array list

motivation

limitations of arrays

- think back to your days of using lists in Python
- how do they compare to arrays in Java?
- what are the pros/cons of arrays?

arrays are fixed-length

- 🔄 an **array** is a fixed-length sequence of elements all of the same type
  - 🙂 simple, fast ($\mathcal{O}(1)$ access)
  - ☹️ fixed-length (what if we don't know how many elements we'll have?)
    - **solution A:** make an array that's longer than you'll ever need
      - 🔄 `bullets = new Bullet[256]; // from HW-03`
      - 🙂 simple
      - ☹️ wastes space, might end up not actually be long enough
    - **solution B:** make an array that *grows as needed* (an **array list**)

array list

array list

- an **array list** (dynamic array, stretchy buffer, **vector**) acts like an array that can grow as needed
  - like an array, the user can **access** the i-th element in an array list
    - **get** the value of an element that is already there
    - **set** the value of an element that is already there
  - unlike an array, the user can always **add** a new element to an array list, no matter how many elements it already has
    - **append** (push back) a new element to the back or end
    - **insert** a new element at any index
  - the user can also **remove** elements from an array list

implementing array list

- what **properties** of lists do we care about?
  - The items
  - Total number of items
- what **actions** do we perform on lists?
  - adding item to end
  - removing item(s)
  - inserting item in middle
  - maybe sorting items? we'll discuss this later…
- these features will guide our implementation
  - properties: **instance variables**
  - actions: **instance methods**

the internal (private) array

- an array list stores its elements inside of an **array**
  - an array list's **capacity** is the length of this array
  - an array list's **size** (**length**) is the number of (non-null) elements currently actually stored in this array

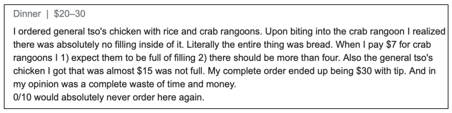
```
class ArrayList<ElementType> {
    private ElementType[] privateArray;
    private int numberOfElementsActuallyStoredInPrivateArray;
    // int capacity() { return privateArray.length; }
    int size() {
        return numberOfElementsActuallyStoredInPrivateArray;
    };
}
```

array list

- ☠ **an array list's capacity is NOT the same thing as its size**
  - imagine `ArrayList<String> bestaurants;` with capacity 5 and size 3

```
// if we could print bestaurants.privateArray...
["Blango", "Sproot", "Sparket", null, null]
```

...we would see 5 – 3 = 2 "empty slots."



Dinner | $20–30

I ordered general tso's chicken with rice and crab rangoons. Upon biting into the crab rangoon I realized there was absolutely no filling inside of it. Literally the entire thing was bread. When I pay $7 for crab rangoons I 1) expect them to be full of filling 2) there should be more than four. Also the general tso's chicken I got that was almost $15 was not full. My complete order ended up being $30 with tip. And in my opinion was a complete waste of time and money.
0/10 would absolutely never order here again.

array list

- ☠ **even though another name for an array list is a "vector," the array list is NOT related to Vector2 (vector from math/physics)**

Java's `ArrayList<ElementType>`

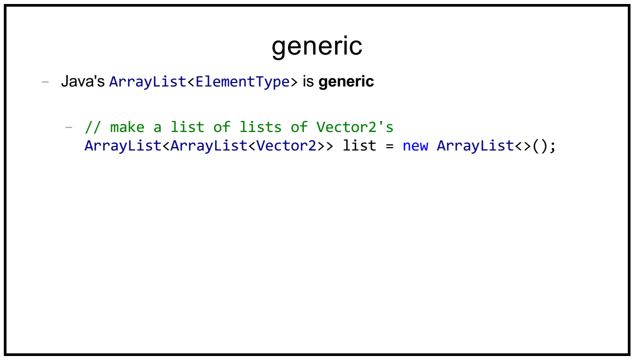
generic

- Java's `ArrayList<ElementType>` is **generic**

Replace ElementType with the type of object being stored in your array list

-
  ```
  // make a list of Foo's
  ArrayList<Foo> list = new ArrayList<>();
  ArrayList<Foo> list = new ArrayList<Foo>();
  ```

