University of Wisconsin
LA CROSSE
Computer Science

Week 06:
Collections and ArrayLists

CS 220: Software Design II — D. Mathias

Arrays

- Primitive data structure
- Pros
  - straightforward
  - universal
  - quick access time
- Cons
  - cannot dynamically grow/shrink
  - can waste memory
  - requires high overhead to manage

Many, Many Data Structures

- Numerous other data structure options besides arrays
  - different structures, different rules, different tradeoffs
- In Java, many implement the `Collection` interface
  - some do not, but these tend to be highly specialized

- Review: What is an interface?

The `Collection` Interface

```
Collection
{interface}

+ *add(E e) : boolean
+ *clear()
+ contains(Object o) : boolean
+ equals(Object o) : boolean
+ hashCode() : int
+ isEmpty() : boolean
+ iterator() : Iterator<E>
+ *remove(Object o) : boolean
+ size() : int
+ toArray() : Object[]
```

Describes **what we can do** with a data structure

Works effectively only if the object stored in the collection has implemented both `equals()` and `compareTo(Object o)`

Optional methods (marked by *) that aren't implemented may throw an UnsupportedOperationException

The list to the left is not complete

Abstract Data Types

abstract data types (*ADT*) describe how methods should modify the stored data, without specifying what underlying actions are required

Example

- `add(E e)` should add a new item to the data structure
- how the data is stored and where in the structure the data is added are undefined

Focuses on the interface, **not** the implementation

A Basic Guide to Data Structures

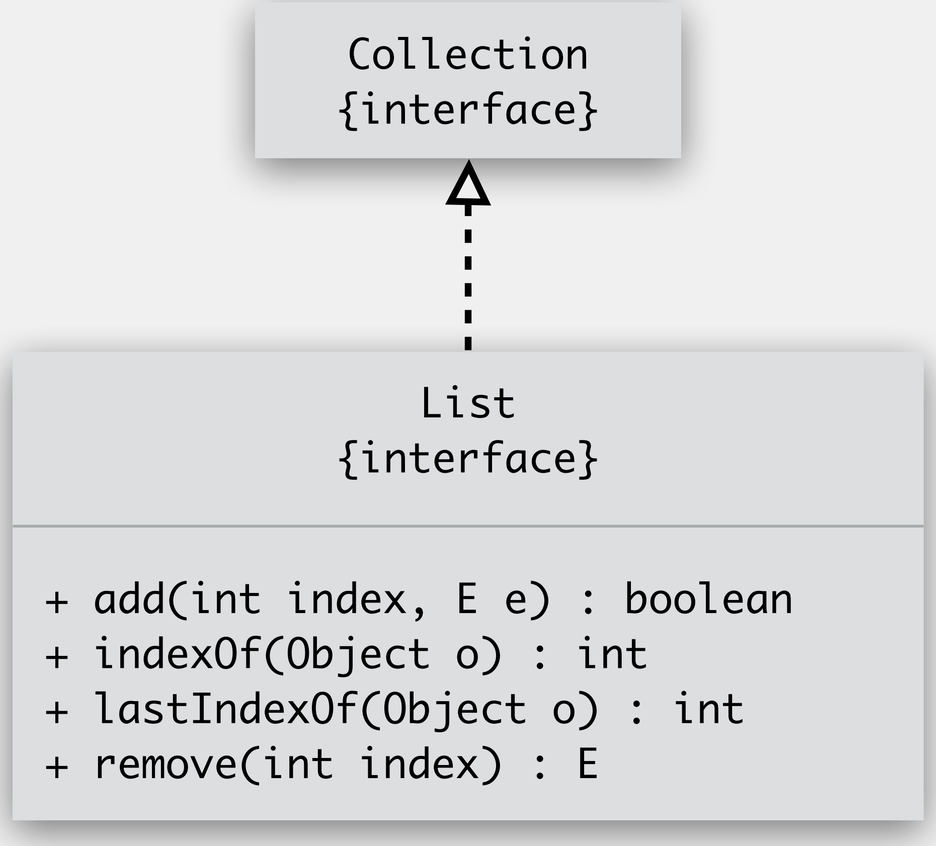
Common data structures are defined by two components

- the *interface* that describes what they do
- the *implementation* that describes how they do it

implementation

interface	resizable array	linked list	hash table	balanced tree
set			`HashSet`	`TreeSet`
list	`ArrayList`	`LinkedList`		
map			`HashMap`	`TreeMap`

The Humble List ADT



Holds data in a linear fashion

We can use `Collection` and `List` interfaces to ask questions

- what is the last index of a particular value?
- is the list empty?
- how many values are in there?

