L4: Interfaces and Implementations, ArrayList

Alex Steiger

CompSci 201: Spring 2024

1/24/24

Logistics, Coming up

- Today, Wednesday, 1/24
  - APT 1 due
- This Friday, 1/26
  - Discussion 2: APTs, Sets, Strings, Git
- Next Monday 1/29
  - Project 0: Person201 due (warmup project)
- Next Wednesday 1/31
  - APT 2 due

Daytime Office Hours

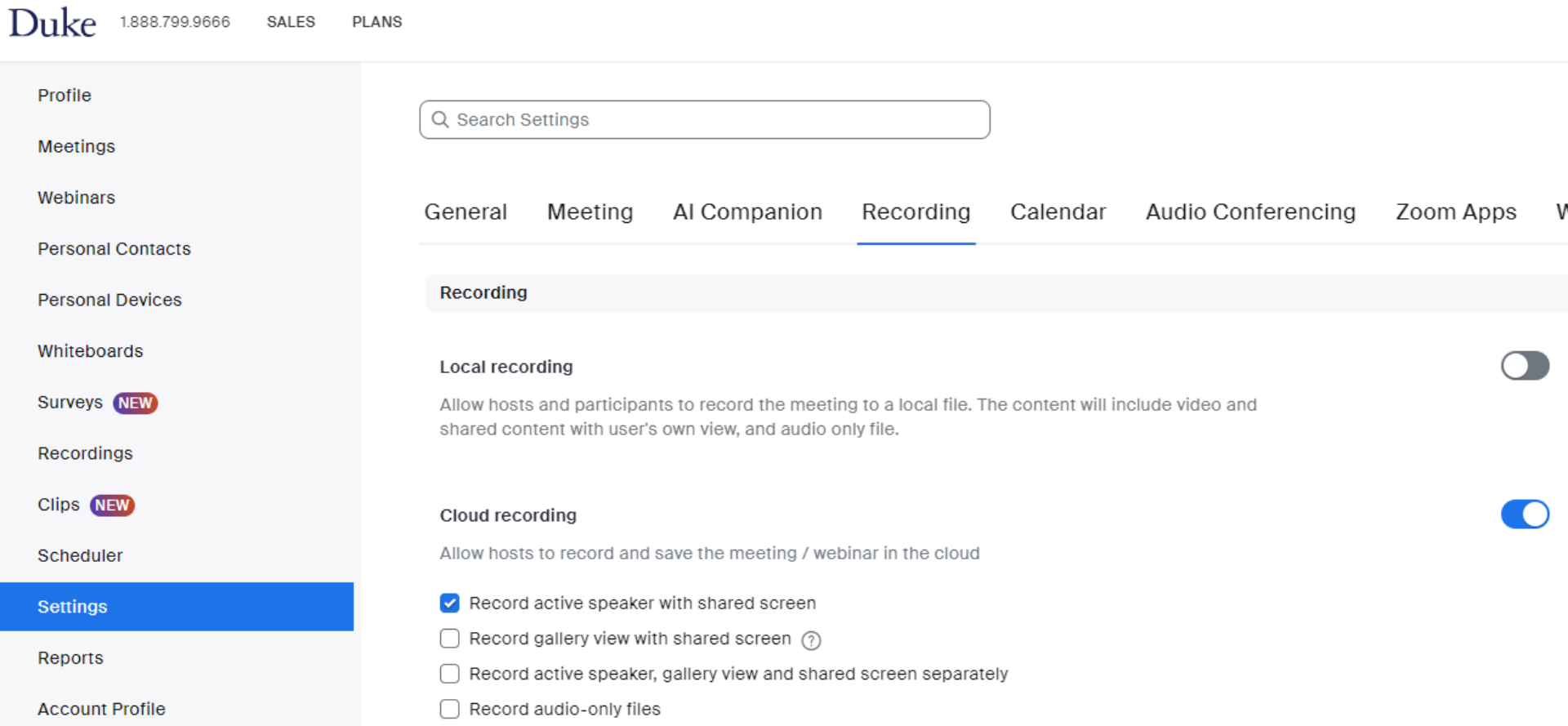
- Mondays 10am-12pm with Mark
  - LSRC D309
- Tuesdays 1-3pm with Eamon
  - LSRC D309
- Thursdays 10-11am, 3-4pm with Alex
  - LSRC D344 and Zoom

Reminder: Course Resources

- Getting Help
- zyBook →
- Java4Python

P0: duke.zoom.us

P0: Enabling Cloud Recordings



P0: Submitting to Gradescope

COMPSCI 201 | Spring 2024

Course ID: 693578

No Published Grades

Description

Edit your course description on the Course Settings page.