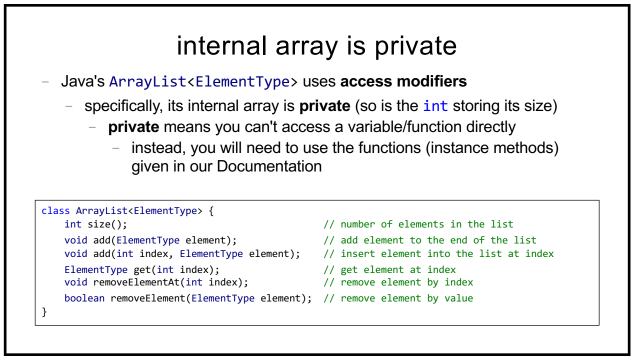
-
  ```
  // make a list of int's
  // NOTE: you must use Integer instead of int inside the <>
  ArrayList<Integer> list = new ArrayList<>();
  ```
- `ArrayList<Boolean> list = new ArrayList<>();`
- `ArrayList<Double> list = new ArrayList<>();`



generic

- Java's `ArrayList<ElementType>` is **generic**

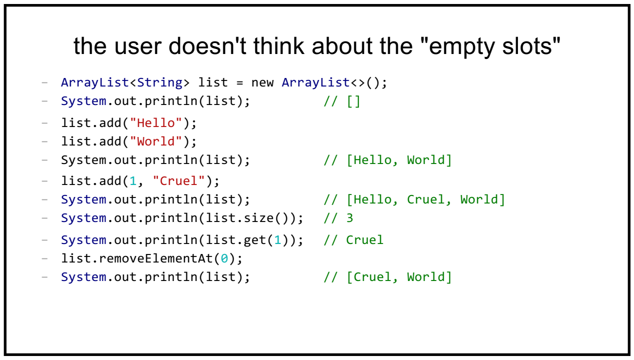
  -
    ```
    // make a list of lists of Vector2's
    ArrayList<ArrayList<Vector2>> list = new ArrayList<>();
    ```



internal array is private

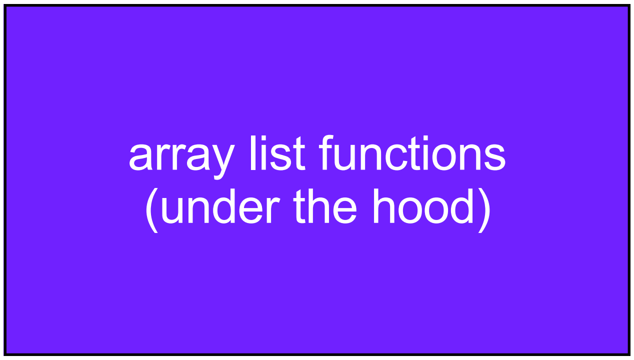
- Java's `ArrayList<ElementType>` uses **access modifiers**
  - specifically, its internal array is **private** (so is the `int` storing its size)
    - **private** means you can't access a variable/function directly
      - instead, you will need to use the functions (instance methods) given in our Documentation

```
class ArrayList<ElementType> {
    int size();                                  // number of elements in the list
    void add(ElementType element);               // add element to the end of the list
    void add(int index, ElementType element);    // insert element into the list at index
    ElementType get(int index);                  // get element at index
    void removeElementAt(int index);             // remove element by index
    boolean removeElement(ElementType element);  // remove element by value
}
```

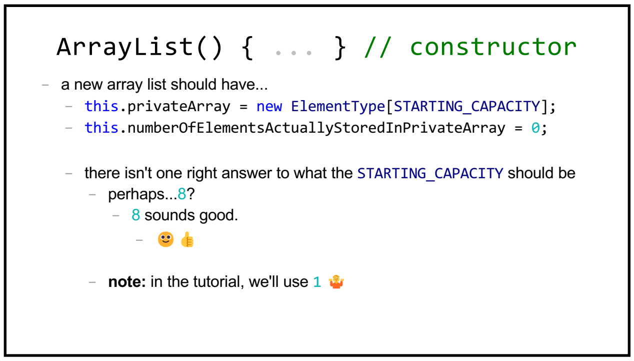


the user doesn't think about the "empty slots"

