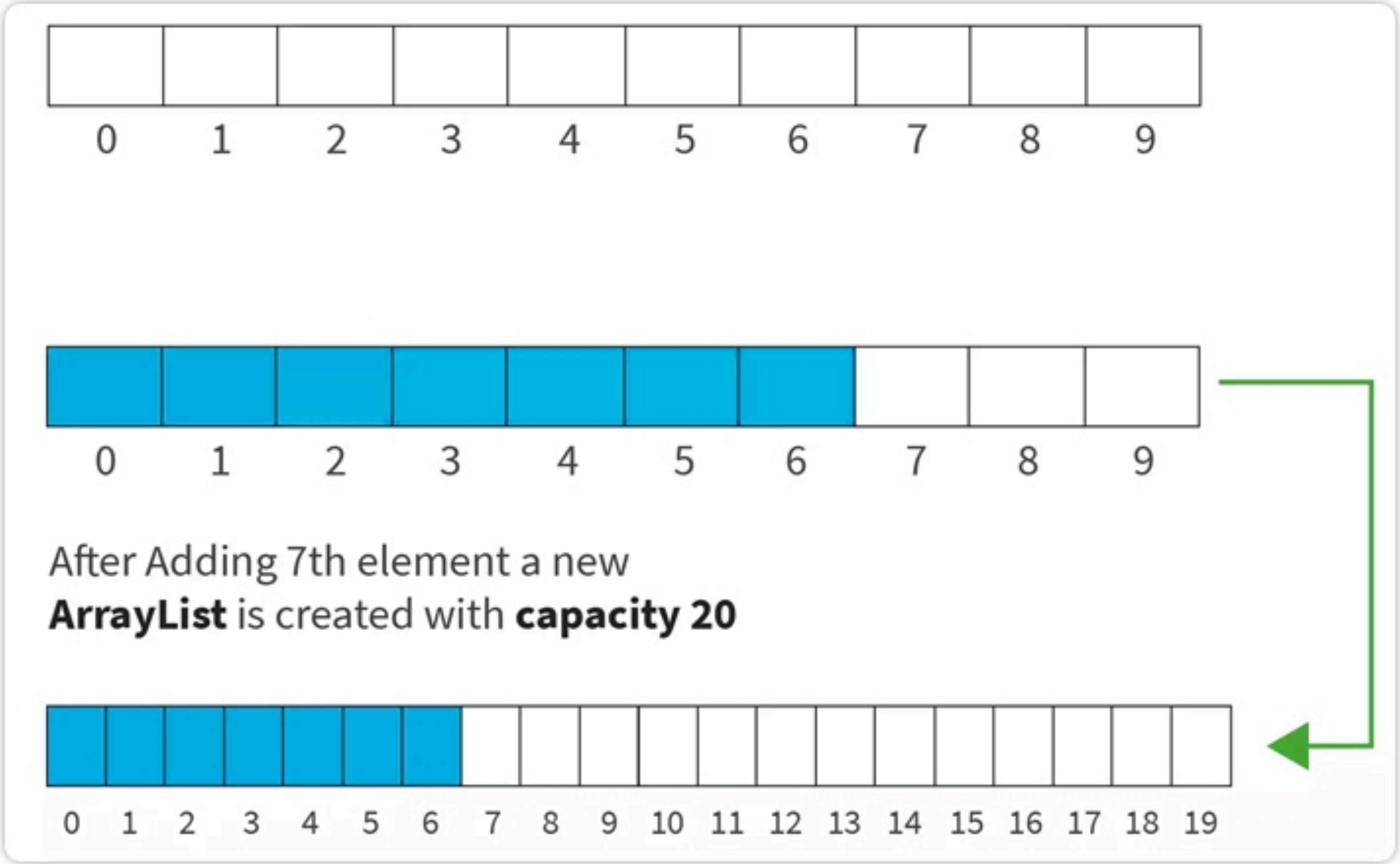
CS62 Class 4: Interfaces, Generics, ArrayLists

Basic Data Structures



from https://www.janbasktraining.com/blog/java-print-arraylist/

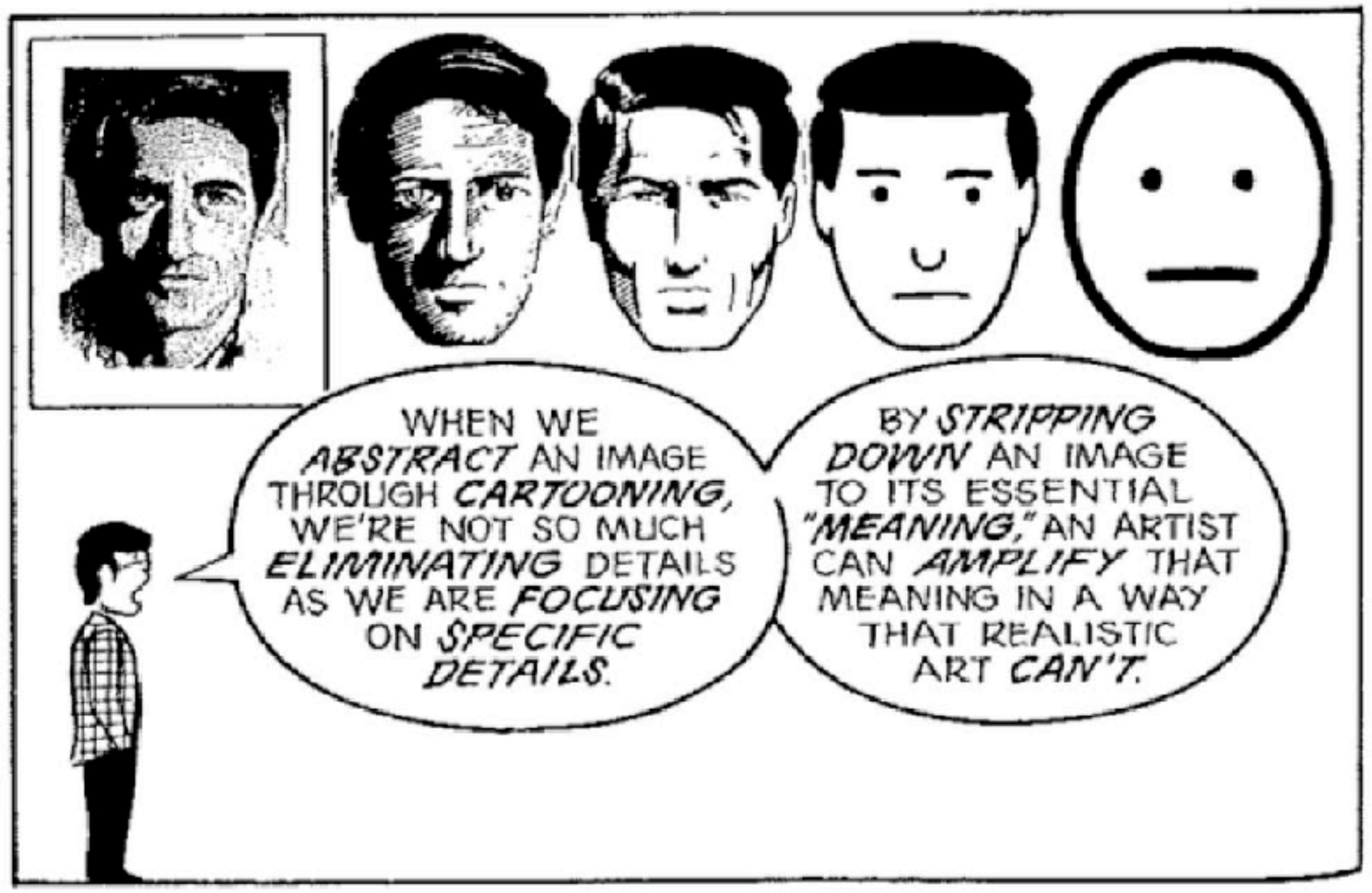
Lecture 4 agenda

- Data structures in general
  - What are abstract data types?
  - Interfaces
  - Why do we care + history
  - Generics
- ArrayLists behavior
- ArrayLists implementation

Data structures in general

Abstract data type

- Sometimes, you'll hear specific data structures be referred to as "abstract data types". This is a *conceptual model* where users **know the behavior of a data structure, but not exactly how it's implemented.**
  - How it's implemented means where in memory things are, what algorithms are used…
- For example, the idea of a "list" is an abstract data type. We know we can add, remove, resize a list, but how exactly that happens is not important.
  - ArrayLists, Singly Linked Lists, Doubly Linked Lists are **not** abstract: we have to implement them.
- Basically, an abstract data type is the same thing as an interface or API


WHEN WE ABSTRACT AN IMAGE THROUGH CARTOONING, WE'RE NOT SO MUCH ELIMINATING DETAILS AS WE ARE FOCUSING ON SPECIFIC DETAILS.
BY STRIPPING DOWN AN IMAGE TO ITS ESSENTIAL "MEANING," AN ARTIST CAN AMPLIFY THAT MEANING IN A WAY THAT REALISTIC ART CAN'T.

Interfaces

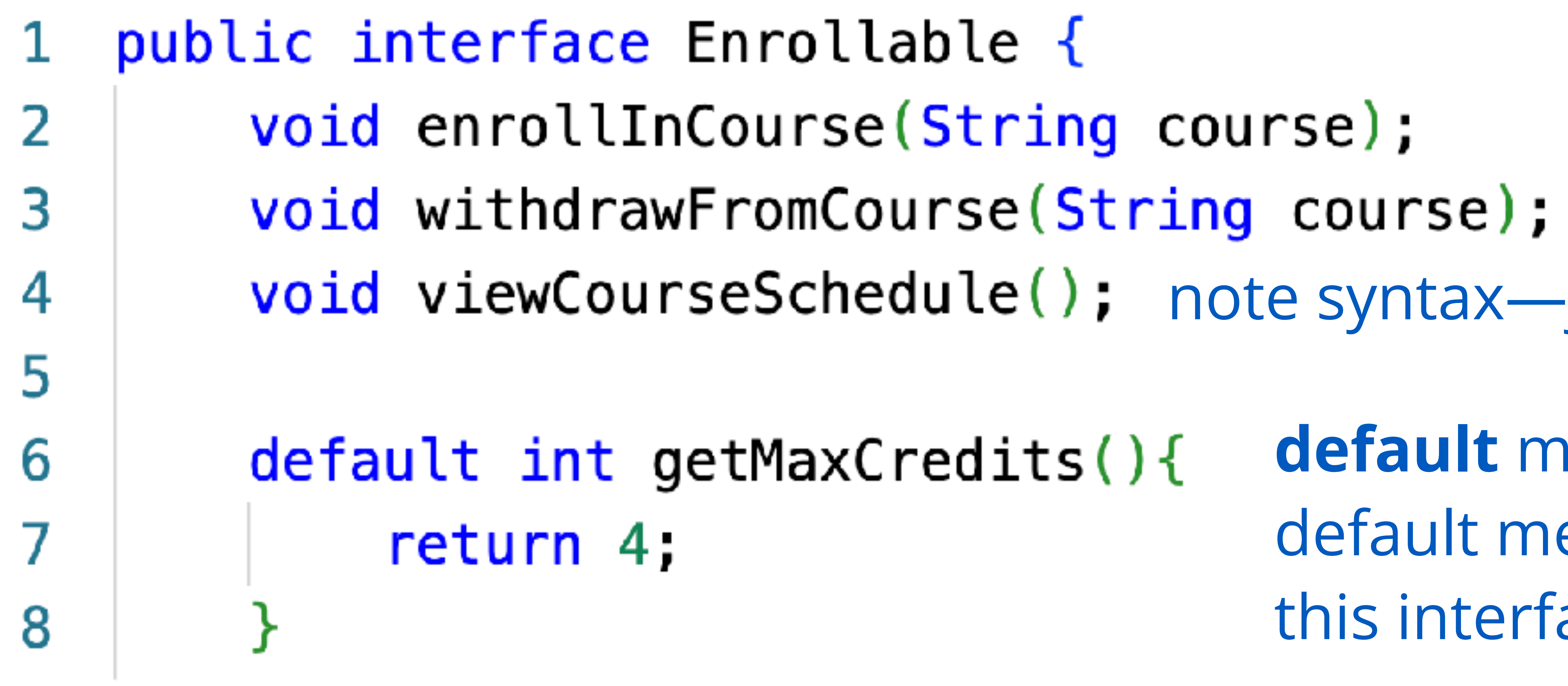
Interfaces: managing abstraction

- An interface is a form of **abstraction** that is a **contract** of what a class must do. As an abstraction, it does not say *how* a class should do it.
- In Java, an interface is a reference type (like a class), that contains **abstract methods** and **default methods**.
- A class that implements an interface is obliged to implement its abstract methods.
- Interfaces cannot be instantiated (no new keyword). They can only be *implemented* by classes or *extended* by other interfaces.