Things To Do

Review and publish grades for P0: Person 201 (Code) now that you're all done grading.

Active Assignments	Released	Due (EST)	Submissions	% Graded	Published
P0: Person 201 (Code)	JAN 22, 2024 9:00 AM	JAN 29, 2024 11:59 PM Late Due Date: FEB 5, 2024 11:59 PM	5	100%	
P0: Person 201 (Analysis)	JAN 22, 2024 9:00 AM	JAN 29, 2024 11:59 PM Late Due Date: FEB 5, 2024 11:59 PM	2	0%	
Optional Gradescope WOTO	JAN 22, 2024 1:00 PM	JAN 26, 2024 11:59 PM Late Due Date: JAN 28, 2024 11:59 PM	9	0%	

OOP (Object-Oriented Programming) Wrapup

Public vs. Private

- **Public** – Can be accessed by code *outside* of the class.
- **Private** – Can *only* be accessed by code *inside* of the class.

Record.java > Record

```java
public class Record {
    public String displayName;
    private int uniqueID;

    public Record(String name, int id) {
        displayName = name;
        uniqueID = id;
    }

```

PublicPrivate.java > ...

```java
public class PublicPrivate {
    Run | Debug
    public static void main (String[] args) {
        Record rec = new Record("Fain", 12345);
        System.out.println(rec.displayName);
        System.out.println(rec.uniqueID);
    }
}
```

Can access this **public** instance variable

Cannot access this **private** instance variable

What about neither?

- **Public** – Can be accessed by code *outside* of the class.
- **Private** – Can *only* be accessed by code *inside* of the class.

Record.java > Record

```
1   public class Record {
2       public String displayName;
3       private int uniqueID;
4
5       public Record(String name, int id) {
6           displayName = name;
7           uniqueID = id;
8       }
9
```

Record.java > ...

```
1   class Record {
2       String displayName;
3       int uniqueID;
4
5       Record(String name, int id) {
6           displayName = name;
7           uniqueID = id;
8       }
9   }
```

- **No modifier** – Can be accessed by code *in the same package*.
- Almost the same as Public for 201 code
- Use Public/Private!

`.contains` for List

contains

```
boolean contains(Object o)
```

Returns `true` if this list contains the specified element. More formally, returns `true` if and only if this list contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.

Specified by:

`contains` in interface `Collection<E>`

Parameters:

`o` - element whose presence in this list is to be tested

Returns:

`true` if this list contains the specified element

What is printed?

```java
public class Blob {
    public String color;
    public String shape;
    public Blob(String color, String shape) {
        this.color = color;
        this.shape = shape;
    }

    @Override
    public boolean equals(Object obj) {
        Blob other = (Blob) obj;
        if (other.shape.equals(this.shape)) {
            return true;
        }
        return false;
    }
}
```

Blobs are equal if they have the same shape (and any colors)

```java
import java.util.ArrayList;

public class BlobDriver {
    Run | Debug
    public static void main(String[] args) {
        ArrayList<Blob> myBlobs = new ArrayList<>();
        String[] colors = {"red", "white", "blue", "green"};
        String[] shapes = {"round", "oblong", "square"};
        for (String color : colors) {
            for (String shape : shapes) {
                Blob newBlob = new Blob(color, shape);
                if (! myBlobs.contains(newBlob)) {
                    myBlobs.add(newBlob);
                }
            }
        }
        System.out.println(myBlobs.size());
    }
}
```

Try adding a Blob of every color-shape combination

3

Why use Classes/objects?

- Because you must in Java
- Formal specification for complex data structures
- Convenience and ease of correct programming
- Composition, Interfaces, & Implementations, Extending & Inheritance – More later!

It’s ok to not be fully “convinced” yet. But OOP has proven itself to be a powerful paradigm for designing complex, scalable software.

Interfaces and Implementations

Abstract Data Type (ADT)

- ADT specifies ***what*** a data structure does (functionality) but not ***how*** it does it (implementation).