At least two different ways to implement

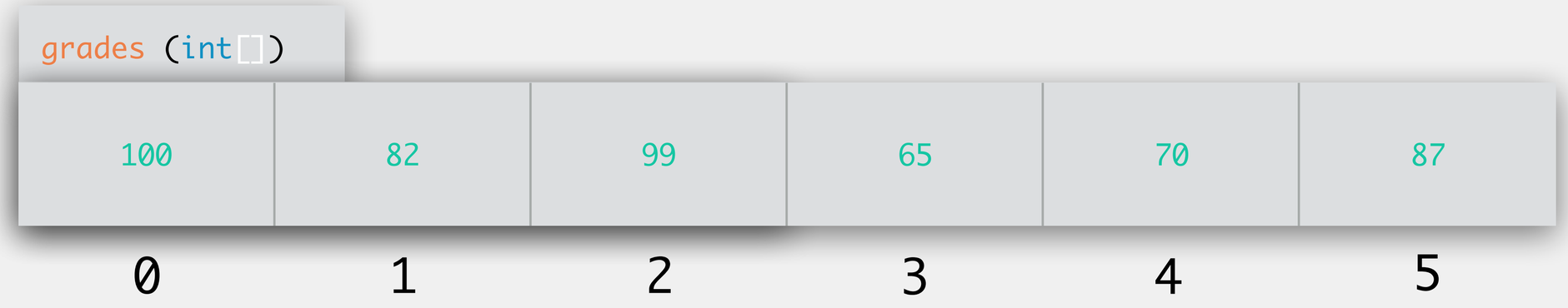
- array
- linked nodes

ArrayList

Class that implements list methods by storing values in an array

Abstracts away many of the actions we manually do

- growing/(shrinking?) the array
- shifting values up/down



Caveat: We will assume our `ArrayList` holds all `String` values

(we will rectify this later to adapt to other types)

ArrayList: Attributes

```
ArrayList

- DEFAULT_CAPACITY : int
- data : String[]
- size : int

...
```

`DEFAULT_CAPACITY`: starting array size

- `final`: can never be changed
  - indicated by all capital spelling/underscore
- `static`: underlined

`data`: array storing the data

`size`: current number of data entries

- starts at 0
- will also describe next empty index

ArrayList: Methods

```
ArrayList
---
...
---
+ add(E e) : boolean
+ add(int index, E e) : void
+ contains(Object o) : boolean
+ ensureCapacity(int minCapacity)
+ equals(Object o) : boolean
+ hashCode() : int
+ indexOf(Object o) : int
+ isEmpty() : boolean
+ iterator() : Iterator<E>
+ lastIndexOf(Object o) : int
+ remove(int index) : E
+ remove(Object o) : boolean
+ size() : int
...
```

Some methods required by `Collection`, some required by `List`

Some unique to `ArrayList`

- e.g., `ensureCapacity(int minCapacity)` allows us to set a minimum capacity for the `ArrayList`

No way for user to define how the `ArrayList` grows

- in practice, grows by 50% when needed

`ArrayList` never shrinks on its own

- `trimToSize()` method will shrink to current size

ArrayList: Constructor

arguments: nothing *or* a single `int` representing the initial array size

returns: a new `ArrayList` object

```
new ArrayList();
new ArrayList(<int>);
```

```
>ArrayList defaultArr = new ArrayList();
>ArrayList arr6 = new ArrayList(6);
>
```

memory

defaultArr (ArrayList) →

```
ArrayList
data = {null,null,null,
        null,null,null,
        null,null,null,
        null}
size = 0
```

arr6 (ArrayList) →

```
ArrayList
data = {null,null,null,
        null,null,null}
size = 0
```