Example

All students (Pomona, Scripps, etc) should be able to enroll in classes.

Pomona students have a different registration system than Scripps ones, so an interface helps us specify *what* should happen, not *how* it happens—not the implementation detail.



```java
public interface Enrollable {
    void enrollInCourse(String course);
    void withdrawFromCourse(String course);
    void viewCourseSchedule();

    default int getMaxCredits(){
        return 4;
    }
```

Methods in an interface are considered “abstract”. An interface **only** includes method **signatures**. Classes that implement the interface fill out the signatures.

note syntax—just ; no {}

default method: need “default” keyword in beginning. A default method has a body. Everything that implements this interface can use this method.

Different from default (vs public/private) data access modifier!

All methods are implicitly public in an interface - no need for “public” data access modifier

Example

Use the “implements” keyword to implement an interface from a class.

```
class PomonaStudent implements Enrollable{

…
   public void enrollInCourse(String course) {
        // implementation
    }

   public void withdrawFromCourse(String course) {
        // implementation
    }

   public void viewCourseSchedule() {
        // implementation
    }
```

```
class ScrippsStudent implements Enrollable{

…
   public void enrollInCourse(String course) {
        // implementation
    }

   public void withdrawFromCourse(String course) {
        // implementation
    }

   public void viewCourseSchedule() {
        // implementation
    }
```

Example

```
class FourthYearPomonaStudent extends PomonaStudent{

…
   public int getMaxCredits(){
        return 6;
    }
}
```

can override default methods of interfaces

Q: Why don't we need "implements Enrollable" for FourthYearPomonaStudent?

A: FourthYearPomonaStudent extends PomonaStudent, which already implements Enrollable

Interface rules

- A class can **implement** multiple interfaces.
  - `classA implements Interface1, Interface2{…}`
  - `PomonaStudent implements Enrollable, Extracurriculars`
- An interface can **extend** multiple interfaces.
  - `public interface GroupedInterface extends Interface1,Interface2{…}`
  - `public interface Extracurriculars extends AcademicClubs, ClubSports`

Remember: a class can only extend one class

Worksheet time!

- Do problem 1 a & b on your worksheet.

```
interface Vehicle {
    public void revEngine();
}

interface Noisemaker {
    public void makeNoise();
}

public class CatBus ______________________ ________________, ________________ {
    @Override
    ___________ __________ _______________ { /* CatBus revs engine, code not shown */ }

    @Override
    ___________ __________ _______________ { /* CatBus makes noise, code not shown */ }

    /** Allows CatBus to make noise at other CatBuses. */
    public void conversation(CatBus target) {
        makeNoise();
        target.makeNoise();
    }
}
```

Worksheet answers

- Do problem 1 a & b on your worksheet.

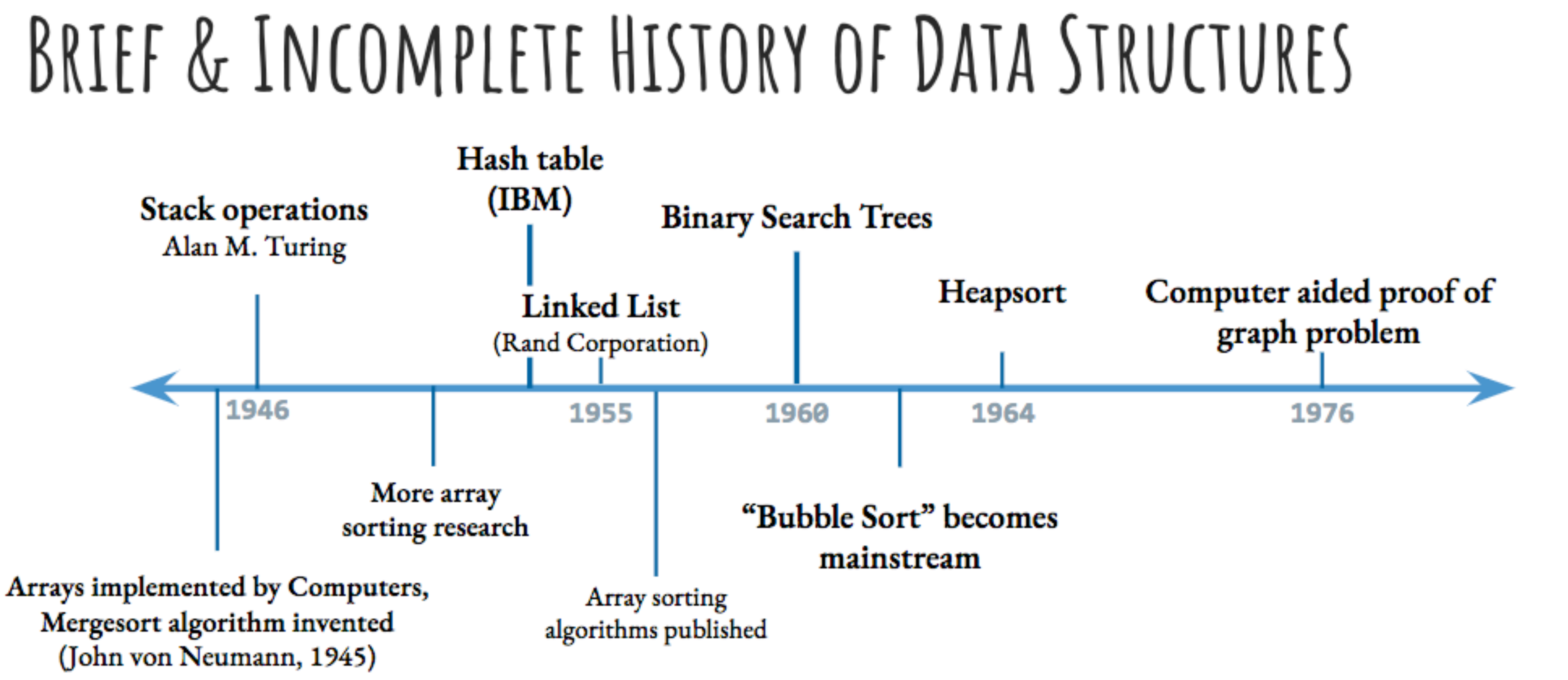
```java
interface Vehicle {
    public void revEngine();
}

interface Noisemaker {
    public void makeNoise();
}

public class CatBus implements Vehicle, Noisemaker {
    @Override
    public void revEngine() {
        // CatBus revs its engine, implementation not shown
    }

    @Override
    public void makeNoise() {
        // CatBus makes noise, implementation not shown
    }

    /** Allows CatBus to make noise at other CatBuses. */
    public void conversation(CatBus target) {
        makeNoise();
        target.makeNoise();
    }
}
```

We can change the method signature so that the type of the parameter **`target`** is **`Noisemaker`** (both **`CatBus`** and **`Goose`** implement **`Noisemaker`**:

```java
/** Allows CatBus to make noise at other both CatBuses and Gooses. */
public void conversation(Noisemaker target) {
    makeNoise();
    target.makeNoise();
}
```

Data structures history

Data structures history

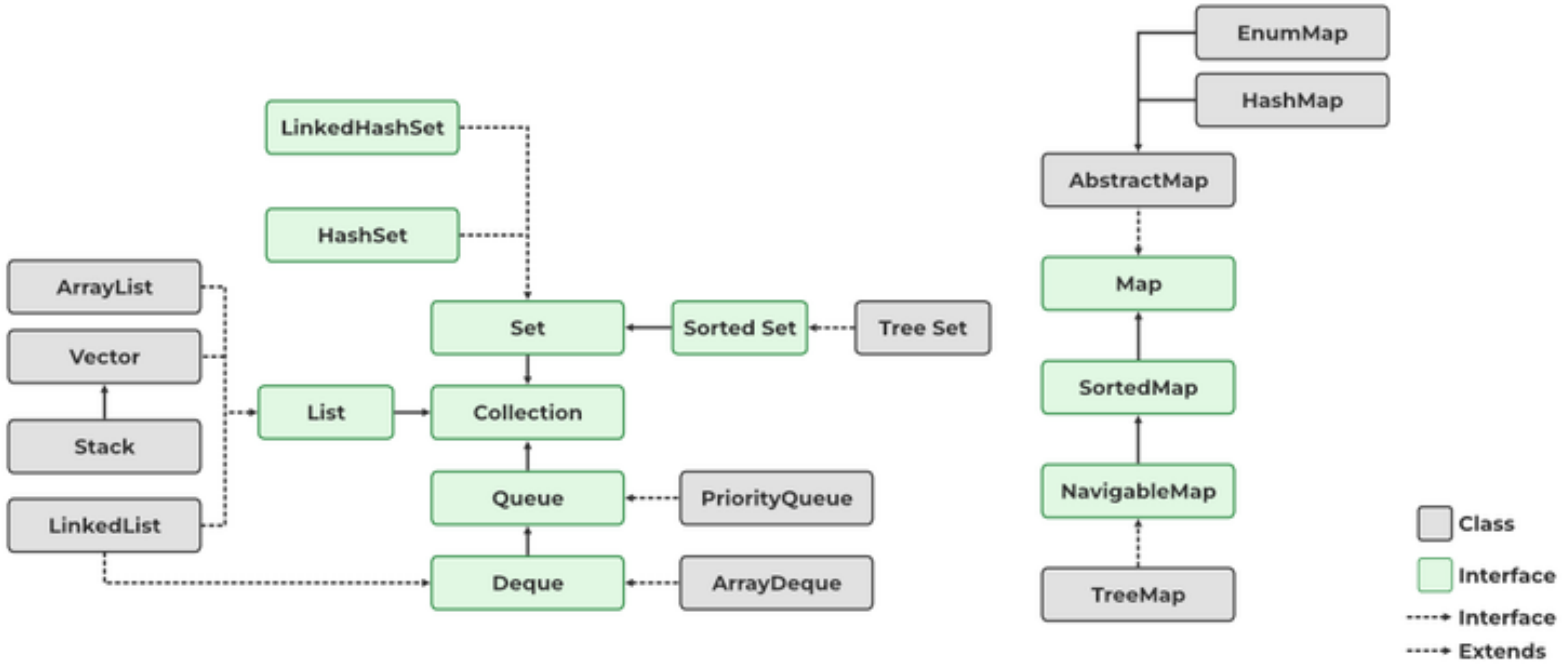

Brief & Incomplete History of Data Structures
Stack operations
Alan M. Turing
Hash table
(IBM)
Binary Search Trees
Linked List
(Rand Corporation)
Heapsort
Computer aided proof of graph problem
1946
1955
1960
1964
1976
More array sorting research
Arrays implemented by Computers, Mergesort algorithm invented (John von Neumann, 1945)
Array sorting algorithms published
"Bubble Sort" becomes mainstream

"Tree" was coined in 1857 by British mathematician Arthur Cayley (link)
The origin of graph theory was 1736 on the Bridges of Königsberg (link)

https://macbookandheels.com/algorithm/2018/10/31/teachingds/

- Lots of concepts in CS came from math (trees, graphs) and have been around for a while
- The invention of many data structures coinciding with World War II is no coincidence
- ArrayLists are not one of these: they are a relatively modern implementation (1998), even though arrays have been around since the 1940s