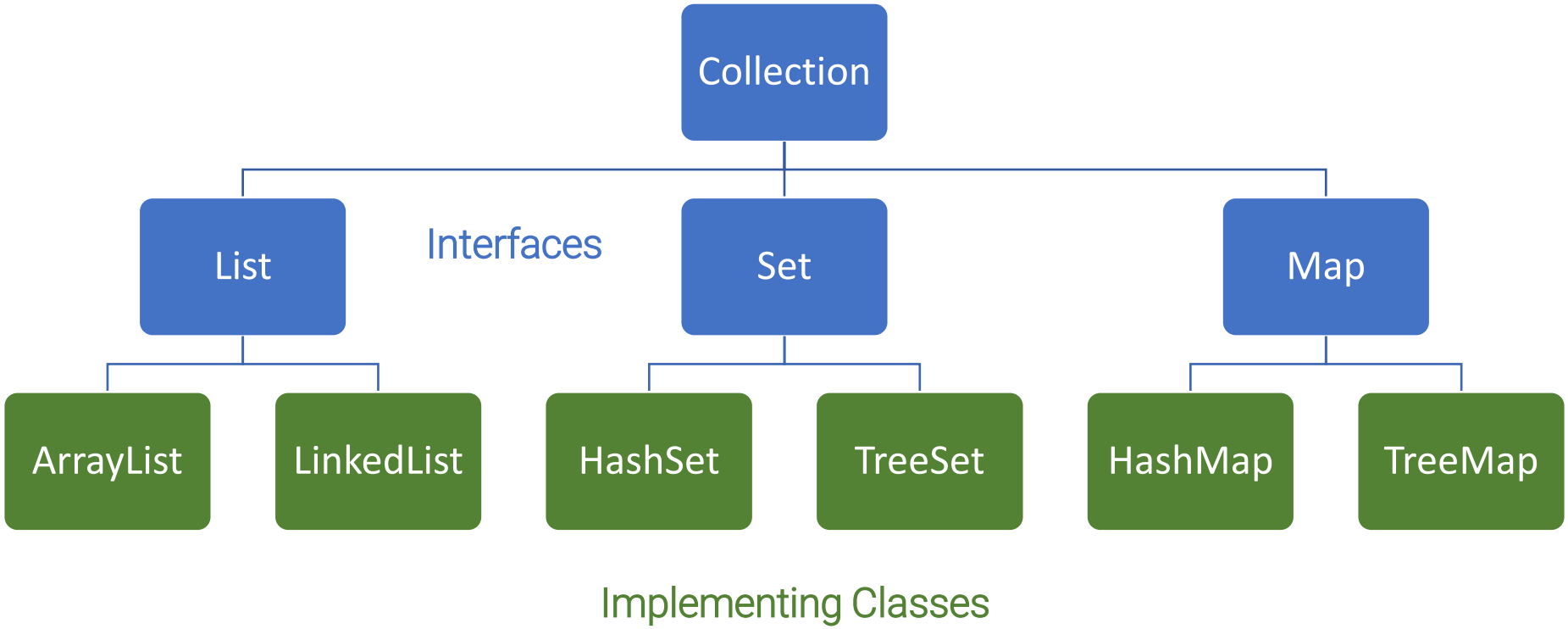
- API (Application Program Interface) perspective: What methods can I call on these objects, what inputs do they take, what outputs do they return?

- For example, an abstract List should…
  - Keep values in an order
  - Be able to add new values, grow
  - Be able to get the first value, or the last, etc.
  - Be able to get the size of the list

Java Interface

- One primary way Java formalizes ADTs is with **interfaces,** which "*specify a set of abstract methods that an implementing class must override and define*." – ZyBook
- 3 most important ADTs we study are all interfaces in Java!
  - **List**: An ordered sequence of values
  - **Set**: An unordered collection of *unique* elements
  - **Map**: A collection that associates keys and values

The Java Collection Hierarchy



What is a collection?

```
public interface Collection<E>
extends Iterable<E>
```

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

- Java API data structures storing groups of objects likely based on the **`Collection`** interface.
- Lists, Sets, Maps, and more
- Useful static methods (such as sorting) in java.util.Collections (like Java.util.Arrays), see API documentation

Interface vs. Implementation

Cannot instantiate an Interface object itself, but rather an ***implementation*** of that Interface

```
1   public class InterfaceExample {
        Run | Debug
2       public static void main(String[
3           List<String> strList = new List<>();
4       }
```

List cannot be resolved to a type Java(16777218)
View Problem Quick Fix... (⌘.)

What is an ***implementation***?

- Must override and implement *all* methods.
- Can have any instance variables.

