ArrayList under the hood

BY TOMJEN DECEMBER, TWO THOUSAND AND FOURTEEN

```
List<string> strings = new ArrayList<string>();
```

How many times have you written something like the above?

Likely many times if you are a Java developer. ArrayList is one of the most common List types if not *the* most common and why not? It is basically an automagically re-sizing version of the array we all know and love and, as some developers know, an Array is exactly what the ArrayList use as its storage, which means that reading and random access to it is nearly as fast as a native Array and so much less likely to trip you up. In almost any case you do not need to consider how it is implemented as such details often does not matter.

However unless you know how it works, how do you know you are in a situation where how it works matters? Sure you could test to see if the ArrayList is involved in the slowdown (and you absolutely should), but you also need to be able to difference between the ArrayList taking significant time because you have a lot of work to do and the ArrayList taking a long time because you are using it in a less than optimal way.

Technically ArrayList could be implemented in any way that satisfies the standard (including requirements on performance complexity, e.g get should be O(1)), but in practice we are almost exclusively interested in how it is implemented in Suns (now Oracles) Java standard, since it is by far the most common and as such is the most important to use right.

Fortunately Oracle Software Corporation has made the source code for OpenJDK available. So today we will be looking at the source code for ArrayList and, in the case of the equals method, the class it inheriets from, AbstractList.

The ArrayList has a relatively large number of methods, so we will focus on size, empty, clear, get, add, contains and equals since these are the methods that are going to be used most.

Size is stored internally in a private variable and simply returned as a standard getter.

```
<p>Empty (which is named a bit confusingly, as it does not empty the list but simply returns <em>whether</em> the list is empty) is therefore straight-for
```

```
public boolean isEmpty() {
    return size == 0;
    }
```

This would not be the case if we were looking at the LinkedList, which is backed by a linked list, because getting the size of it is O(n) whereas simply checking if the next pointer is null is O(1); this is another reason to use the ArrayList.

```
<p>Get is also simple, as you would expect, except that it checks that the range is within the size of the arraylist (since the backing store may be bigge
```

```
public E get(int index) {
    rangeCheck(index);
    return elementData(index);
      }


</p>
```

The add method appears simple enough, but all the complexity is hidden in the ensureCapacityInternal method:

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
    }


<p>This method in turn calls the growth method (in case the internal size is not large enough), which is more complex</p>
```

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity &gt;&gt; 1);
    if (newCapacity - minCapacity &lt; 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE &gt; 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

```
<p>The code is central to performance of the arraylist, but it isn’t exactly readable, partially because it has the handle the case where the backing stor
```

What the code does is relatively simple though, it starts of by computing the new capacity as the old capacity plus (oldCapacity >> 1) which is the same as dividing it by two, rounding down. This gives a new capacity that is 1.5 times the previous size, then checks if that is enough to satisfy the requirement, otherwise it sets the total new size to whatever the requirement is, finally the code copies the old array into a new array of the newly computed size. elementData is the internal backing storage so there is now enough space allocated for the new items.

If the call to hugeCapacity happens, the capacity is set to Integer.MAX_SIZE but this is an edge case and not important for us.

Why multiply with something like 1.5 though? Why not just grow to whatever the requirement is? Or just go with 2 times that (being a nice “round” number and all)?

Lets first look at the case where we only allocated as much memory as is needed at any point. To make this easier we will not remove any items from the array list, once they are added.

Suppose we are writing some log-parser that generates an object for each log entry and we need to collect all the entries into an Array List (perhaps so that some calculations can be run on them, to generate some form of report). Under the proposed allocate-just-what-we-need scheme we would have to do the following for *each log record*:

1. Search the free chain for a free record that is big enough to contain the new array.
2. Mark it as allocated.
3. Copy all the previous entries to the new array.
4. Mark the old array as de-allocated.
5. Return the now de-allocated slice to the memory system.

Searching for a free record on the free chain in the memory allocation system is O(n), where n is the length of the free chain (*not* how much space it takes up), copying the memory is O(m) where m is the size of the memory used, inserting into the free chain may be cheap (ie O(1)) or dear (if it is implemented using something more complicated but then finding an appropriate record if often cheaper)

Oh and since this is Java, which has garbage collection, we don’t free any memory right away so the deallocated arrays will quickly take up a lot of memory, forcing frequent garbage collections. This kills performance.

If we allocate memory as needed we allocate O(n) times, while wasting no memory. By allocating twice as much memory as is needed we only allocate memory O(log(n)) times, while wasting at most 50% of the memory used — that is we get an exponential improvement in allocation at a linear cost in memory.

So far so good, but that still leaves the question of the 1.5 multiplier.

This has to do with memory fragmentation. If we assume that our program has been running for a while and keeps adding data (perhaps those log files are huge), it isn’t unlikely that only a few objects are growing. If the backing array has to grow to twice its own size then it will always be bigger than the combined sum of its previous sizes – this is true for any exponent that is at least two. This has the practical consequence that we can never reuse previously assigned memory, because it is never going to be large enough to fit the new memory requirements as such there is going to be a bigger and bigger memory area that we cannot use, leading to waste. If we only grow the size of the backing storage by 1.5 then by the third reallocation we can reuse the two previously allocated memory areas, which means that the memory fragmentation will be fairly limited.