The Java Collections Framework



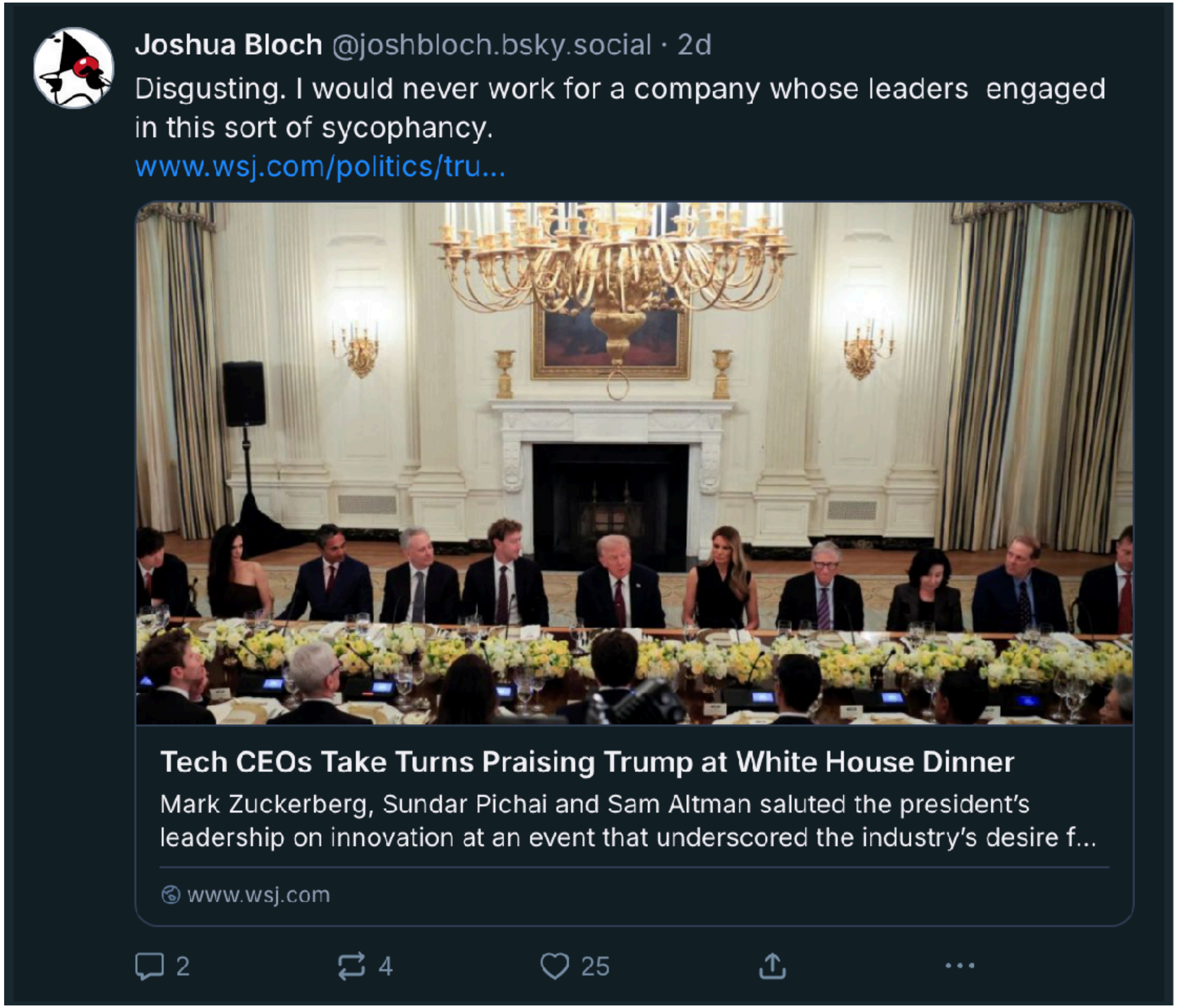
- Built in data structure classes that you may import
- We used it in the Silver Dollar lab. Today, we'll write our own ArrayList implementation.

```java
import java.util.ArrayList;
```

https://www.geeksforgeeks.org/collections-in-java-2/

Joshua Bloch: main Java Collections Framework architect

- The person who made ArrayLists isn't some long dead historic figure; he was an engineer at Sun Microsystems (the company that made Java) but now teaches at CMU
- He's currently pretty politically active on BlueSky



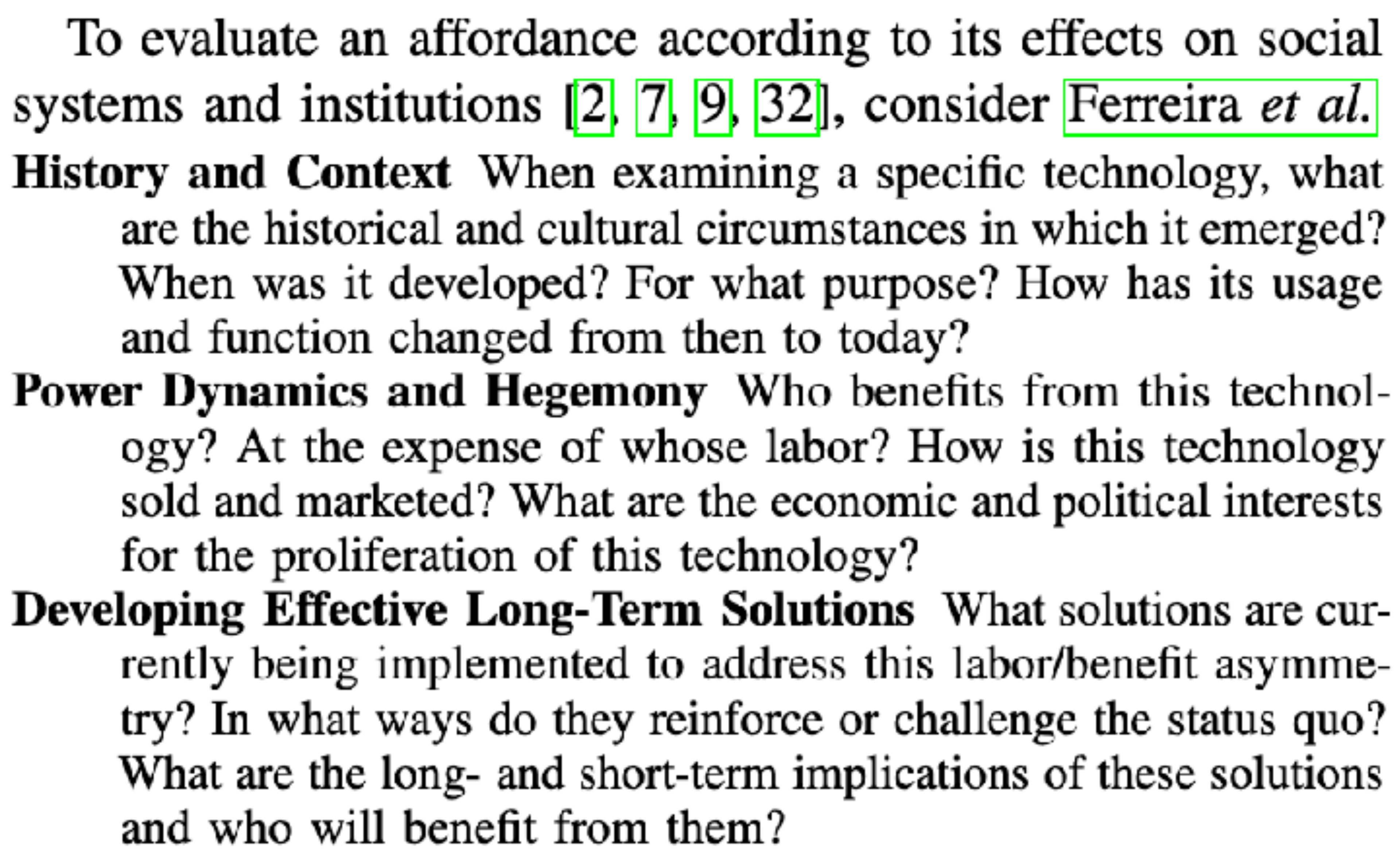
Joshua Bloch @joshbloch.bsky.social · 2d
Disgusting. I would never work for a company whose leaders engaged in this sort of sycophancy.
www.wsj.com/politics/tru...

Tech CEOs Take Turns Praising Trump at White House Dinner
Mark Zuckerberg, Sundar Pichai and Sam Altman saluted the president's leadership on innovation at an event that underscored the industry's desire f...
www.wsj.com

2 4 25

My (liberal arts-y) wish for you: know history

- It's one thing to know how data structures work (most data structures courses)
- But it's another to know when to appropriately and ethically how to use them and understand their history
- STS (science technology society) scholar Bruno Latour calls this "opening the black box of science" - a real person had to create everything we use in this class to solve real problems, and understanding this history gives you a deeper insight into how technology is used by society (and how it can encode bias, etc.)
- Hopefully, this knowledge will equip you for your final project



To evaluate an affordance according to its effects on social systems and institutions [2, 7, 9, 32], consider Ferreira *et al.*

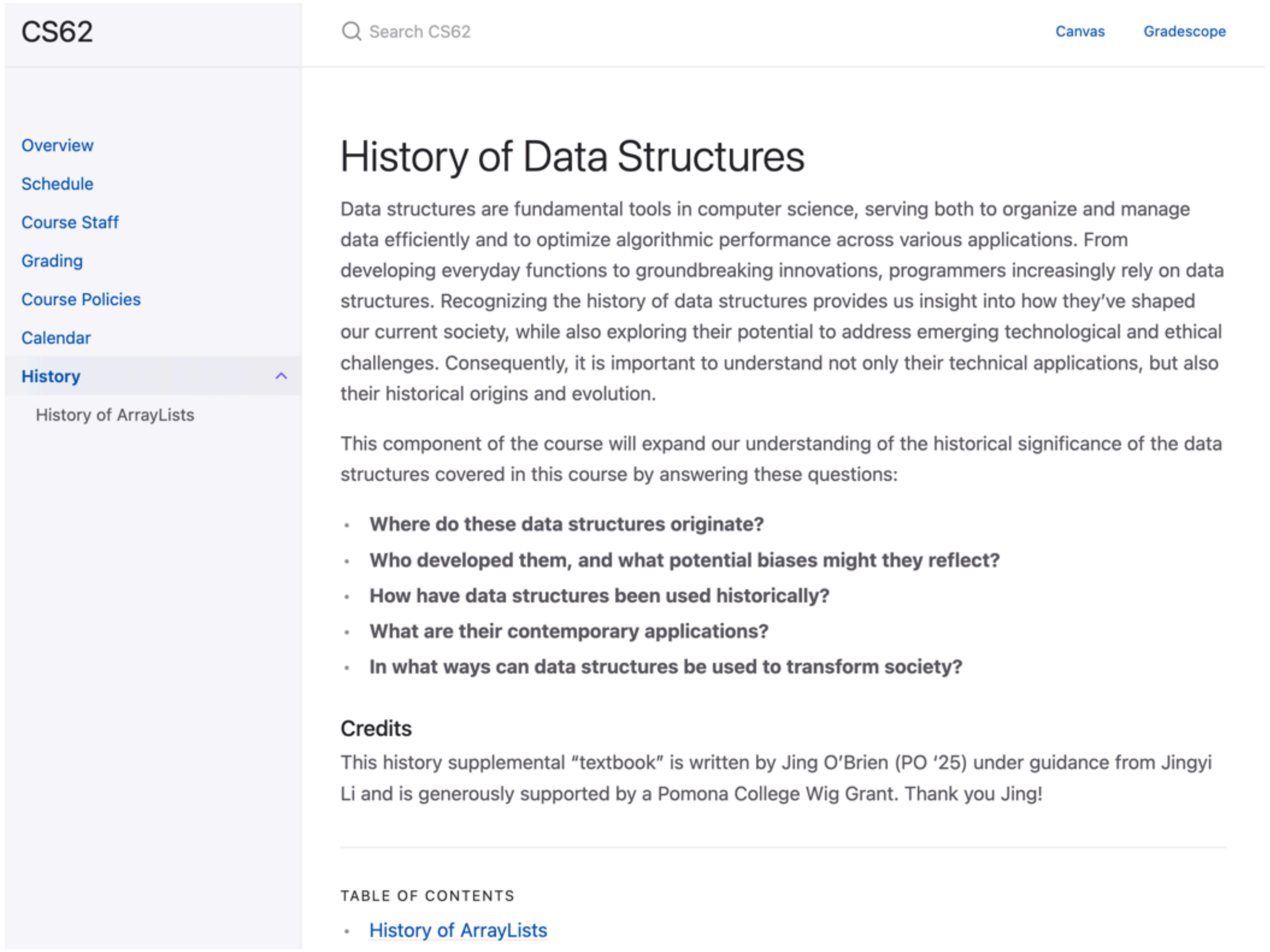
History and Context When examining a specific technology, what are the historical and cultural circumstances in which it emerged? When was it developed? For what purpose? How has its usage and function changed from then to today?