ArrayList: Add Methods

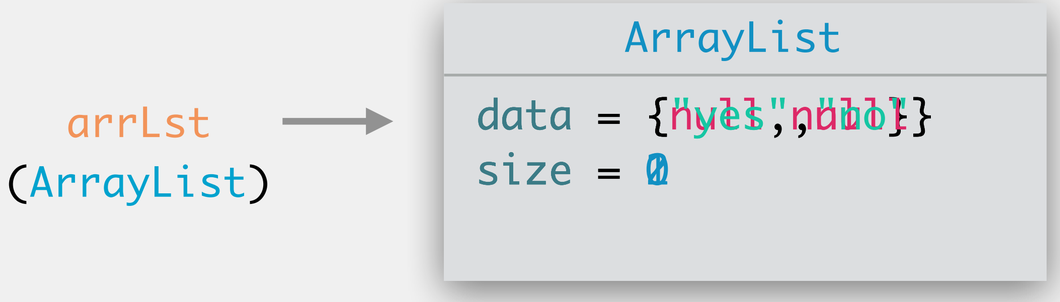
arguments: String to add

returns: nothing

behavior: adds value to end of ArrayList; increases size if required

```
>ArrayList arrLst = new ArrayList(2);
>arrLst.add("yes");
>arrLst.add("no");
>arrLst.add("maybe");
```

memory



ArrayList: Add Methods

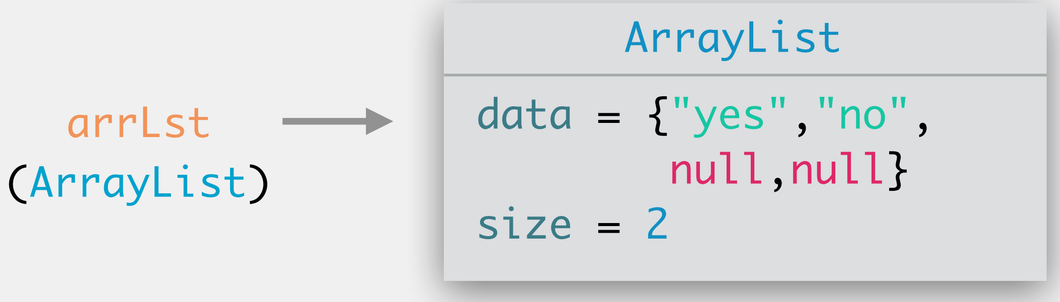
arguments: String to add

returns: nothing

behavior: adds value to end of ArrayList; increases size if required

```
ArrayList arrLst = new ArrayList(2);
arrLst.add("yes");
arrLst.add("no");
arrLst.add("maybe");
>
```

memory



ArrayList: Add Methods

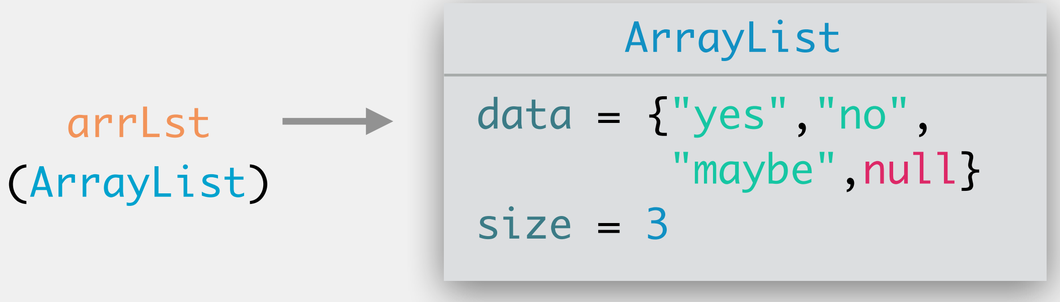
arguments: String to add

returns: nothing

behavior: adds value to end of ArrayList; increases size if required

```
ArrayList arrLst = new ArrayList(2);
arrLst.add("yes");
arrLst.add("no");
arrLst.add("maybe");
>
```