```
- ArrayList<String> list = new ArrayList<>();
- System.out.println(list);         // []
- list.add("Hello");
- list.add("World");
- System.out.println(list);         // [Hello, World]
- list.add(1, "Cruel");
- System.out.println(list);         // [Hello, Cruel, World]
- System.out.println(list.size());  // 3
- System.out.println(list.get(1));  // Cruel
- list.removeElementAt(0);
- System.out.println(list);         // [Cruel, World]
```



array list functions (under the hood)

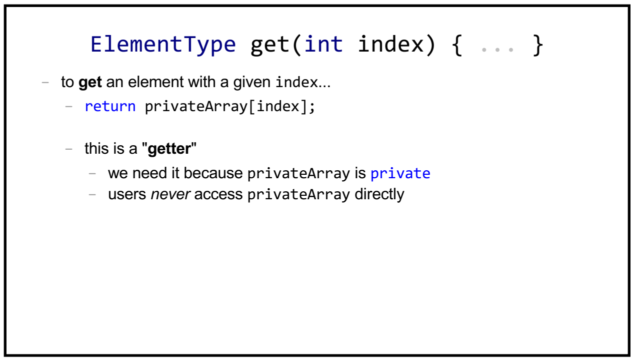


`ArrayList() { ... } // constructor`

- a new array list should have...
  - `this.privateArray = new ElementType[STARTING_CAPACITY];`
  - `this.numberOfElementsActuallyStoredInPrivateArray = 0;`

  - there isn't one right answer to what the `STARTING_CAPACITY` should be
    - perhaps...8?
      - 8 sounds good.
        - 🙂 👍

    - **note:** in the tutorial, we'll use 1 🤷



`ElementType get(int index) { ... }`

- to **get** an element with a given `index`...
  - `return privateArray[index];`

  - this is a **"getter"**
    - we need it because `privateArray` is `private`
    - users *never* access `privateArray` directly

`void add(ElementType element) { ... }`

- to **append** (push back) a new element to the back (end) of an array list...
  - write the new element to the first available empty slot in `privateArray`
  - increment `numberOfElementsActuallyStoredInPrivateArray`

- potential problems?
  - what if `privateArray` is full?
  - need to make room, but how?

array list

- an **array list** (dynamic array, stretchy buffer, **vector**) acts like an array that can grow as needed
  - like an array, the user can **access** the i-th element in an array list
    - **get** the value of an element that is already there
    - **set** the value of an element that is already there
  - unlike an array, the user can always **add** a new element to an array list, no matter how many elements it already has
    - **append** (push back) a new element to the back or end
    - **insert** a new element at any index
  - the user can also **remove** elements from an array list

generic

- Java's `ArrayList<ElementType>` is **generic**

Replace ElementType with the type of object being stored in your array list

```
// make a list of Foo's
ArrayList<Foo> list = new ArrayList<>();
ArrayList<Foo> list = new ArrayList<Foo>();
```

```
// make a list of int's
// NOTE: you must use Integer instead of int inside the <>
ArrayList<Integer> list = new ArrayList<>();
```

```
ArrayList<Boolean> list = new ArrayList<>();
```

```
ArrayList<Double> list = new ArrayList<>();
```

internal array is private

- Java's `ArrayList<ElementType>` uses **access modifiers**
  - specifically, its internal array is **private** (so is the `int` storing its size)
    - **private** means you can't access a variable/function directly
      - instead, you will need to use the functions (instance methods) given in our Documentation