Power Dynamics and Hegemony Who benefits from this technology? At the expense of whose labor? How is this technology sold and marketed? What are the economic and political interests for the proliferation of this technology?

Developing Effective Long-Term Solutions What solutions are currently being implemented to address this labor/benefit asymmetry? In what ways do they reinforce or challenge the status quo? What are the long- and short-term implications of these solutions and who will benefit from them?

See https://kevinl.info/do-abstractions-have-politics/

Thus, we have a history textbook!



CS62

Search CS62

Canvas Gradescope

- Overview
- Schedule
- Course Staff
- Grading
- Course Policies
- Calendar
- History
  - History of ArrayLists

History of Data Structures

Data structures are fundamental tools in computer science, serving both to organize and manage data efficiently and to optimize algorithmic performance across various applications. From developing everyday functions to groundbreaking innovations, programmers increasingly rely on data structures. Recognizing the history of data structures provides us insight into how they've shaped our current society, while also exploring their potential to address emerging technological and ethical challenges. Consequently, it is important to understand not only their technical applications, but also their historical origins and evolution.

This component of the course will expand our understanding of the historical significance of the data structures covered in this course by answering these questions:

- **Where do these data structures originate?**
- **Who developed them, and what potential biases might they reflect?**
- **How have data structures been used historically?**
- **What are their contemporary applications?**
- **In what ways can data structures be used to transform society?**

Credits

This history supplemental "textbook" is written by Jing O'Brien (PO '25) under guidance from Jingyi Li and is generously supported by a Pomona College Wig Grant. Thank you Jing!

TABLE OF CONTENTS

- History of ArrayLists

- Last year, my RA Jing O'Brien researched the history of data structures. Read the pages to supplement learning purely "technical" material!

https://cs.pomona.edu/classes/cs62/history/

Why do we need data structures?

- To organize and store data so that we can perform efficient operations on them based on our needs: imagine walking to an unorganized library and trying to find your favorite title or books from your favorite author.
- We can define efficiency in different ways.
  - Time: How fast can we perform certain operations on a data structure?
  - Space: How much memory do we need to organize our data in a data structure?
  - Affordances: How does this data structure bake in assumptions for how we should think about the problem? What can we and can we not do with it?
- There is no data structure that fits all needs.
  - That's why we're spending a semester looking at different data structures.
  - So far, the only data structure we have encountered is arrays.
- The goal of this class is for you to understand trade offs between data structures, to choose the appropriate data structure for the appropriate problem

Types of operations on data structures

- Insertion: adding a new element in a data structure.
- Deletion: Removing (and possibly returning) an element.
- Searching: Searching for a specific data element.

- Replacement: Replacing an existing element with a new one (and possibly returning old).
- Traversal: Going through all the elements.
- Sorting: Sorting all elements in a specific way.
- Check if empty: Check if data structure contains any elements.

- Not a single data structure does all these things efficiently.
- You need to know both the kind of data you have, the different operations you will need to perform on them, and any technical limitations to pick an appropriate data structure.

Linear vs non-linear data structures

- Linear: elements arranged in a linear sequence based on a specific order.
  - E.g., Arrays, ArrayLists, linked lists, stacks, queues.
  - Linear memory allocation: all elements are placed in a contiguous block of memory. E.g., arrays and ArrayLists.
  - Use of pointers/links: elements don’t need to be placed in contiguous blocks. The linear relationship is formed through pointers. E.g., singly and doubly linked lists.
- Non-linear: elements arranged in non-linear, mostly hierarchical relationship.
  - E.g., trees and graphs.

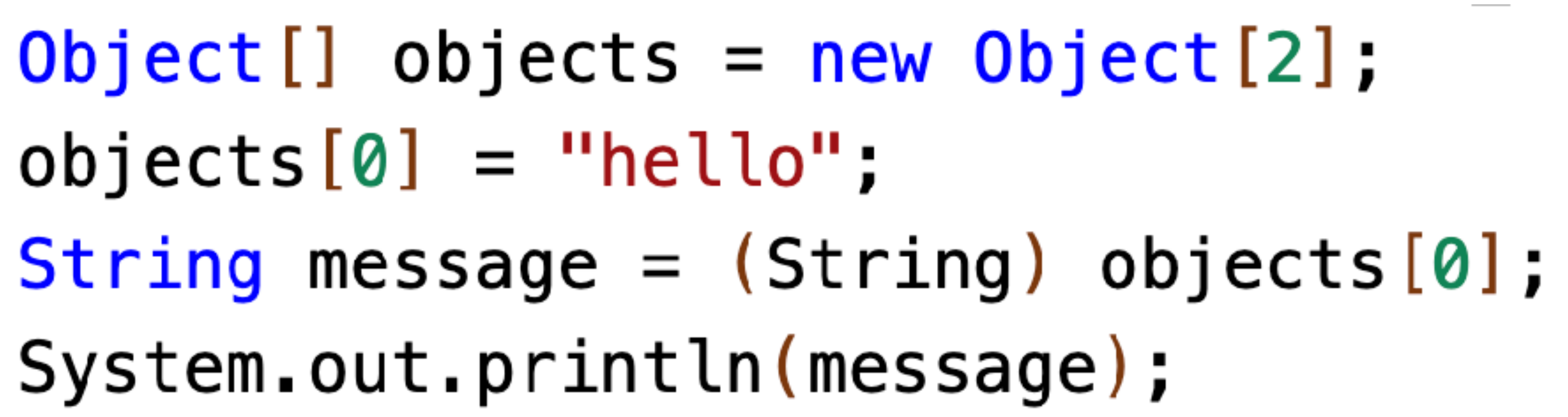
Towards building our own linear data structures…

- Arrays in Java are OK, but they’re not resizable. Let’s define our own data structure that supports adding elements, getting them at an index, removing them, etc…
- As such, we will build an **interface List** that forces any data structure that implements it to implement these operations.
- But what about types? We want our List interface to be able to hold objects of any type.

Generics

Lists should support any type of element

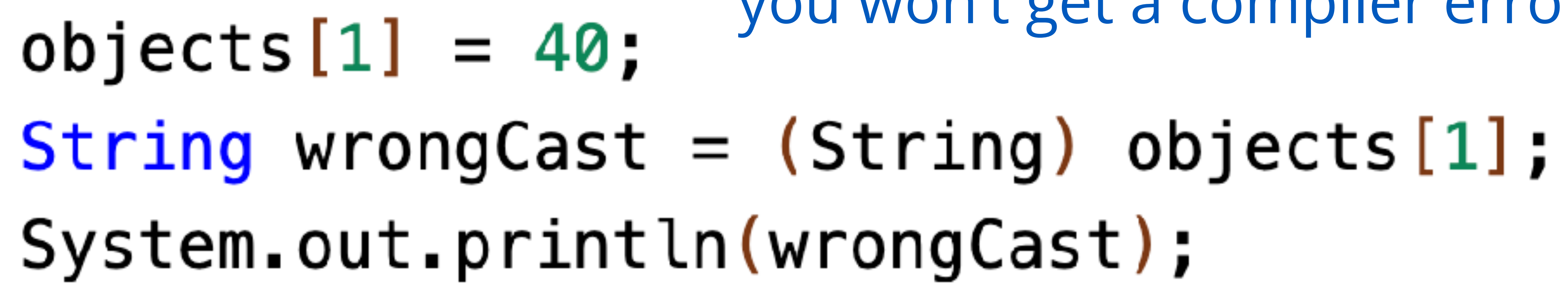
- We want our data structure to support any type of elements, as long as they are of the same type. We could use the class Object but this requires casting to the desired type:



```java
Object[] objects = new Object[2];
objects[0] = "hello";
String message = (String) objects[0];
System.out.println(message);
```

casting objects[0] to String, since it's type Object

but we might accidentally mix types! that's not OK! but you won't get a compiler error



```java
objects[1] = 40;
String wrongCast = (String) objects[1];
System.out.println(wrongCast);
```

results in runtime error: ClassCastException

Why generics help

- Generics are **type** parameters which stand in for any possible type.
- Let's say we want to create an interface that defines a new List data structure, but we don't know yet what type of object should be in the List. That's where we can us a *generic type*.

```
public interface List <E> {
    void add(E element);
    void add(int index, E element);
    E get(int index);
    boolean isEmpty();
}
```

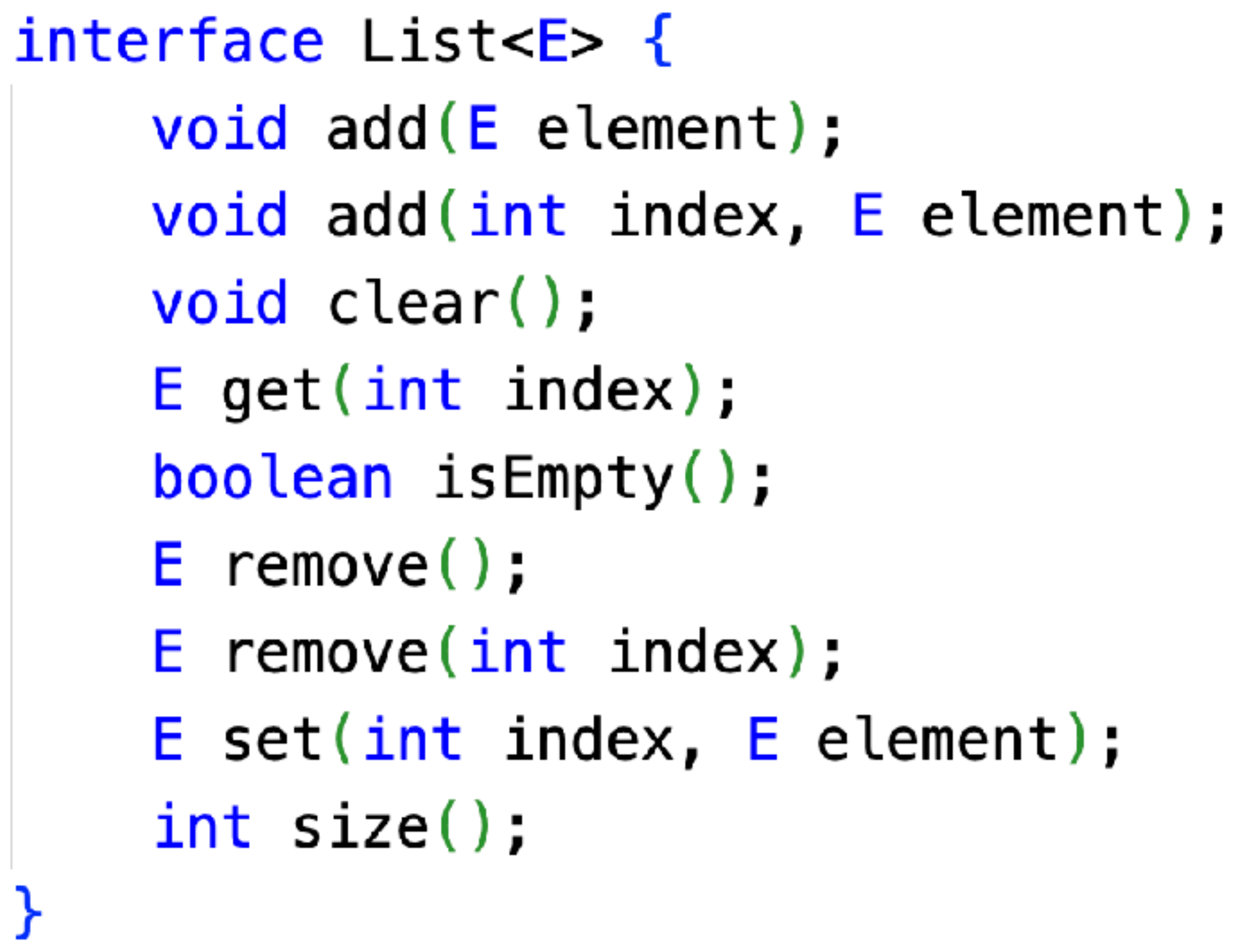
We use "<>" to denote generics in an interface or class declaration

By convention, generics are E (element) or T (type)

E stands in for int, String, PomonaStudent, etc… it can be any type when you actually create a new List object.