memory



ArrayList: Add Methods

arguments: index (`int`) position to add to, String to add

returns: `boolean` as to whether the add was successful

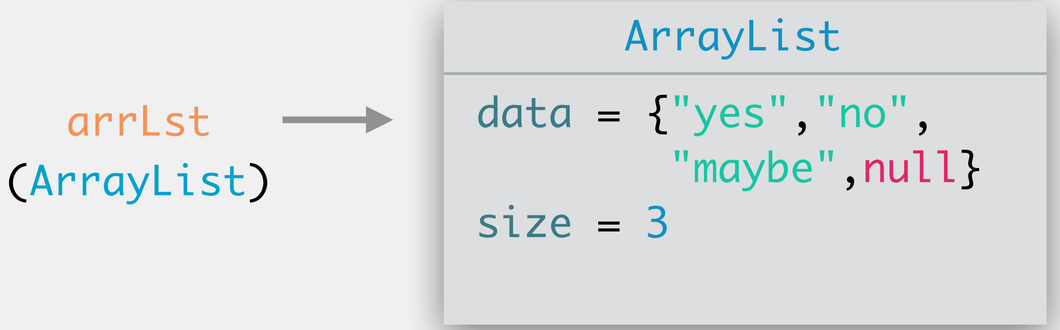
will always return true for ArrayList

throws: `IndexOutOfBoundsException` if index is negative or greater than size

behavior: adds value to index; shifts values down; increases size if required

```
  ArrayList arrLst = new ArrayList(2);
  arrLst.add("yes");
  arrLst.add("no");
  arrLst.add("maybe");
>arrLst.add(0, "unknown");
```

memory



ArrayList: Add Methods

arguments: index (`int`) position to add to, String to add

returns: `boolean` as to whether the add was successful

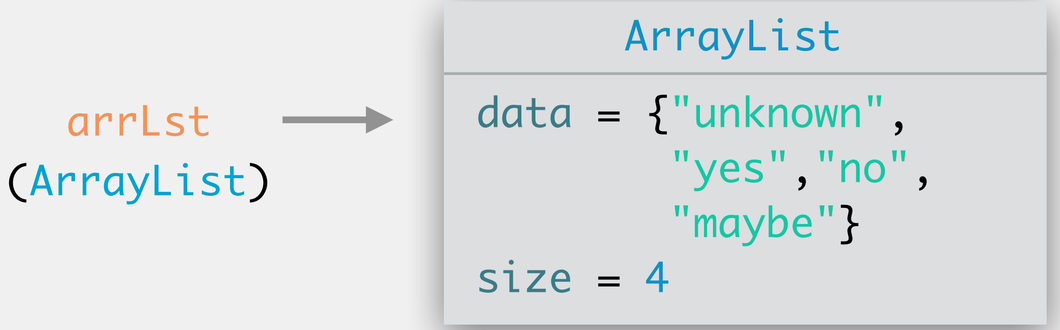
will always return true for ArrayList

throws: `IndexOutOfBoundsException` if index is negative or greater than size

behavior: adds value to index; shifts values down; increases size if required

```
ArrayList arrLst = new ArrayList(2);
arrLst.add("yes");
arrLst.add("no");
arrLst.add("maybe");
arrLst.add(0, "unknown");
>
```

memory



How Do We Increase The Array Length?

- Basic premise
  - create new array with size 1.5 * current size
  - copy values from old array to new array
  - point array variable to new array
- Implemented in a *private* method
  - prevents outsiders from modifying the size, losing data
  - will only be used internally by the class itself

ArrayList: ensureCapacity

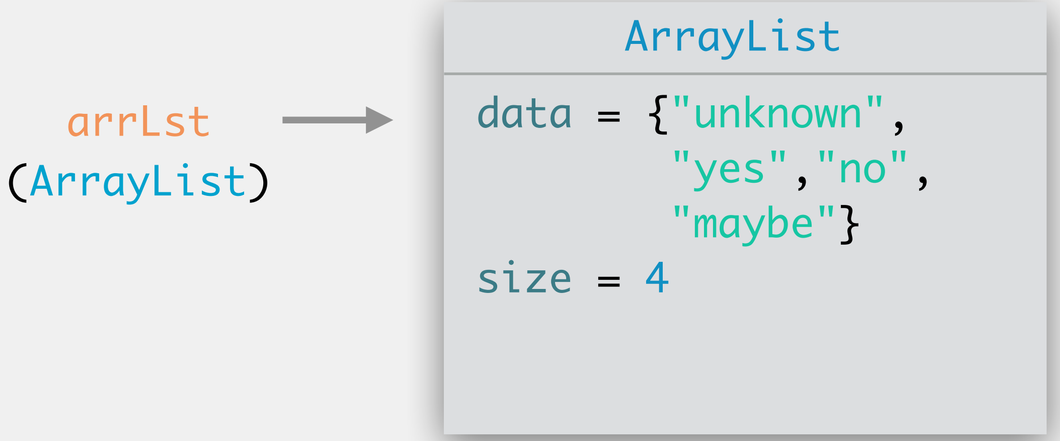
arguments: minimum capacity (`int`) for array's size