```
class ArrayList<ElementType> {
    int size();                                    // number of elements in the list
    void add(ElementType element);                 // add element to the end of the list
    void add(int index, ElementType element);      // insert element into the list at index
    ElementType get(int index);                    // get element at index
    void removeElementAt(int index);               // remove element by index
    boolean removeElement(ElementType element);    // remove element by value
}
```

`ArrayList() { ... } // constructor`

- a new array list should have...
  - `this.privateArray = new ElementType[STARTING_CAPACITY];`
  - `this.numberOfElementsActuallyStoredInPrivateArray = 0;`

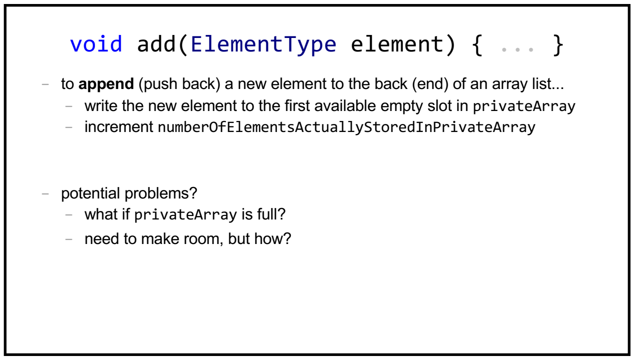
  - there isn't one right answer to what the `STARTING_CAPACITY` should be
    - perhaps...8?
      - 8 sounds good.
        - 🙂 👍

    - **note:** in the tutorial, we'll use 1 🤷

`ElementType get(int index) { ... }`

- to **get** an element with a given `index`...
  - `return privateArray[index];`

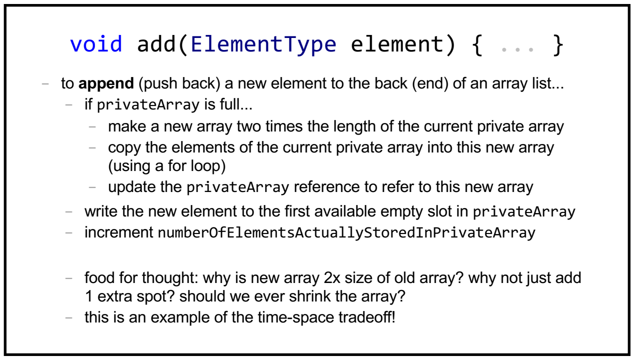
  - this is a **"getter"**
    - we need it because `privateArray` is `private`
    - users *never* access `privateArray` directly



`void add(ElementType element) { ... }`

- to **append** (push back) a new element to the back (end) of an array list...
  - write the new element to the first available empty slot in `privateArray`
  - increment `numberOfElementsActuallyStoredInPrivateArray`