```
<p>The clear function is comparatively simple:</p>

    public void clear() {
        modCount++;

        // Let gc do its work
        for (int i = 0; i &lt; size; i++)
            elementData[i] = null;

        size = 0;
          }


<p>The modCount is used to try to detect concurrent operations, but other than that the arraylist simply nulls out any items in its backing store (it co
<p>Finally the contains method:</p>


    public boolean contains(Object o) {
        return indexOf(o) &gt;= 0;
          }
```

The only thing that is worth nothing about this method is that it accepts Object, rather than T, which means you can test for the presence of any object. This is useful in case we have a list of lists since lists are equal so long as their content is equal and in the same order, the lists do not have to be the same type. This may also be the case for user defined objects, which is why ArrayList has been implemented to allow us to use any class. This has bitten me on a couple of occasions where I tested for the presence of some object that was a wrong type – normally Java would warn you if you tried to do that.

```
  <p>Since contains use the indexOf method, lets look at that too:</p>


      public int indexOf(Object o) {
          if (o == null) {
              for (int i = 0; i &lt; size; i++)
                  if (elementData[i]==null)
                      return i;
          } else {
              for (int i = 0; i &lt; size; i++)
                  if (o.equals(elementData[i]))
                      return i;
          }
          return -1;
            }
```

The only interesting thing here is that in case the object is null, we test the objects against null, otherwise call their equals function with the object as argument. In either case we break when we have a match and return -1 otherwise.

```
  <p>If you grep the source for equals you won’t find an implementation of the equals method. It is instead implemented in the AbstractList class, from wh

      public boolean equals(Object o) {
          if (o == this)
              return true;
          if (!(o instanceof List))
              return false;

          ListIterator&lt;E&gt; e1 = listIterator();
          ListIterator e2 = ((List) o).listIterator();
          while (e1.hasNext() &amp;&amp;; e2.hasNext()) {
              E o1 = e1.next();
              Object o2 = e2.next();
              if (!(o1==null ? o2==null : o1.equals(o2)))
                  return false;
          }
          return !(e1.hasNext() || e2.hasNext());
            }
```

The test for whether the other object is reference equal to us is pretty standard, the interesting thing happens if it is a list. Since we do not know what type of list it is we don’t know if taking the size of the list is an expensive operation (which it would be if either of the lists were LinkedLists) or cheap (which it is in the ArrayList) so the equals method simply traverses an iterator from each list and checks that each element is either null or equal to the other and that, at the end, the iterators have both been run through completely.

Since ArrayLists are fairly common we could optimize the ArrayList class by overriding the equals method and have it check if the other object is an ArrayList too – and if so, start by comparing their size, then skip the creation of the iterators and compare each item in their list with a call to get (saving having to create and then garbage collect two objects), because we know this is cheap.

This in an optimization that I have done a couple of times to shave a few milliseconds of the inner interation of a loop that were going to be executed tens of thousands of times with a user waiting on the results, where it is well worth it. For almost all your development needs however, you can completely ignore how the equals method works.

But at least you know.

---

1. Clever code is often a code smell on its own, but in this case the extra performance gained is likely worth it, considering how commonly used the ArrayList is. ↩

</string></string>