DIYList.java > DIYList

```
1   import java.util.List;
2   public class DIYList implements List {
3
4   }
5
```

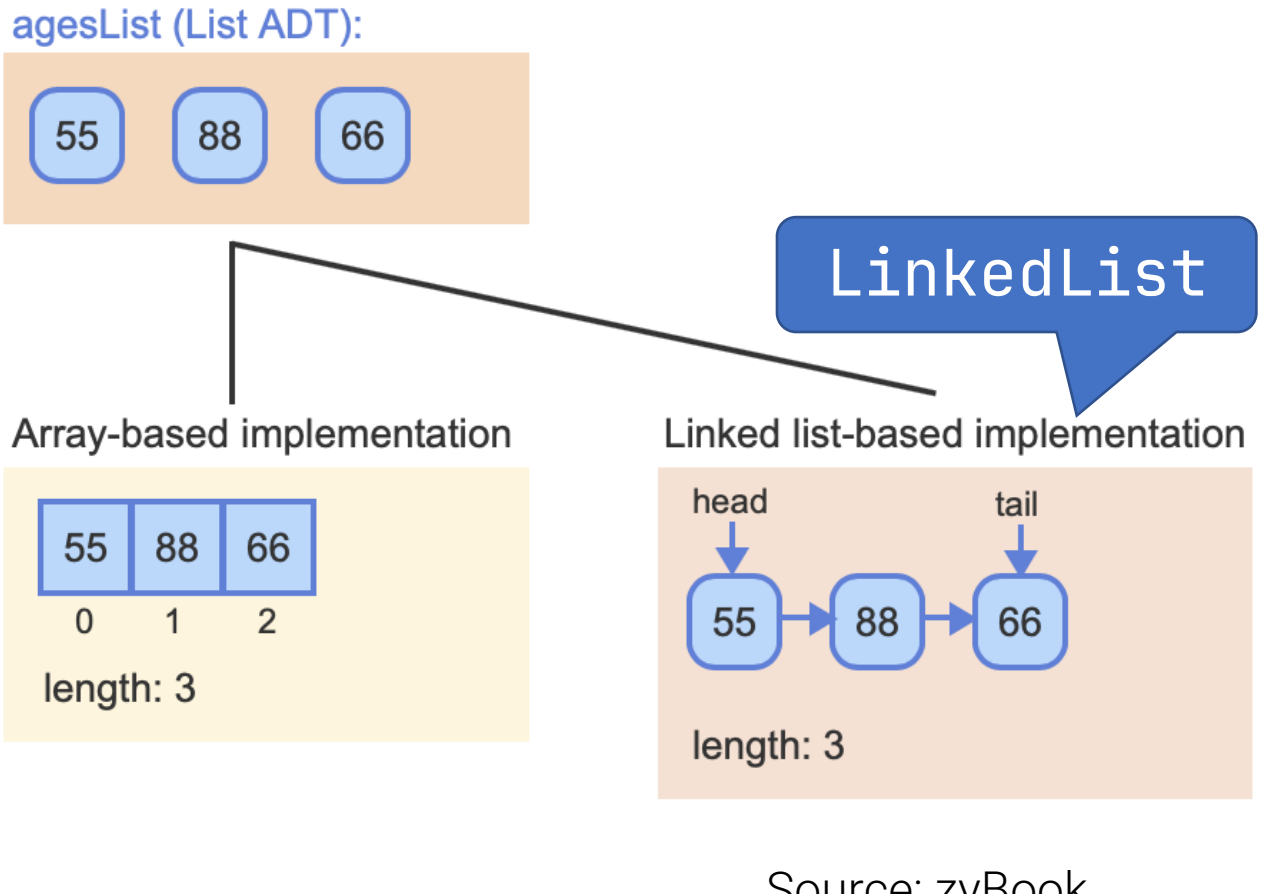
Add unimplemented methods

```
6   public class DIYList implements List {
7
8       @Override
9       public int size() {
10          // TODO Auto-generated method stub
11          return 0;
12      }
```

Multiple Implementations of the Same Interface

```
agesList = new List
Append(agesList, 55)
Append(agesList, 88)
Append(agesList, 66)
Print(agesList)
```

Print result: 55, 88, 66



Source: zyBook

Implementations must have all methods of the Interface

Doesn't matter for correctness whether the argument Lists are ArrayList or LinkedList, because both implement **`.contains()`**.

```
17    public static List<String> inBothLists(List<String> aList,
18                                           List<String> bList) {
19        List<String> retList = new ArrayList<>();
20        for (String s : aList) {
21            if (bList.contains(s)) {
22                retList.add(s);
23            }
24        }
25        return retList;
26    }
```

Method doesn't even "know" how **`aList`** and **`bList`** are implemented.