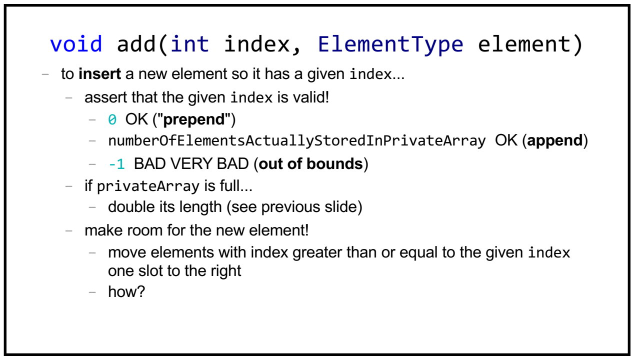
- potential problems?
  - what if `privateArray` is full?
  - need to make room, but how?



`void add(ElementType element) { ... }`

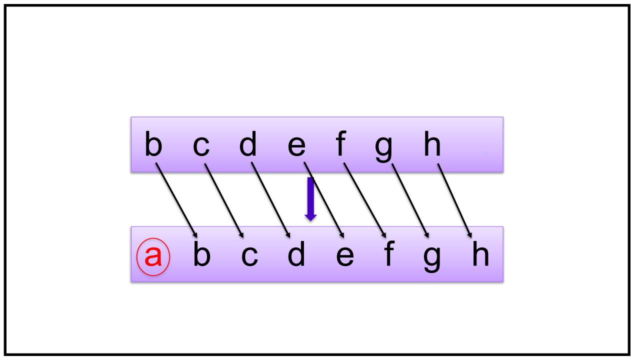
- to **append** (push back) a new element to the back (end) of an array list...
  - if `privateArray` is full...
    - make a new array two times the length of the current private array
    - copy the elements of the current private array into this new array (using a for loop)
    - update the `privateArray` reference to refer to this new array
  - write the new element to the first available empty slot in `privateArray`
  - increment `numberOfElementsActuallyStoredInPrivateArray`

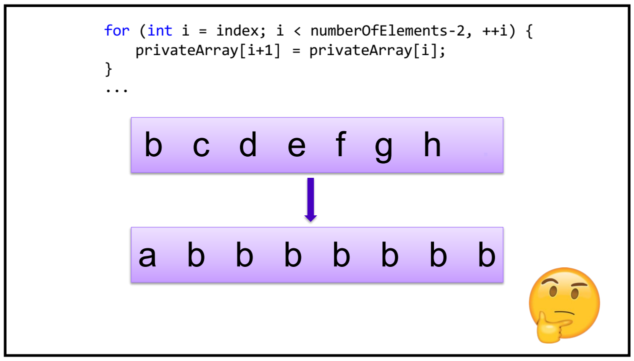
  - food for thought: why is new array 2x size of old array? why not just add 1 extra spot? should we ever shrink the array?
  - this is an example of the time-space tradeoff!



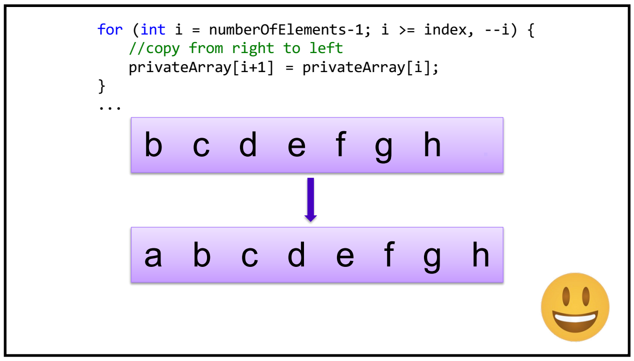
`void add(int index, ElementType element)`

- to **insert** a new element so it has a given `index`...
  - assert that the given `index` is valid!
    - `0` OK ("**prepend**")
    - `numberOfElementsActuallyStoredInPrivateArray` OK (**append**)
    - `-1` BAD VERY BAD (**out of bounds**)
  - if `privateArray` is full...
    - double its length (see previous slide)
  - make room for the new element!
    - move elements with index greater than or equal to the given `index` one slot to the right
    - how?

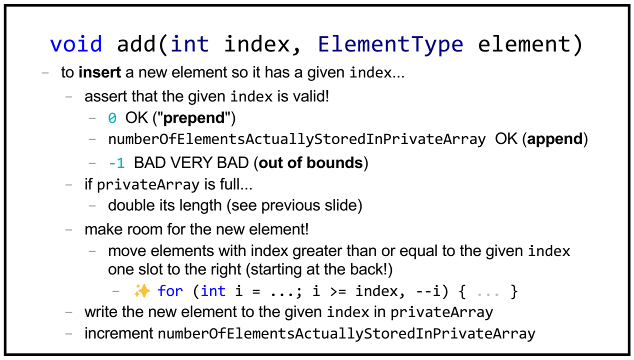




