g. at the end of the list)
remove(i)       //  Removes the i-th element from list
remove(e)       //  Removes element e from the list (if it is there)
clear()         //  Empties the list.
isEmpty()       //  Returns true if empty, false if not empty.
size()          //  Returns number of elements in the list
:
```

In the next several lectures, we will look at some ways of implementing lists. The first way is to use an array. The data structure we will describe is called an *array list*.

Using an array to represent a list

Suppose we have an array `a[ ]`. For now I won't specify the type because it is not the main point here. We want to use this array to represent a list. Suppose the list has `size` elements. We will keep the elements at positions 0 to `size-1` in the array, so element i in the list is at position i in the array. This is hugely important so I'll say it again. With an array list, the elements are squeezed into the lowest indices possible so that there are no holes. This property requires extra work when we add an remove elements, but the benefit is that we always know where the ith element of the list is, namely it is at the ith slot in the array.

Let's sketch out algorithms for the operations above. Here we will not worry about syntax so much, and instead just write it the algorithm as pseudocode.

We first look at how to access an element in an array list by a read (get) or write (set).

`get(i)`

To get the i-th element in a list, we can do the following:

```
if (i >= 0) &  (i < size)
   return a[i]
```

Note that we need to test that the index `i` makes sense in terms of the list definition. If the condition fails and we didn't do the test, we would get in index out of bounds exception.

We set the value at position i in the list to a new value as follows:

set(i,e)

```
if (i >= 0) &  (i < size)
   a[i] = e
```

[ASIDE: This code replaces the existing value at that position in the list. If there were a previous value in that slot, it would be lost. An alternative is to return this previous element, for example,

```
tmp = a[i]
if (i >= 0) & (i < size)
    a[i] = e
return tmp
```

In fact, the Java ArrayList method `set` does return the element that was previously at that position in the list. See
`https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html#set(int,%20E)`]

Next we "add" an element `e` to i-th position in the list. However, rather than replacing the element at that position which we did with a `set` operation, the `add` method *inserts* the element. To make room for the element, it displaces (shifts) the elements that are currently at index $i, i+1, ..., \texttt{size}-1$. Here we can assume that $i \leq \texttt{size}$. If we want to add to the end of the list then we would add at position `size`. Moreover, we also assume for the moment that the size of the list is strictly less than the number of slots of the underlying array, which is typically called the length (`a.length`) of the array. It is also called the *capacity* of the array. The capacity is greater than the size, except in the special case that the array is full.

add(i,e)

```
if ((i > 0) & (i <= size) & (size < length)){
   for (j = size; j > i; j--)
       a[j] = a[j-1]              //  shift to bigger index
   a[i] = e                       //  insert into now empty slot
   size = size + 1
}
```

The above algorithm doesn't deal with the case that the the array is full i.e. `size == length`. We need to add code to handle that case, namely we we need to make a new and bigger array. This code would go at the beginning and would insure that the condition `size < length` when the above code runs.

```
if (size == length){
   b = new array with 2 * length slots
   for (int j=0; j < length; j++)
       b[j] = a[j]                //  copy elements to bigger array
   a = b
}
```

The method `add(i, e)` only adds – that is, inserts – to a slot which holds an element in the list. What if we want to add at the end of the list? Calling `add(size, e)` will generate an index out of bounds exception, so that won't work. Instead we use just `add(e)`. See the Exercises.

Overloading

In Java, it is possible to have two methods that have the same name but whose parameters have different types. This is possible even at the code compilation stage (when the java code is translated into the lower level byte code) because variables have a declared type, so when an instruction calls a method and supplies arguments to that method, the arguments have a type and the compiler knows which version of the method is being called. We will say more about this throughout the course.

For now, note that `add` is overloaded because there is an `add(int i, E e)` and an `add(E e)` method where `e` is of type `E`. Lists in Java also have overloaded `remove` methods, where `remove(int i)` removes the element at index `i` and an `remove(E e)` removes the first occurance of an element `e`.

Adding n elements to an empty array

Suppose we wish to work with an array list. We would like to add n elements. Let's start with an array of length 1. (We wouldn't do this, but it makes the math easier to start at 1.)

```
for i = 1 to n
     add( e_i )
```

where `e_i` refers to the ith element in the list. Here I am assuming that these elements already exist somewhere else and I am just adding them to this new array list data structure. Note that the `add( e_i)` operation adds to the end of the current list.

How much work is needed to do this? In particular, how many times will we fill up a smaller array and have to create a new array of double the array size? How many copies do we need to make from the full small array to the new large arrays? It requires just a bit of math to answer this question, and it is math we will see a lot in this course. So let's do it!

If you double the array capacity k times (starting with capacity 1), then you end up with array with 2^k slots. So let's say $n = 2^k$, i.e. $k = \log_2 n$. That means we double it $k = \log_2 n$ times.

When we double the size of the underlying array and then fill it, we need to copy the elements from the smaller array to the lower half of the bigger array (and then eventually fill the upper half of the bigger array with new elements). When $k = 1$, we copy 1 element from an array of size 1 to an array of size 2 (and then add the new element in the array of size 2). When $k = 2$ and we create a new array of size 4, and then copy 2 elements (and then eventually add the 2 new elements). The number of copies from smaller to larger arrays is:

$$1 + 2 + 4 + 8 + \cdots + 2^{k-1}.$$

Note that the last term is 2^{k-1} since we are copying into the lower half of an array with capacity $n = 2^k$. See the figures in the slides.

As I showed last lecture,

$$1 + 2 + 4 + 8 + \cdots + 2^{k-1} \;=\; 2^k - 1$$

and so the number of copies from smaller to larger arrays is $2^k - 1$ or $n - 1$. So using the doubling array length scheme, the amount of work you need to do in order to add n items to an array is about twice what you would need to do if the capacity of the array was at least n at the beginning. *The advantage to using the doubling scheme rather than initializing the array to be huge is that you don't need to know in advance what the number of elements in the list will be.*

Removing an element from an arraylist is very similar to adding an element, but the steps go backwards. We again shift all elements by one position, but now we shift down by 1 rather than up by 1. The `for` loops goes forward from slot i to $size - 2$, rather than backward from $size$ to $i + 1$

`remove(i)`

```
if ((i >= 0) & (i < size)){
   tmp = a[i]                        // save it for later
   for (k = i; k < size-1; k++){
      a[k] = a[k+1]                  //  copy back by one position
   }
   size = size - 1
   a[size] = null      //  optional,  but perhaps cleaner
   return tmp
}
```

One final, general, and important point: Adding or removing from the other end of the list where `i == size` is fast, since few shift operations are necessary. However, if the array has many elements in it and if you add a new element to position 0 or you remove the element at position 0, then you need to do a lot of shifts which is inefficient. We will return to this point in the next few lectures when we compare array lists to linked lists.