Since **`retList`** is an **`ArrayList`** which implements **`List`**, it is a valid return.

ArrayList Implementation

Algorithmic tradeoffs depend on the implementation

Often, we are interested in how the **efficiency** of operations on data structures depends on **scale**. For an **ArrayList** with N values how efficient is…

- **`get()`**. Direct lookup in an Array. "Constant time" – does not depend on size of the list.
- **`contains()`**. Loops through Array calling **`.equals()`** at each step. Takes longer as list grows.
- **`size()`**. Returns value of an instance variable tracking size, does not depend on size of the list.
- **`add()`**. Depends.

How does `ArrayList` `add` work?

Implements **`List`** (can grow) with **`Array`** (cannot grow). How?

Keep an Array with extra space at the end. Two cases when adding to end of ArrayList:

1. Space left – add to first open position.
2. No space left – Create a new (larger) array, copy everything, then add to first open position.

Array representing List

DIY (do it yourself) ArrayList

Live Coding 

How efficient is `ArrayList add`?

For an **`ArrayList`** with N values, 2 cases:

1. Space left – One Array assignment statement, *constant time*, does not depend on list size.
2. No space left – Copy entire list! Takes N array assignments!

How often are we in the second slow case? Depends on *how much we increase the Array size by in case 2.*

ArrayList Growth

Starting with Array length 1, if you keep creating a new Array that…

Is twice as large (geometric growth)

- Must copy at sizes:
  - 1, 2, 4, 8, 16, 32, …
- Total values copied to add N values:
  - 1+2+4+8+16+…N

Has 100 more positions (arithmetic growth)

- Must copy at sizes:
  - 1, 101, 201, 301, …
- Total values copied to add N values:
  - 1+101+201+301+…+N

Algebra to our rescue!

ArrayList Growth and Algebra

Geometric growth

$$1 + 2 + 4 + \cdots + N$$

$$= \sum_{i=0}^{\approx \log_2 N} 2^i$$

$$\approx 2N$$

Geometric series formula:

$$\sum_{i=0}^{n} a\, r^i = a\left(\frac{1 - r^{n+1}}{1 - r}\right)$$

Arithmetic growth

$$1 + 101 + 201 + \cdots + N$$

$$= \sum_{i=0}^{\approx N/100} 1 + 100i$$

$$\approx \frac{N^2}{200}$$

Arithmetic series formula:

$$\sum_{i=1}^{n} a_i = \left(\frac{n}{2}\right)(a_1 + a_n)$$

Math and Expectations in 201

- ***Do not*** expect you to formally derive closed form expressions / give proofs.
- ***Do*** expect you to recognize:
  - Geometric growth: $1 + 2 + 4 + \cdots + N$ is *linear*, $\approx 2N$.
  - Arithmetic growth: $1 + 101 + 201 + \cdots + N$ is *quadratic*, $\approx \frac{N^2}{200}$.
- Patterns like these show up again and again!

```
3       int n = 100;
4       int numIterations = 0;
5       for (int i=0; i<n; i++) {
6           for (int j=0; j<i; j++) {
7               numIterations += 1;
8           }
9       }
```

```
numIterations: 4950
n*(n-1)/2): 4950
```

Experiment to verify hypothesis

Live Coding 

`ArrayList` `add` (to end) is (amortized) efficient

According to the Java 17 API documentation: "The add operation runs in *amortized constant time…*" – What does that mean?

- With geometric growth (e.g., double size of Array whenever out of space): Need $\approx 2N$ copies to `add` N elements to `ArrayList`.

- The *average* number of copies per add is thus $\frac{2N}{N} = 2$, a constant that does not depend on N.

`ArrayList` add to the front is not efficient

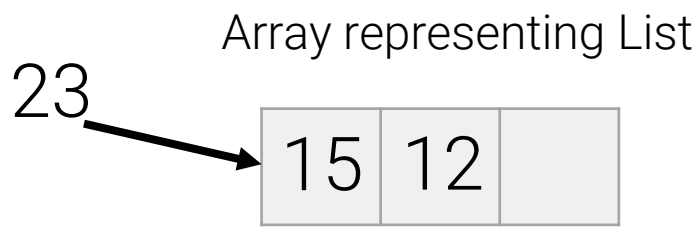
add

```
public void add(int index,
                E element)
```

[Java 17 API documentation of add]

Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

Always requires shifting the entire **`Array`**, even if there is space available.



`ArrayList` `contains` revisited

`contains` loops through the `Array` calling `.equals()` at each step. May check every element!

`list.contains(33)`