Lec 6: Array Lists

Sam McCauley

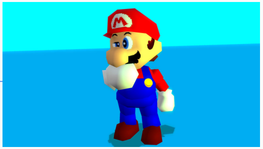
February 20, 2026

Admin

- Any questions before we get started?

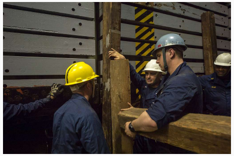
Array Lists

Arrays



- What's *wrong* with arrays in Java? What's the biggest downside?
- Probably: can't resize them!
- Also: don't support helpful methods like "is this element in the array"
- Today: write our own Java class to address these issues

ArrayList



- We’ll write a class called `ArrayListInt` that can support list-like operations on a sequence of integers, using an array as a back end
- Java has an `ArrayList` library that we’ll use later in the course
    - We’ll name our methods to match theirs, and mostly match their functionality
- Point of today: by building it ourselves, can get insight into *how* it works
- Will help in the next few weeks as we begin to compare data structure performance

ArrayList Functionality: Array Operations

Array-like functionality:

- `int size()`: returns the number of elements stored in our list
- `int get(int index)`: returns the element stored in slot `index`
- `int set(int index, int newElement)`: sets the element stored at `index` to be `newElement`. Returns the element *previously* stored at `index`.

ArrayList Functionality: Expanding/Contracting the List

- `void add(int newElement)`: adds `newElement` to the end of the list.[1]
- `void add(int index, int newElement)`: adds `newElement` to slot `index` in the list, pushing all later elements in the list down by one slot.
- `int remove(int index)`: removes the element at index `index`, moving all later elements in the list up by one slot. Returns the removed element.

[1] In the real `ArrayList`, add returns a `boolean` value that is always `true`, for reasons that don't affect our implementation.

ArrayList Functionality: Search

- `int indexOf(int element):` returns the first index in the list that contains `element`, or -1 if there is no such index (the list does not contain the element).

- `boolean contains(int element)`: returns `true` if `element` is in the list, and `false` otherwise.

Let’s Strategize Before Coding

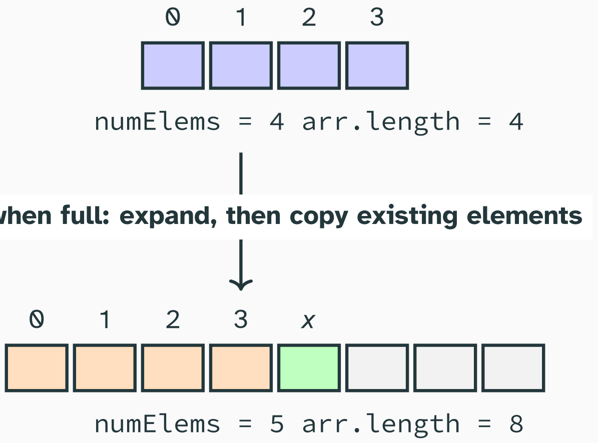
What Should We Store?

- Need an array of `ints` to store our elements
- When we discuss resizing, we'll see we need one more piece of data:
    - How many elements are currently stored in the array
    - Eventually: may not be the same as the length of the array