returns: nothing

behavior: increases array length to the minimum capacity; does nothing if minimum capacity is less than or equal to the size

```
 ArrayList arrLst = new ArrayList(2);
 /* values added */
>arrLst.ensureCapacity(9);
```

memory



ArrayList: ensureCapacity

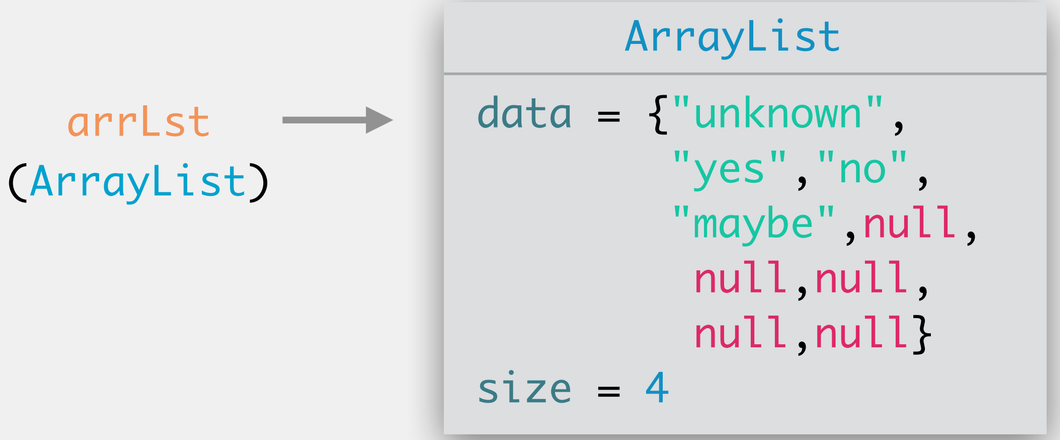
arguments: minimum capacity (`int`) for array's size

returns: nothing

behavior: increases array length to the minimum capacity; does nothing if minimum capacity is less than or equal to the size

```
ArrayList arrLst = new ArrayList(2);
/* values added */
arrLst.ensureCapacity(9);
>
```

memory



Consider an `ArrayList` with an array of length 5. You want to add 100 values to it. If you do not call `ensureCapacity`, how many times will the array need to be increased? (increase is * 1.5 + 1)

7 times (length: 107)

Consider an `ArrayList` with an array of length 5. You want to add 1,000,000 values to it. If you do not call `ensureCapacity`, how many times will the array need to be increased?

30 times (length: 1,215,485)

If you **know** you will be adding lots of data, call `ensureCapacity` first!