- Benefits:
  - Type safety (can't mix types in a list anymore)
  - No explicit casting needed anymore
  - Errors are caught at compile time instead of run time

Generics example



```java
interface List<E> {
    void add(E element);
    void add(int index, E element);
    void clear();
    E get(int index);
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
    int size();
}
```

```java
public class ArrayList<E> implements List<E>{…}
```

- In the invocation, all occurrences of the formal type parameters are replaced by the actual type argument

-
  ```java
  ArrayList<String> list = new ArrayList<String>();
  list.add("hello");
  String s = list.get(0);   // no cast
  ```

ArrayLists Intro

2 big limitations of Arrays

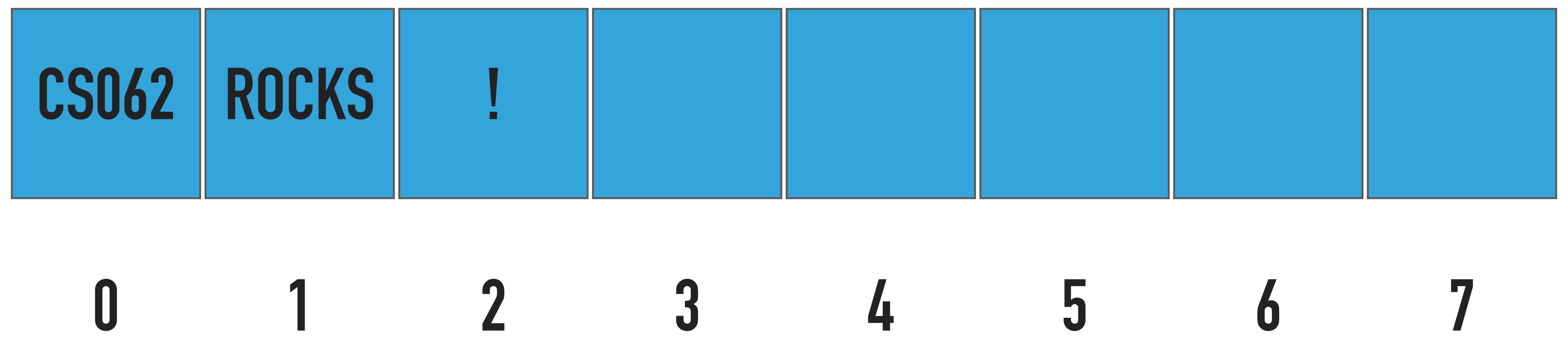
- Fixed-size.

- Do not work well with generics (would have to cast every time).
  - `E[] myArray = (E[]) new Object[capacity];`

- We want resizable arrays that support any type of object.
  - We just fixed the any type problem with generics.
  - Now we'll see how to implement resizable arrays.

ArrayList (or dynamic/growable/resizable/mutable array)

- Dynamic linear data structure that is zero-indexed.
- Sequential data structure that requires consecutive memory cells (ArrayList[0] is next to ArrayList[1] in memory)
- Implemented with an **underlying array** of a specific capacity.
  - But the user does not see that!

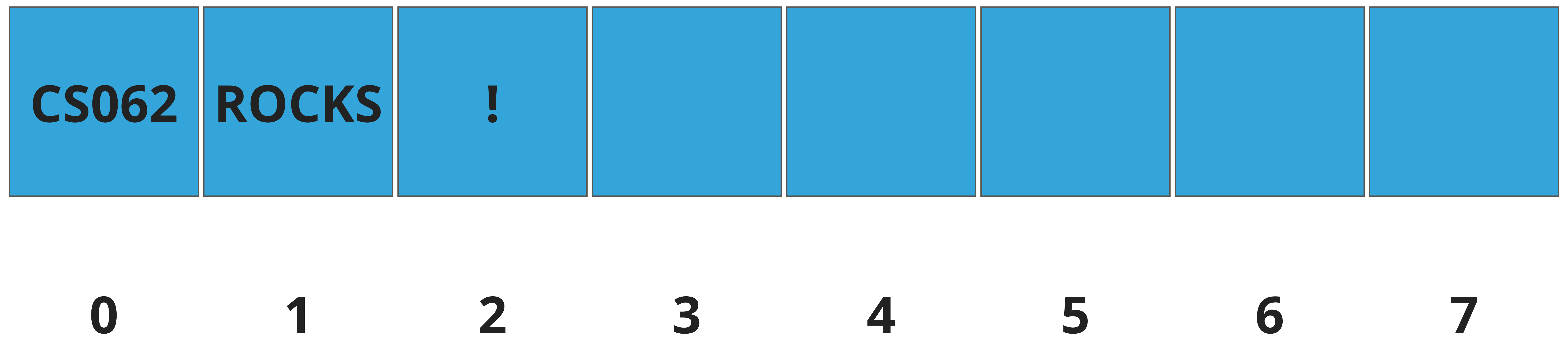


For example, an ArrayList of size 3 with elements {“CS62”, “ROCKS”, “!”}, is actually an Array of size 8 with {“CS62”, “ROCKS”, “!”, null, null, null, null, null}.

ArrayList behavior demo

ArrayLists

Remember, an ArrayList is implemented using Arrays



Capacity = 8

Size = 3

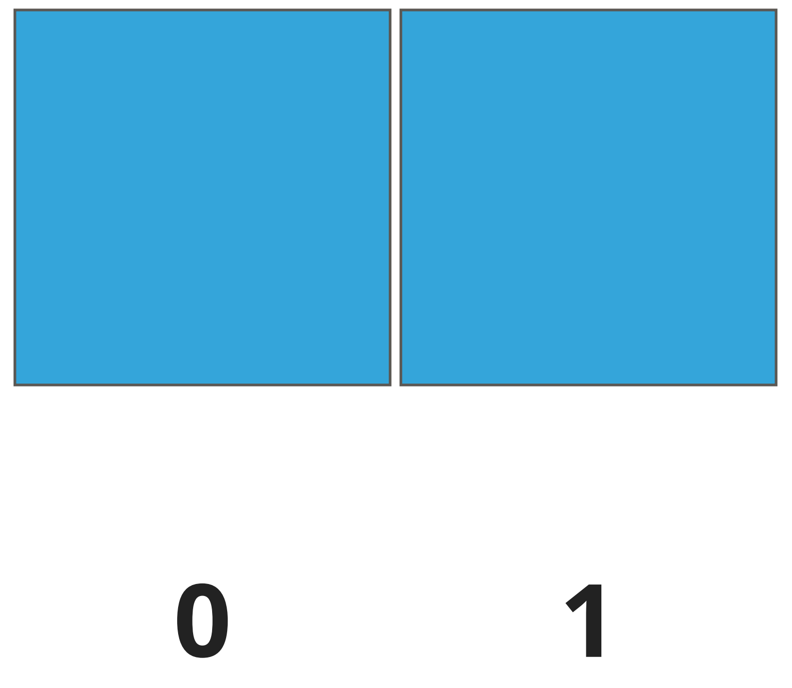
`ArrayList()`: Constructs an ArrayList

What should happen?

```
ArrayList<String> al = new ArrayList<String>();
```

`ArrayList()`: Constructs an ArrayList

When we first make an ArrayList, it is with a size 2 Array.



```
ArrayList<String> al = new ArrayList<String>();
```

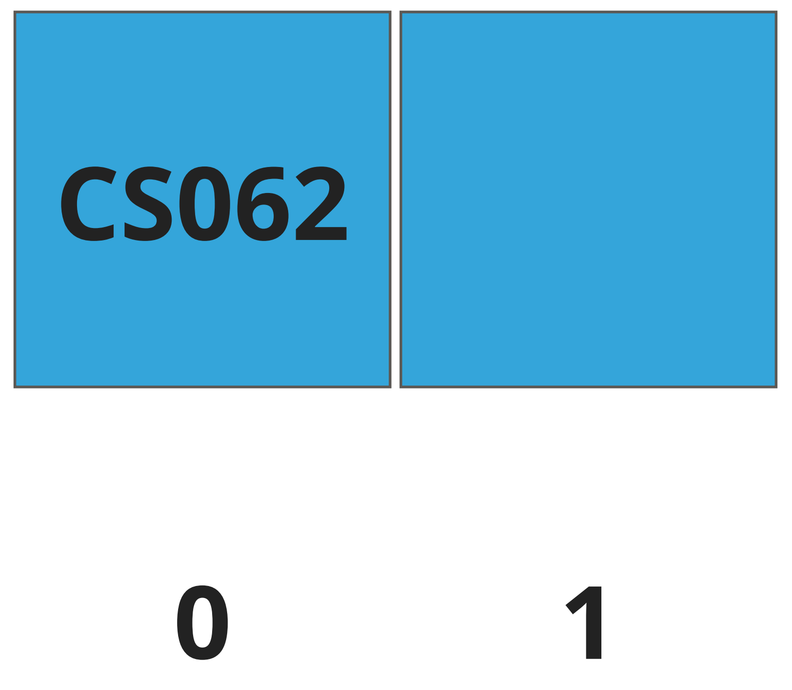
Capacity = 2

Size = 0

What should happen?

```
al.add("CS062");
```

`add(E element)`: **Appends the element to the end of the ArrayList**



```
al.add("CS062");
```

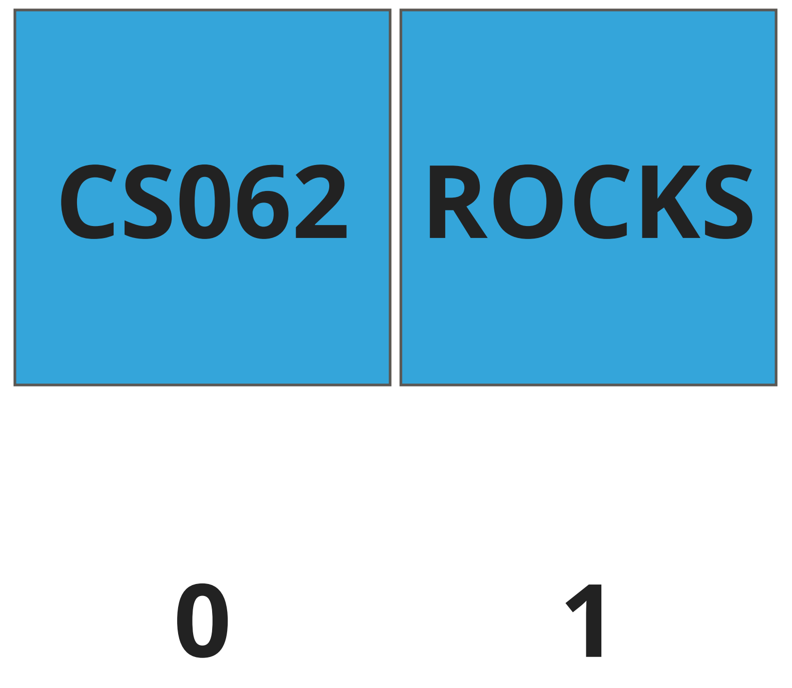
Capacity = 2

Size = 1

What should happen?

```
al.add("ROCKS");
```

`add(E element)`: Appends the element to the end of the ArrayList



```
al.add("ROCKS");
```