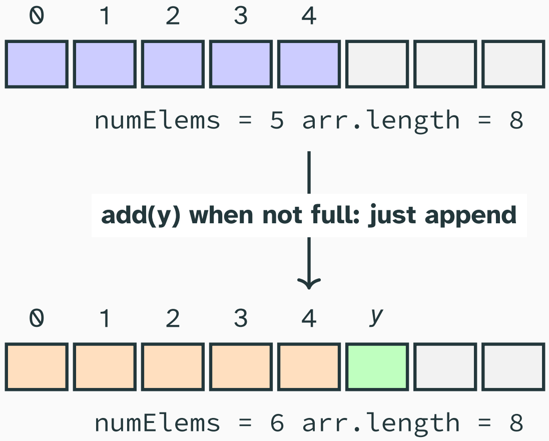
Resizing an Array

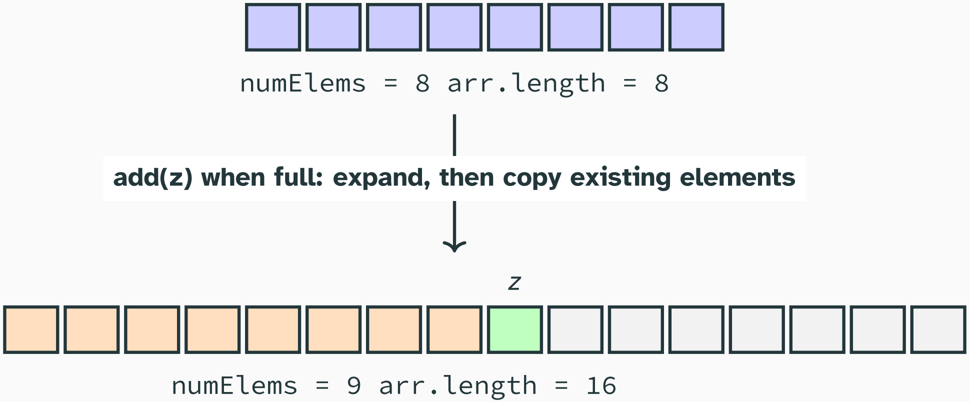
- What do we have to do when we want to `add()` an element to an array that's already full?
  - We have to build a new, larger array!
  - Must *copy* all elements from the smaller array into the larger array. (This is expensive.)
- How big should it be? Any ideas?
  - Strategy 1: make the array one slot larger
  - Strategy 2: *double* the size of the array

Let’s look at Strategy 2 (doubling) in more detail


0
1
2
3
numElems = 4 arr.length = 4
add(x) when full: expand, then copy existing elements
0
1
2
3
x
numElems = 5 arr.length = 8

Blue cells were full before the `add()`. The brown cells are “copied” elements. The green cell is where the new element goes. The white cells are empty.


0 1 2 3 4
numElems = 5 arr.length = 8
add(y) when not full: just append
0 1 2 3 4 y
numElems = 6 arr.length = 8


numElems = 8 arr.length = 8
add(z) when full: expand, then copy existing elements
z
numElems = 9 arr.length = 16

Strategy 1 Discussion

Let’s say we make the array one slot larger to handle the new element

In pairs: can you come up with an upside of this approach? A downside?

- Upside: very space-efficient! The array is only as big as is necessary

- Downside: need to copy over all elements *every* time we `add()`

Strategy 2 Discussion

Let’s say we *double* the array size to handle the new element

In pairs: can you come up with an upside of this approach? A downside?

- Downside: space-inefficient! The array can be twice as big as it needs to be
- Downside: We’ll have “empty” slots we need to track in our class
- Upside: far fewer copies. Once we copy, we don’t need to copy again until the array size doubles

Quantitative Comparison

Let's say we call `add()` 1 million times. Let's look at the cost of each strategy.

- If we grow the array by 1 every time, we will copy 1 element, then 2, then 3, then 4, up to 999,999. The total number of elements we copy is:

  $$1 + 2 + 3 + \ldots + 999,999 = 499,999,500,000.$$

  So just short of 500 billion elements copied.
- If we double the size of the array every time, we will copy 1 element, then 2, then 4, then 8, then 16, up to 524,288. The total number of elements we copy is:

  $$1 + 2 + 4 + 8 + \ldots + 524,288 = 1,048,575.$$

  So our total number of elements copied is just over 1 million.
- Doubling is, in this case, 500,000 times faster than adding 1 slot each time. Likely worth the extra space usage!

Let’s fill in the ArrayListInt class together

Important Takeaways

- When we have the same code twice, should *factor it out* into a new method
  - Does improve code length (not a priority)
  - More importantly: improves organization; makes corrections easier!
-