prevents incremental creation/reallocation of arrays

ArrayList: trimToSize

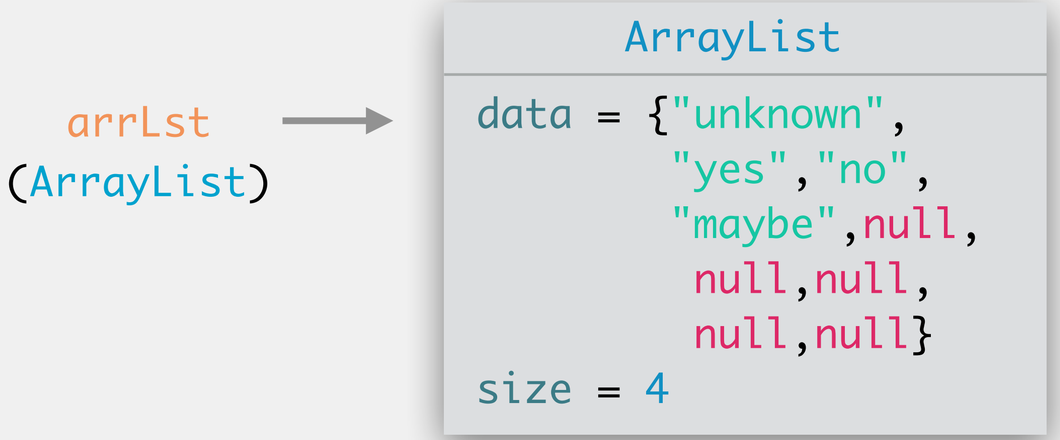
arguments: none

returns: nothing

behavior: decreases array length to match size; does nothing if array is full

```
 ArrayList arrLst = new ArrayList(2);
 /* values added */
>arrLst.trimToSize();
```

memory



ArrayList: trimToSize

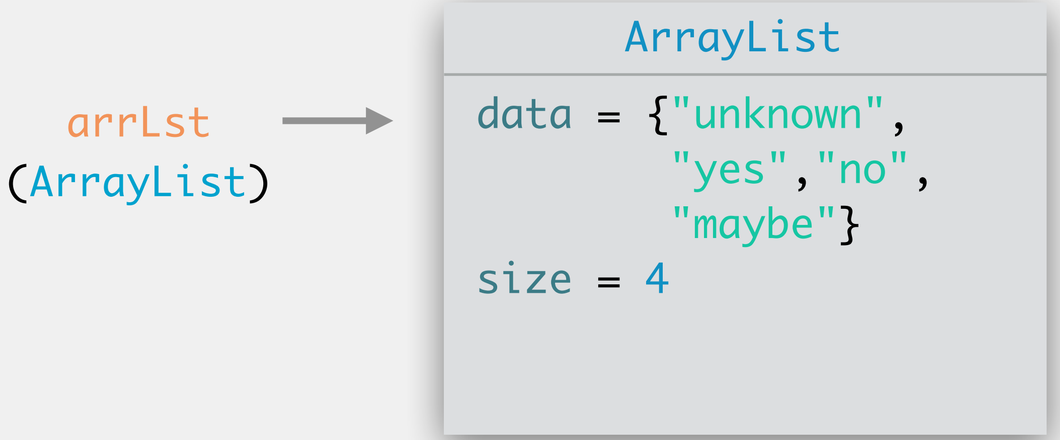
arguments: none

returns: nothing

behavior: decreases array length to match size; does nothing if array is full

```
ArrayList arrLst = new ArrayList(2);
/* values added */
arrLst.trimToSize();
>
```

memory



If you **know** you just deleted (relatively) lots of values, and know you won't be using the space soon, call `trimToSize()` afterwards!

frees up unused memory

ArrayList: remove Methods

arguments: index (`int`) position of element to remove

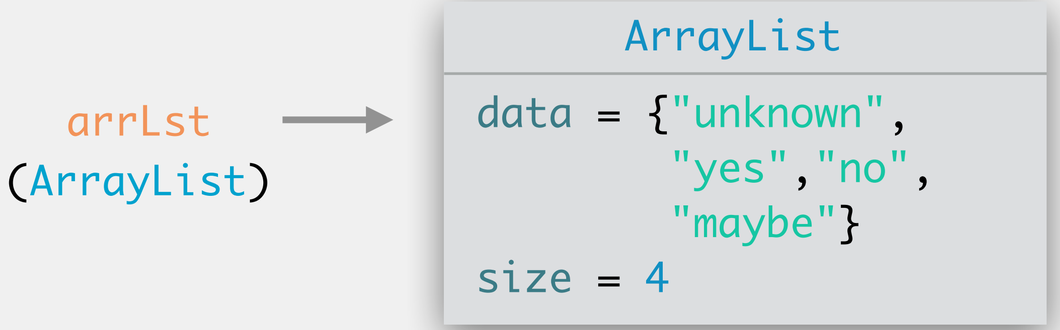
returns: `String` of the value removed

throws: `IndexOutOfBoundsException` if index is negative or greater than or equal to size

behavior: removes value at index; shifts values up

```
  ArrayList arrLst = new ArrayList(2);
  /* values added */
>arrLst.remove(1);
```