Capacity = 2

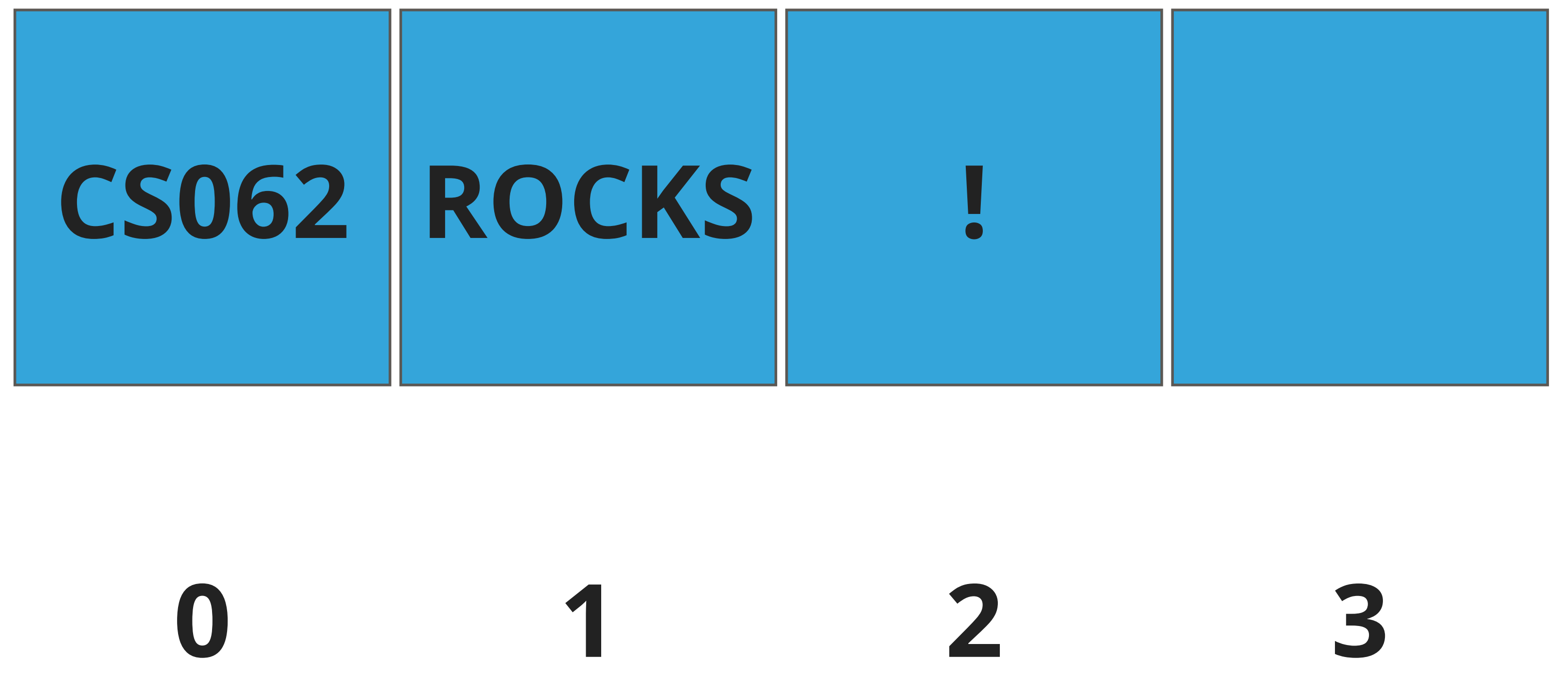
Size = 2

What should happen?

```
al.add("!");
```

`add(E element)`: Appends the element to the end of the ArrayList

- Double capacity since it's full and then add new element



```
al.add("!");
```