```
for (int i = index; i < numberOfElements-2, ++i) {
    privateArray[i+1] = privateArray[i];
}
...
```

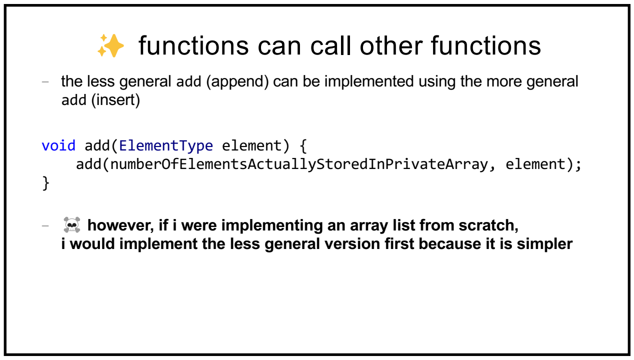


```
for (int i = numberOfElements-1; i >= index, --i) {
    //copy from right to left
    privateArray[i+1] = privateArray[i];
}
...
```



`void add(int index, ElementType element)`

- to **insert** a new element so it has a given `index`...
  - assert that the given `index` is valid!
    - `0` OK ("**prepend**")
    - `numberOfElementsActuallyStoredInPrivateArray` OK (**append**)
    - `-1` BAD VERY BAD (**out of bounds**)
  - if `privateArray` is full...
    - double its length (see previous slide)
  - make room for the new element!
    - move elements with index greater than or equal to the given `index` one slot to the right (starting at the back!)
      - ✨ `for (int i = ...; i >= index, --i) { ... }`
  - write the new element to the given `index` in `privateArray`
  - increment `numberOfElementsActuallyStoredInPrivateArray`

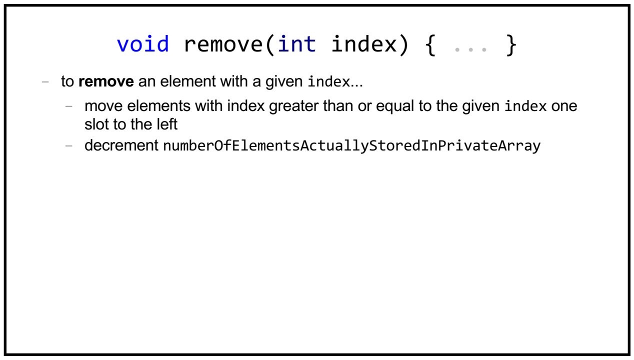


✨ functions can call other functions

- the less general add (append) can be implemented using the more general add (insert)

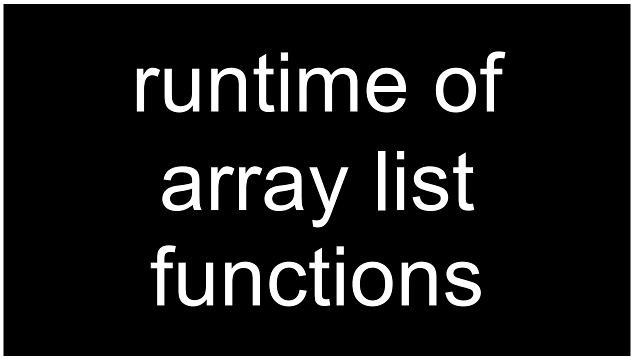
```
void add(ElementType element) {
    add(numberOfElementsActuallyStoredInPrivateArray, element);
}
```

- ☠️ **however, if i were implementing an array list from scratch, i would implement the less general version first because it is simpler**

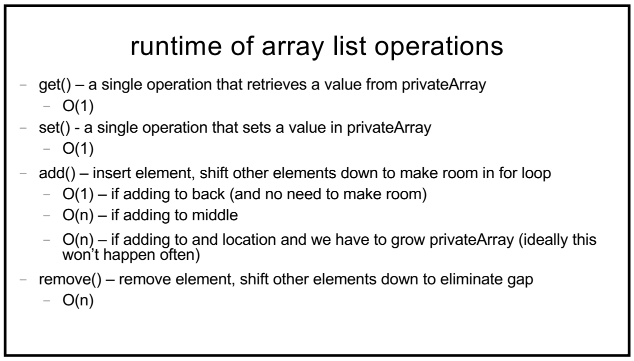


`void remove(int index) { ... }`

- to **remove** an element with a given `index`...
  - move elements with index greater than or equal to the given `index` one slot to the left
  - decrement `numberOfElementsActuallyStoredInPrivateArray`

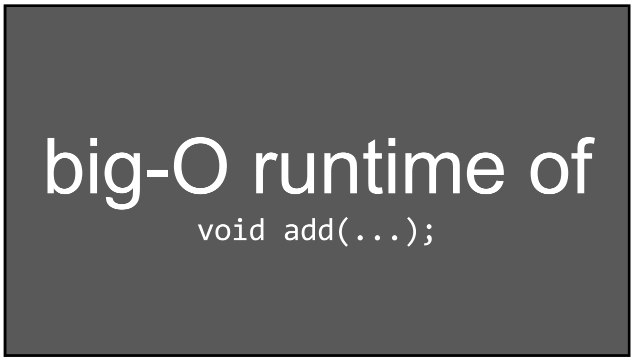


runtime of array list functions



runtime of array list operations

- get() – a single operation that retrieves a value from privateArray
  - O(1)
- set() - a single operation that sets a value in privateArray
  - O(1)
- add() – insert element, shift other elements down to make room in for loop
  - O(1) – if adding to back (and no need to make room)
  - O(n) – if adding to middle
  - O(n) – if adding to and location and we have to grow privateArray (ideally this won't happen often)
- remove() – remove element, shift other elements down to eliminate gap
  - O(n)



big-O runtime of

`void add(...);`

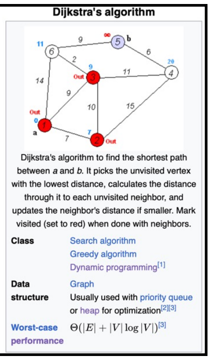
big-O runtime

- **big-O runtime** (**running time**, **time complexity**) gives us a big-picture idea of how long a function will take to run (execute, finish)
  - a function that operates on n *things,* could be...

$\mathcal{O}(1)$ (**constant time**)	FAST