memory



ArrayList: remove Methods

arguments: index (`int`) position to remove

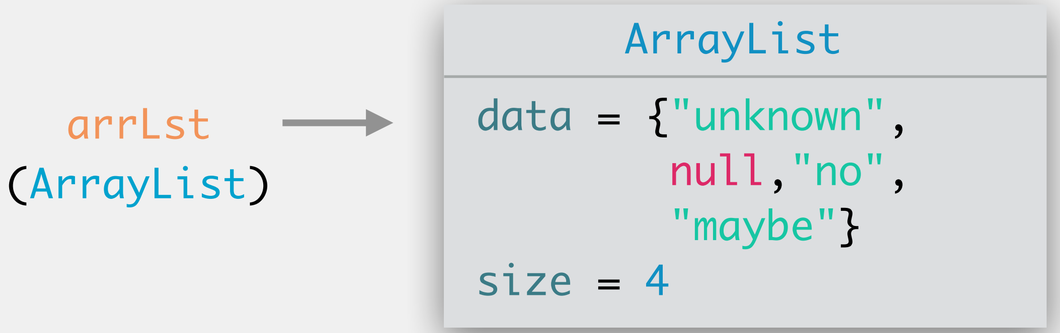
returns: `String` of the value removed

throws: `IndexOutOfBoundsException` if index is negative or greater than or equal to size

behavior: removes value at index; shifts values up

```
ArrayList arrLst = new ArrayList(2);
/* values added */
arrLst.remove(1);
>
```

memory



ArrayList: remove Methods

arguments: index (`int`) position to remove

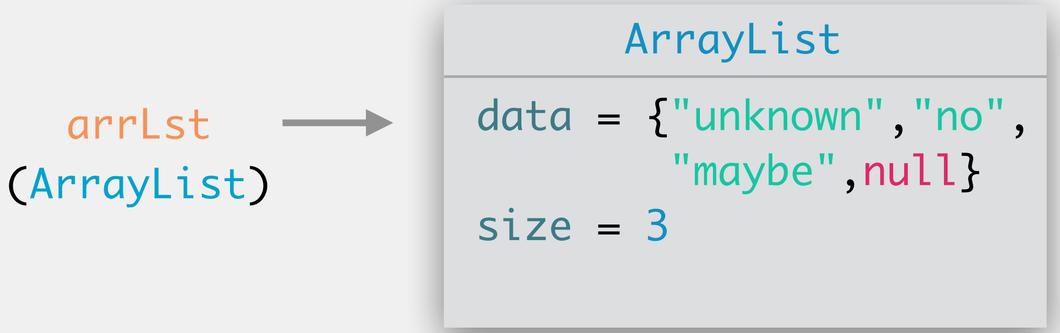
returns: `String` of the value removed

throws: `IndexOutOfBoundsException` if index is negative or greater than or equal to size

behavior: removes value at index; shifts values up

```
ArrayList arrLst = new ArrayList(2);
/* values added */
arrLst.remove(1);
>
```

memory



ArrayList: remove Methods

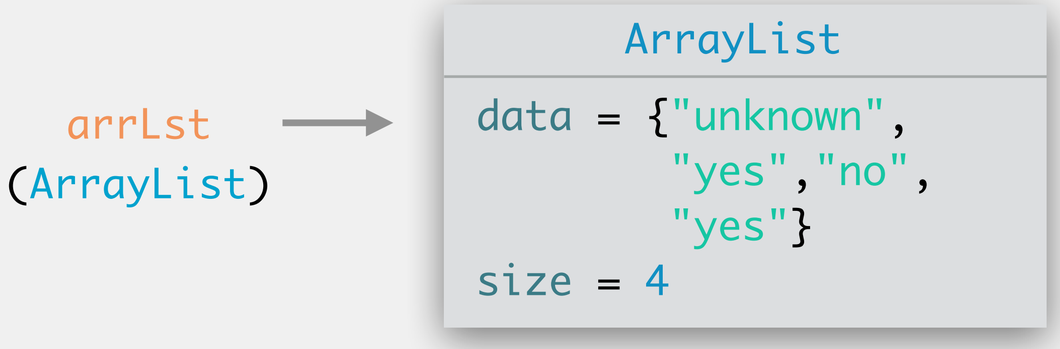
arguments: value (`String`) to remove; removes first occurrence