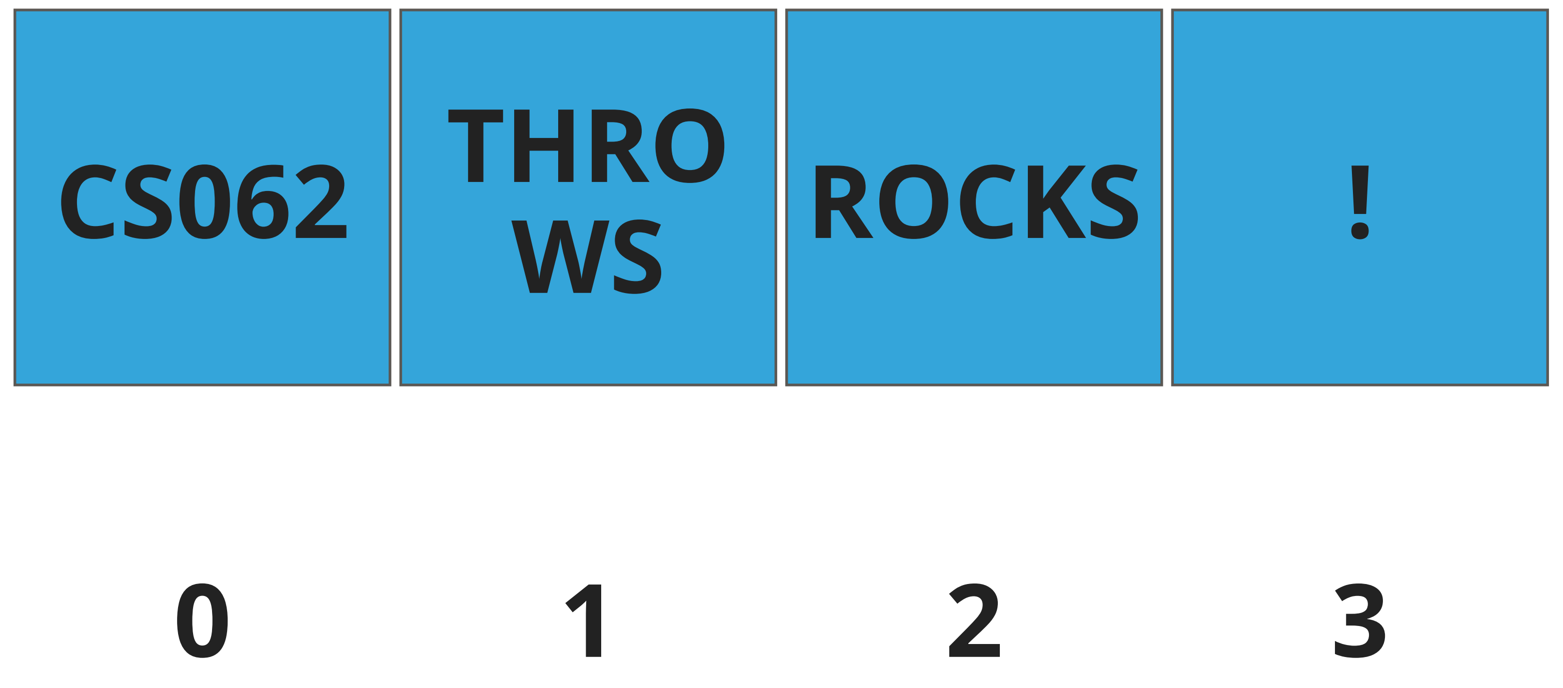
Capacity = 4

Size = 3

What should happen?

```
al.add(1, "THROWS");
```

`add(int index, E element)`: **Adds element at the specified index**



Capacity = 4

Size = 4

- Shift elements to the right

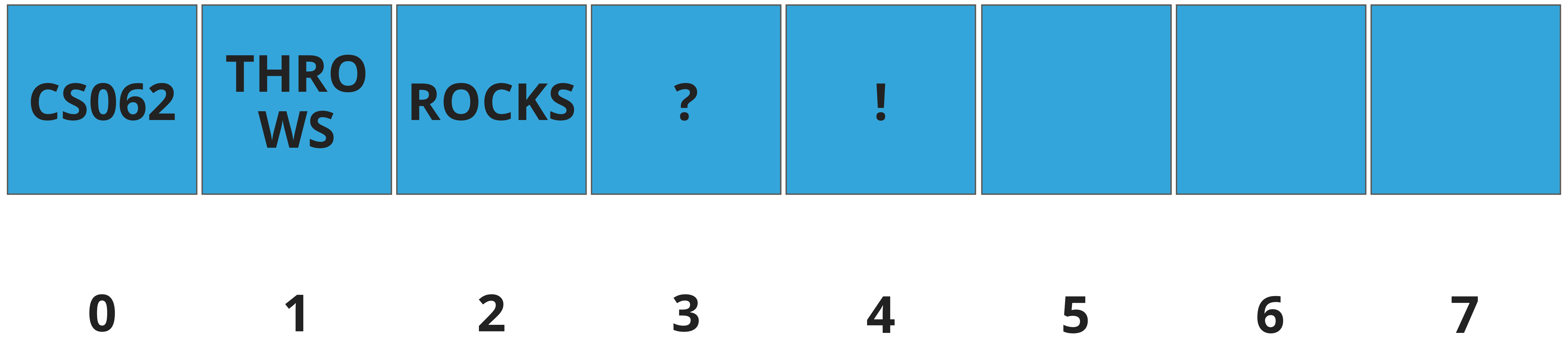
```
al.add(1, “THROWS”);
```

What should happen?

```
al.add(3, “?”);
```

`add(int index, E element)`: **Adds element at the specified index**

- Double capacity since full
- Shift elements to the right



Capacity = 8

Size = 5

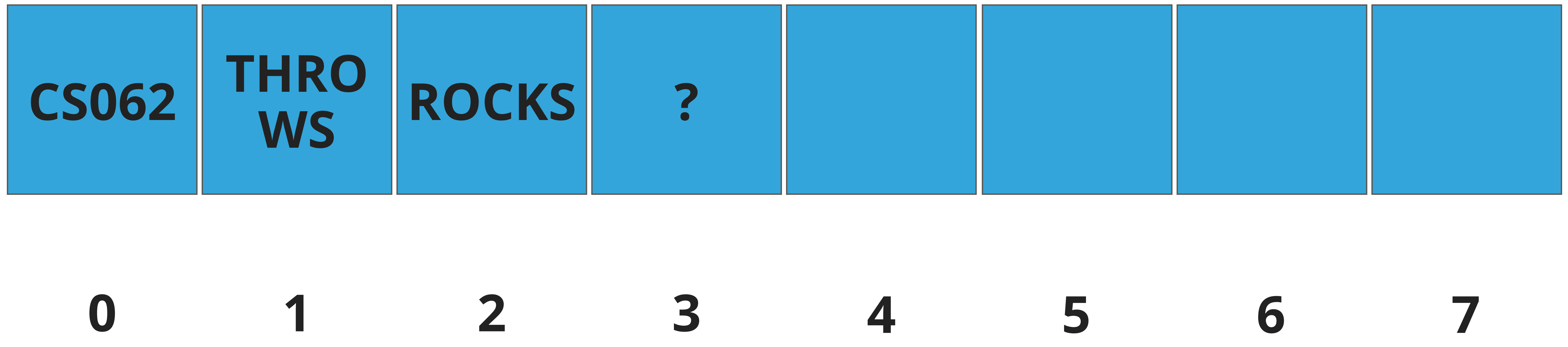
```
al.add(3, “?”);
```

What should happen?

```
al.remove();
```

`remove()`: Removes and returns element from the end of ArrayList

- Remove and Return last element



Capacity = 8

Size = 4

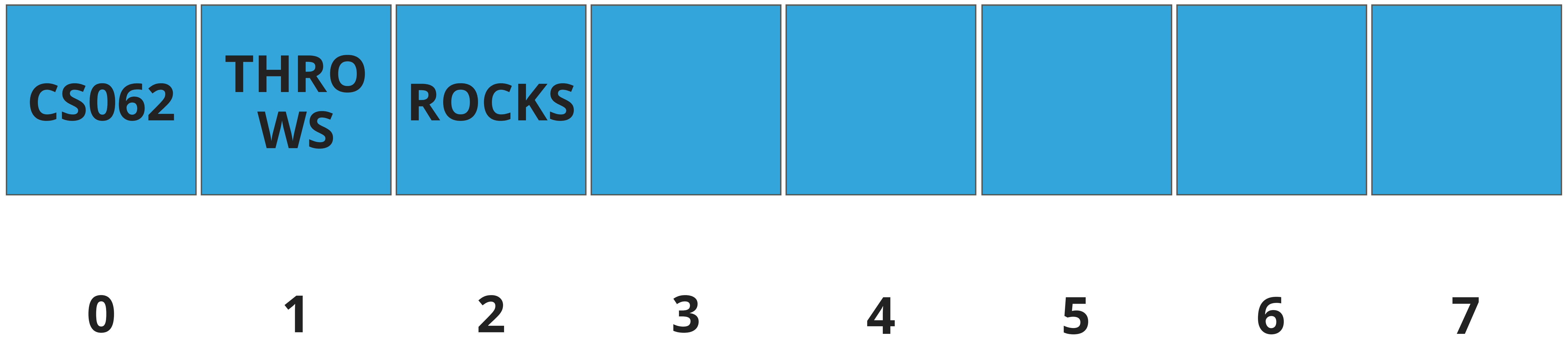
```
al.remove();
```

What should happen?

```
al.remove();
```

`remove()`: Removes and returns element from the end of ArrayList

- Remove and Return last element



Capacity = 8

Size = 3

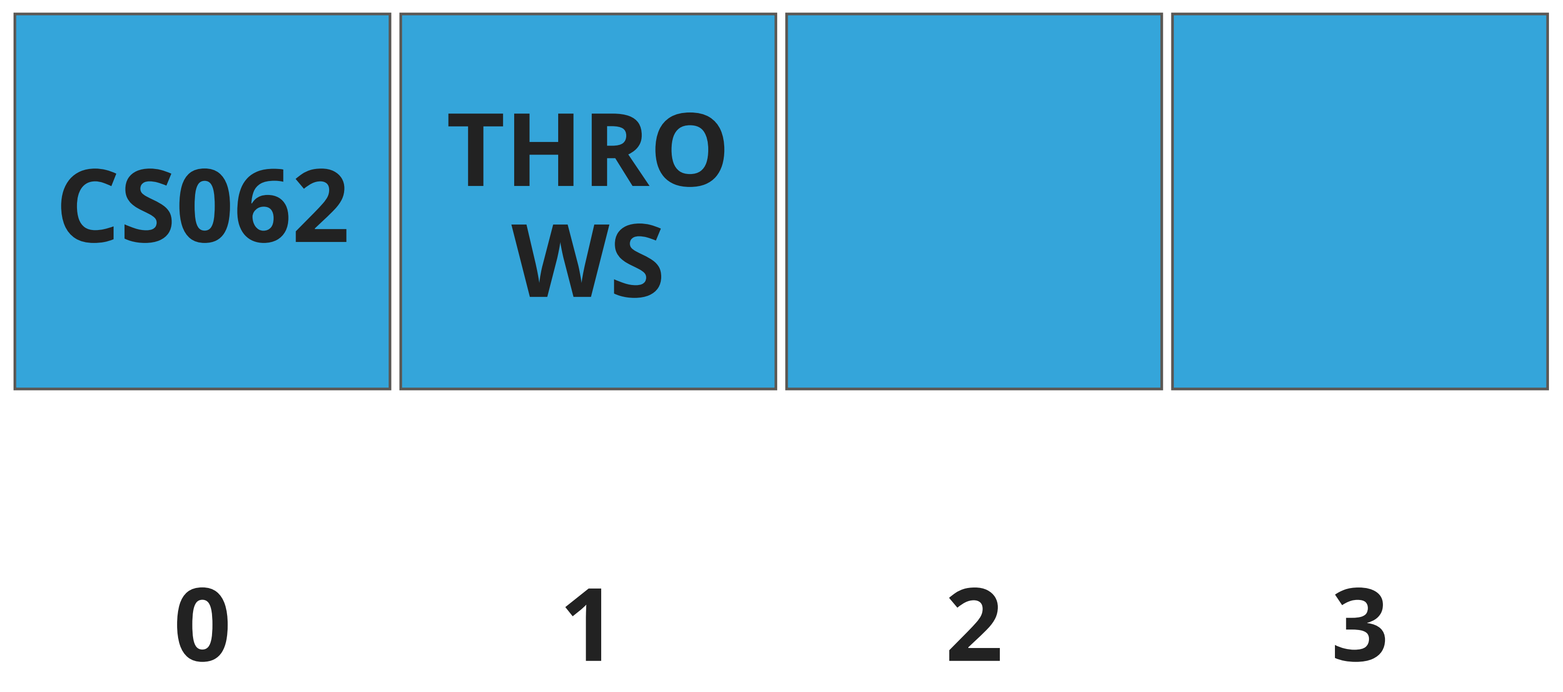
```
al.remove();
```

What should happen?

```
al.remove();
```

`remove()`: **Removes and returns element from the end of ArrayList**

- Remove and return last element
- Halve capacity when 1/4 full



```
al.remove();
```

Capacity = 4

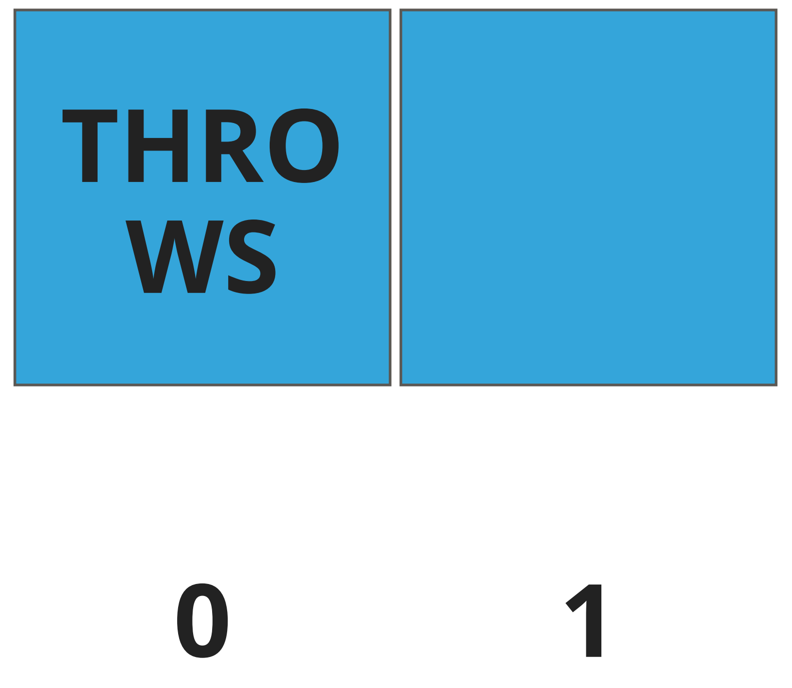
Size = 2

What should happen?

```
al.remove(0);
```

`remove(int index)`: Removes and returns element from specified index

- Remove element from index
- Shift elements to the left
- Halve capacity when 1/4th full



```
al.remove(0);
```

Capacity = 2

Size = 1

Reminder: Interface List

```java
interface List<E> {
    void add(E element);
    void add(int index, E element);
    void clear();
    E get(int index);
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
    int size();
}
```

List is an abstract data type.

ArrayLists (along with SinglyLinkedLists, DoublyLinkedLists) need to implement these following methods.

Standard Operations of `ArrayList<E>` class

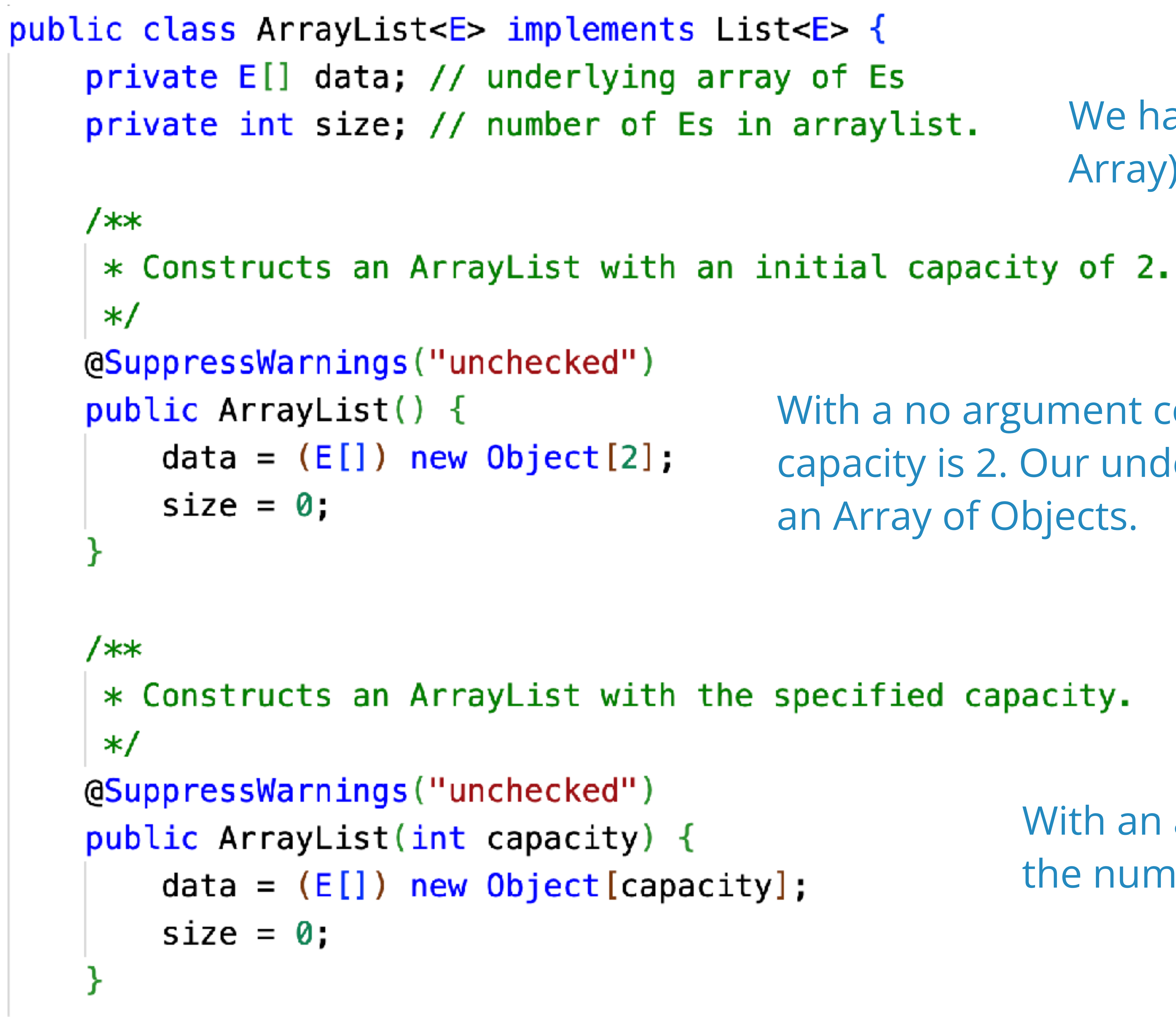
- `ArrayList()`: Constructs an empty ArrayList with an initial capacity of 2 (can vary across implementations, another common initial capacity is 10).
- `ArrayList(int capacity)`: Constructs an empty ArrayList with the specified initial capacity.
- `isEmpty()`: Returns true if the ArrayList contains no elements.
- `size()`: Returns the number of elements in the ArrayList.
- `get(int index)`: Returns the element at the specified index.
- `add(E element)`: Appends the element to the end of the ArrayList.
- `add(int index, E element)`: Inserts the element at the specified index and shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).
- `E remove()`: Removes and returns the element at the end of the ArrayList.
- `E remove(int index)`: Removes and returns the element at the specified index. Shifts any subsequent elements to the left (subtracts one from their indices).
- `E set(int index, E element)`: Replaces the element at the specified index with the specified element and returns the olde element.
- `clear()`: Removes all elements.

ArrayList Implementation

Our own implementation of ArrayLists

- We will implement the List interface.
- We will work with generics because we want ArrayLists to hold objects of any singular type.
- We will use an underlying array and we will keep track of how many elements we have in our ArrayList.

Instance variables & constructors



```java
public class ArrayList<E> implements List<E> {
    private E[] data; // underlying array of Es
    private int size; // number of Es in arraylist.

    /**
     * Constructs an ArrayList with an initial capacity of 2.
     */
    @SuppressWarnings("unchecked")
    public ArrayList() {
        data = (E[]) new Object[2];
        size = 0;
    }

    /**
     * Constructs an ArrayList with the specified capacity.
     */
    @SuppressWarnings("unchecked")
    public ArrayList(int capacity) {
        data = (E[]) new Object[capacity];
        size = 0;
    }
```

We have 2 instance variables: our data (in an Array) and size (number of elements present)