$\mathcal{O}(\log_2 n)$ (**logarithmic time**, **log time**)	fast
$\mathcal{O}(n)$ (**linear time**)	meh
$\mathcal{O}(n^2)$ (**quadratic time**)	slow
$\mathcal{O}(2^n)$ (**exponential time**)	SLOW

big-O runtime

- **big-O runtime** can get complicated
  - sometimes there aren't just n *things*...
    - there are n thing A's, m thing B's, etc.
      - 🔮 **stay tuned for CSCI 256!**
  - sometimes we get more specific about *when* a function takes a certain length of time to run
    - in the **worst case**?
    - in the **best case**?
    - ~~in the **average case**?~~
    - in the "long run"? (**amortized**)



Dijkstra's algorithm

Dijkstra's algorithm to find the shortest path between *a* and *b*. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

Class	Search algorithm Greedy algorithm Dynamic programming[1]
Data structure	Graph Usually used with priority queue or heap for optimization[2][3]
Worst-case performance	$\Theta(\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert)$[3]

best case and worst case runtime of adding to the back of an array list

best case

- the **best case** is the shortest time a function can possibly take to run
  - for adding an element to the back of an array list, this is when the array list's internal/private array still has at least one "empty slot"
    - in this case, we have to...
      - write an element to an array $\mathcal{O}(1)$
      - increment a counter + $\mathcal{O}(1)$
      - ---
      - $\mathcal{O}(1)$ 🙂

worst case

- the **worst case** is the longest time a function can possibly take to run
  - for adding an element to the back of an array list, this is when the array list's internal/private array is full (with n elements)
    - in this case, we have to...
      - allocate an array of $2n$ elements $\mathcal{O}(n)$
      - copy of over n elements + $\mathcal{O}(n)$
      - write an element to an array + $\mathcal{O}(1)$
      - increment a counter + $\mathcal{O}(1)$
      - updating reference to internal array + $\mathcal{O}(1)$
      - ---
      - $\mathcal{O}(n)$ 😐

sometimes the best case and worst case are very different

amortized runtime of adding to the back of an array list

amortized

- the **amortized runtime** is how long a function takes (on average) "in the long run"
  - for adding an element to the back of an array list...

		□
wasn't full	1	■
was full	2	■■
was full	4	■■■□
wasn't full	1	■■■■
was full	8	■■■■■□□□
wasn't full	1	■■■■■■□□
wasn't full	1	■■■■■■■□
wasn't full	1	■■■■■■■■
was full	16	■■■■■■■■■□□□□□□□
	...	
	1	■■■■■■■■■■■■■■■■...■■■■

amortized

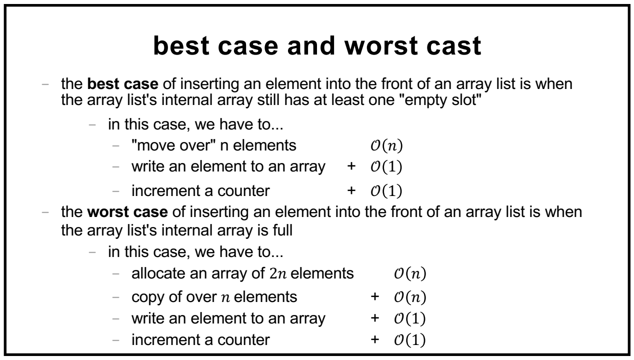
- the **amortized runtime** is how long a function takes (on average) "in the long run"