returns: `boolean` indicating whether or not the list changed

behavior: removes first occurrence of value; shifts values up

```
 ArrayList arrLst = new ArrayList(2);
 /* values added */
>arrLst.remove("yes");
```

memory



ArrayList: remove Methods

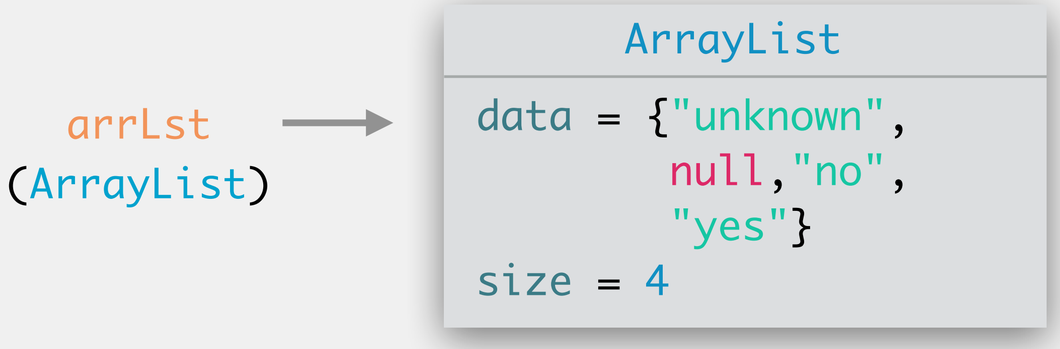
arguments: value (`String`) to remove; removes first occurrence

returns: `boolean` indicating whether or not the list changed

behavior: removes first occurrence of value; shifts values up

```
ArrayList arrLst = new ArrayList(2);
/* values added */
arrLst.remove("yes");
>
```

memory



ArrayList: remove Methods

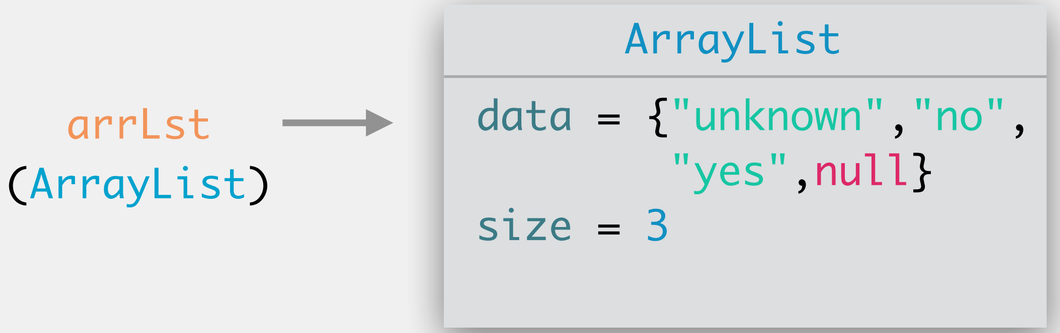
arguments: value (`String`) to remove; removes first occurrence

returns: `boolean` indicating whether or not the list changed

behavior: removes first occurrence of value; shifts values up

```
ArrayList arrLst = new ArrayList(2);
/* values added */
arrLst.remove("yes");
>
```

memory



Exercise: ArrayList Methods

Write code for the following `ArrayList` methods:

`public int size()`

returns the size of the array list

`public boolean isEmpty()`

returns `true` if the list if empty, `false` if not

`public void clear()`

resets every value in the array to `null`; `isEmpty()` should be `true` after executing this method

`public String get(int index)`

returns the value at the specified index; throws an `IndexOutOfBoundsException` if the index is outside the appropriate bounds

Exercise: ArrayList Methods

Write code for the following `ArrayList` methods:

`public String set(int index, String s)`

replaces the value at `index` with `s`; returns the string originally stored at `index`

`public boolean contains(String s)`

returns `true` if `s` is contained in the list, `false` if not

`public int indexOf(String s)`

returns the index of the first occurrence of `s`; returns -1 if the string does not exist in the list

`public int lastIndexOf(String s)`

returns the index of the last occurrence of `s`; returns -1 if the string does not exist in the list