  - for adding an element to the back of an array list...
    - $\frac{1+2+4+1+8+1+1+1+16+1+1+1+1+1+1+1+32+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+\cdots}{n}$
    - $\frac{\mathcal{O}(1+\cdots+1)}{n}+\frac{\mathcal{O}(1+2+4+8+16+\cdots n)}{n}$
    - ...
    - $\frac{\mathcal{O}(n)}{n}+\frac{\mathcal{O}(n)}{n}$
    - $\mathcal{O}(1)$ 🙂

the amortized run time is less "pessimistic" than worst case

the most relevant runtime depends on **context**

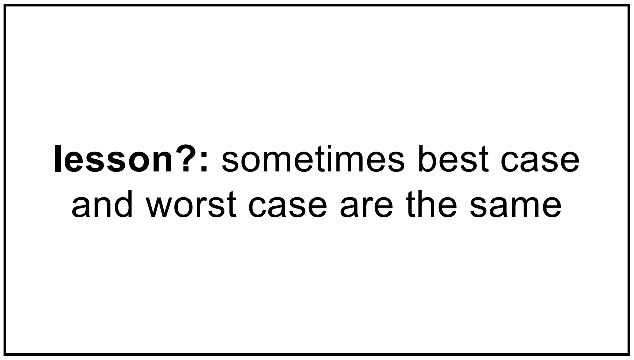
💭 when might worst case be more relevant?
💭 when might amortized be more relevant?

best case and worst case runtime of inserting into the *front* of an array list



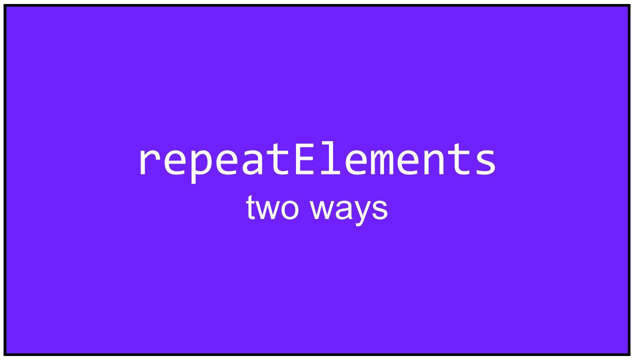
best case and worst cast

- the **best case** of inserting an element into the front of an array list is when the array list's internal array still has at least one "empty slot"
  - in this case, we have to...
    - "move over" n elements $O(n)$
    - write an element to an array + $O(1)$
    - increment a counter + $O(1)$
- the **worst case** of inserting an element into the front of an array list is when the array list's internal array is full
  - in this case, we have to...
    - allocate an array of $2n$ elements $O(n)$
    - copy of over n elements + $O(n)$
    - write an element to an array + $O(1)$
    - increment a counter + $O(1)$



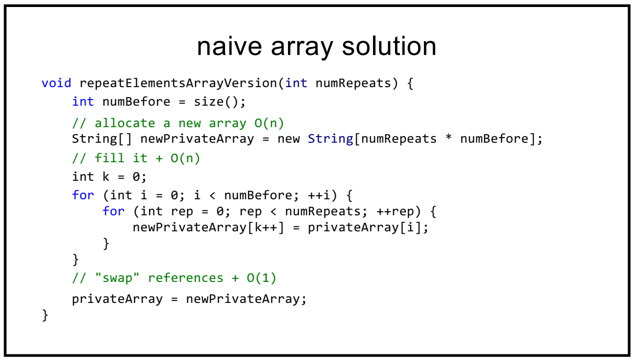
lesson?: sometimes best case and worst case are the same

abstraction



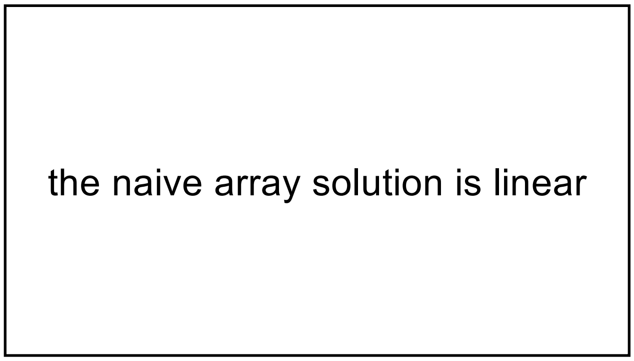
`repeatElements`

two ways



naive array solution

```
void repeatElementsArrayVersion(int numRepeats) {
    int numBefore = size();
    // allocate a new array O(n)
    String[] newPrivateArray = new String[numRepeats * numBefore];
    // fill it + O(n)
    int k = 0;
    for (int i = 0; i < numBefore; ++i) {
        for (int rep = 0; rep < numRepeats; ++rep) {
            newPrivateArray[k++] = privateArray[i];
        }
    }
    // "swap" references + O(1)
    privateArray = newPrivateArray;
}
```



the naive array solution is linear

naive array list solution

```
void repeatElements(int numRepeats) {
    int numBefore = size();
    for (int i = 0; i < numBefore; ++i) { // n *
        int index = i * numRepeats;
        for (int rep = 0; rep < (numRepeats - 1); ++rep) {
            add(index, get(index)); // insert an element
        }                           //  O(n) because we have to
    }                               //  "move stuff over"
}
```

the naive array list solution is $n * O(n) = O(n^2)$ (quadratic)!! ☹

🧪

lesson: ignoring what's actually happening under the hood is dangerous!