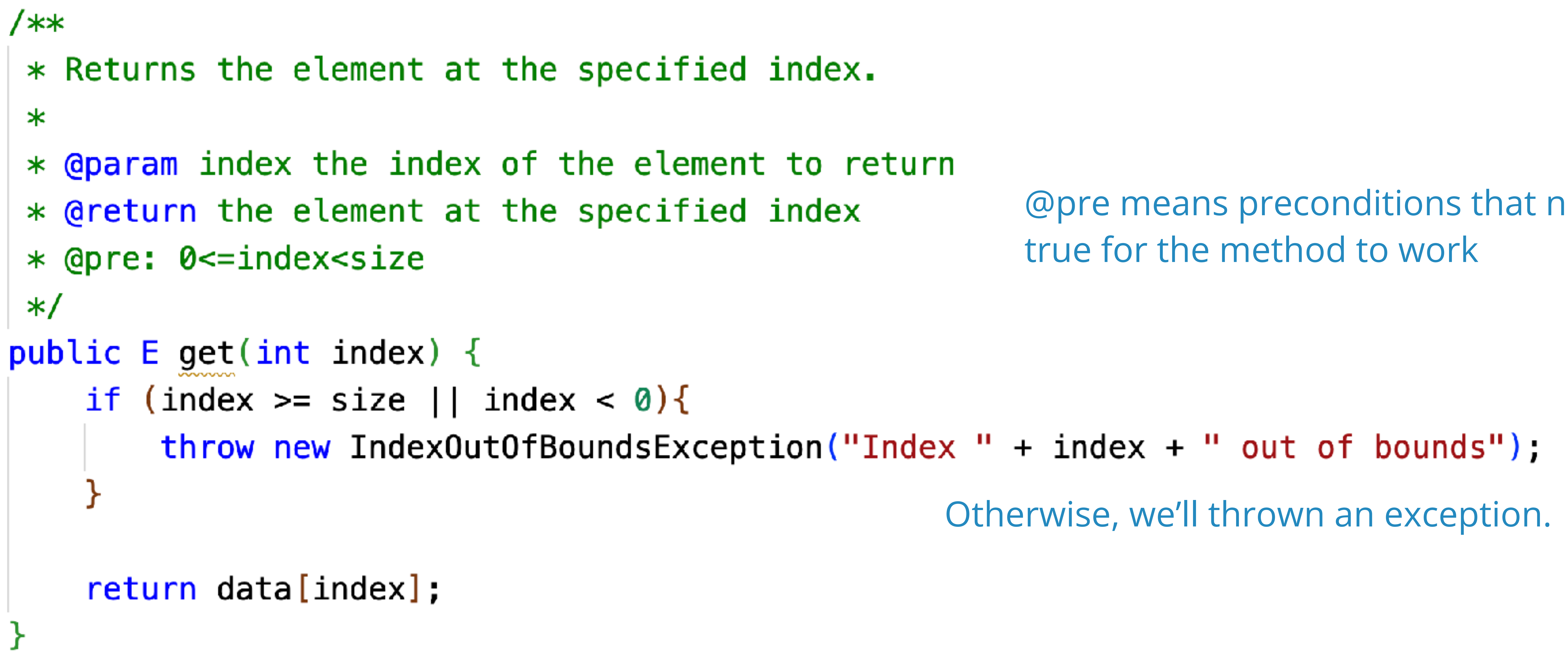
With a no argument constructor, the default capacity is 2. Our underlying implementation is an Array of Objects.

With an argument, we just set the capacity to the number passed in.

isEmpty(), size()

```java
/**
 * Returns true if the ArrayList contains no Es.
 *
 * @return true if the ArrayList does not contain any E
 */
public boolean isEmpty() {
    return size == 0;
}

/**
 * Returns the number of Es in the ArrayList.
 *
 * @return the number of Es in the ArrayList
 */
public int size() {
    return size;
}
```

Getting an element at index i



```java
/**
 * Returns the element at the specified index.
 *
 * @param index the index of the element to return
 * @return the element at the specified index
 * @pre: 0<=index<size
 */
public E get(int index) {
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }

    return data[index];
}
```

@pre means preconditions that need to be true for the method to work

Otherwise, we'll thrown an exception.

Remember, data is just a plain old Java Array, so we access elements by indexing into it.

Setting an element at index i, return replaced element

```java
/**
 * Replaces the element at the specified index with the specified E.
 *
 * @param index the index of the element to replace
 * @param element element to be stored at specified index
 * @return the old element that was replaced
 * @pre: 0<=index< size
 */
public E set(int index, E element) {
    //check if index is in range
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }
    //retreivew old element at index
    E old = data[index];
    //update index with new element
    data[index] = element;
    //return old element
    return old;
}
```

Adding to the end of the ArrayList

Where does the element go into the underlying Array? There’s no Array.append()…

What’s the relationship between where we should add and the variable size?

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	...

initial array size = 0

6	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	...

add(6) size = 1

6	9	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	...

add(9) size = 2

6	9	-1	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	...

add(-1) size = 3

A: The next item we want to add will go into position “size”.

ArrayList invariants

- The position of the next item to be inserted is always **size**.
- **size** is always the number of items in the ArrayList.
- The last item in the list is always in position size - 1.

Adding an element to the ArrayList

```java
/**
 * Appends the element to the end of the ArrayList. Doubles its capacity if
 * necessary.
 *
 * @param element the element to be inserted
 */
public void add(E element) {




    data[size] = element;
    size++;
}
```

What are we missing?

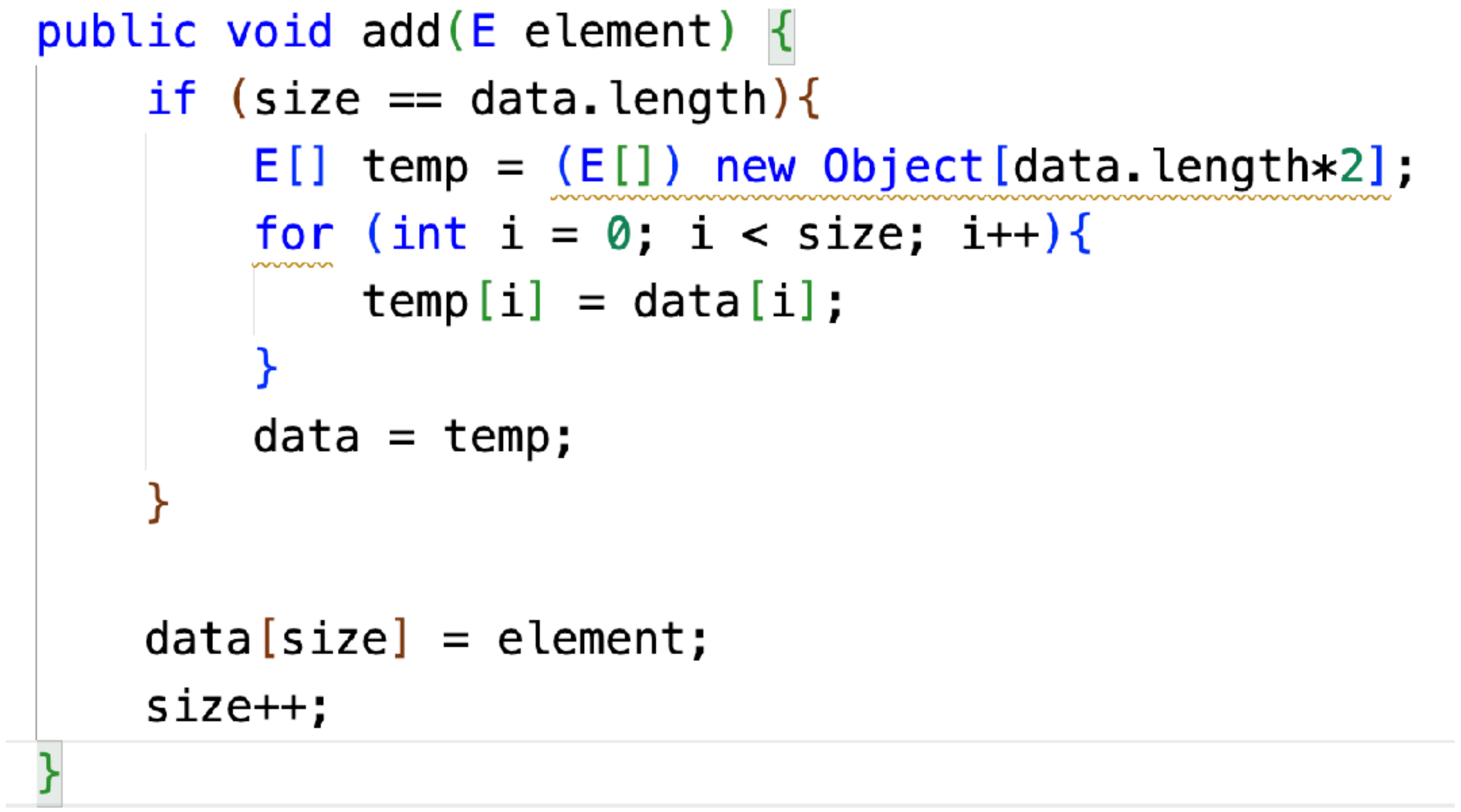
Resizing if it gets too big.

Assign element to index “size” (we know that size will always be the last empty index)

Increment size

Resizing ArrayList

- Resizing algorithm (plain text): If the Array gets full, create a new Array that's double the size. Copy all old elements into this new Array.



```java
public void add(E element) {
    if (size == data.length){
        E[] temp = (E[]) new Object[data.length*2];
        for (int i = 0; i < size; i++){
            temp[i] = data[i];
        }
        data = temp;
    }

    data[size] = element;
    size++;
}
```

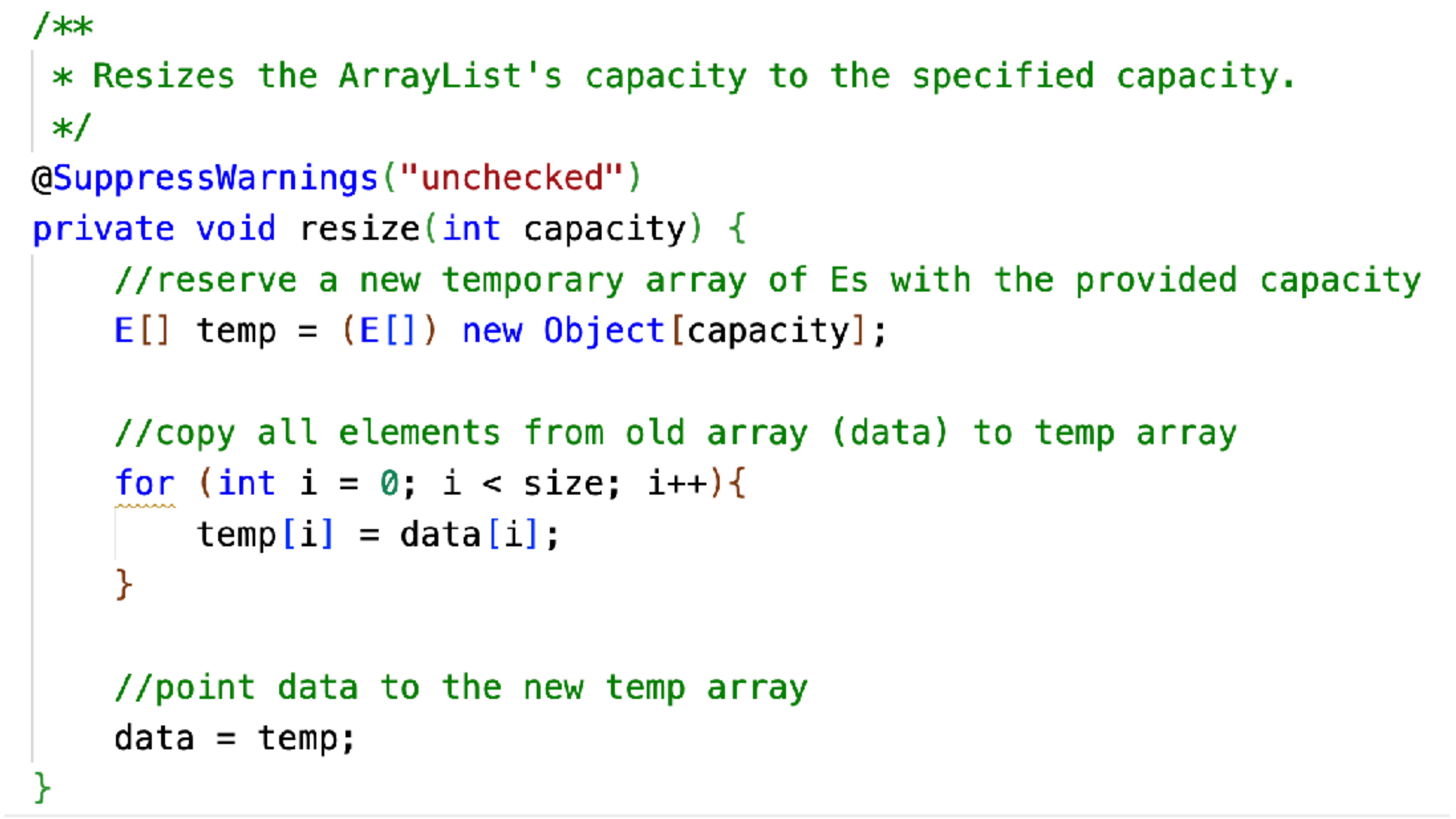
What could be better about how we've organized our code?

Resizing is really its own method. Let's move it out of add.

Should it take any arguments? If so, what?

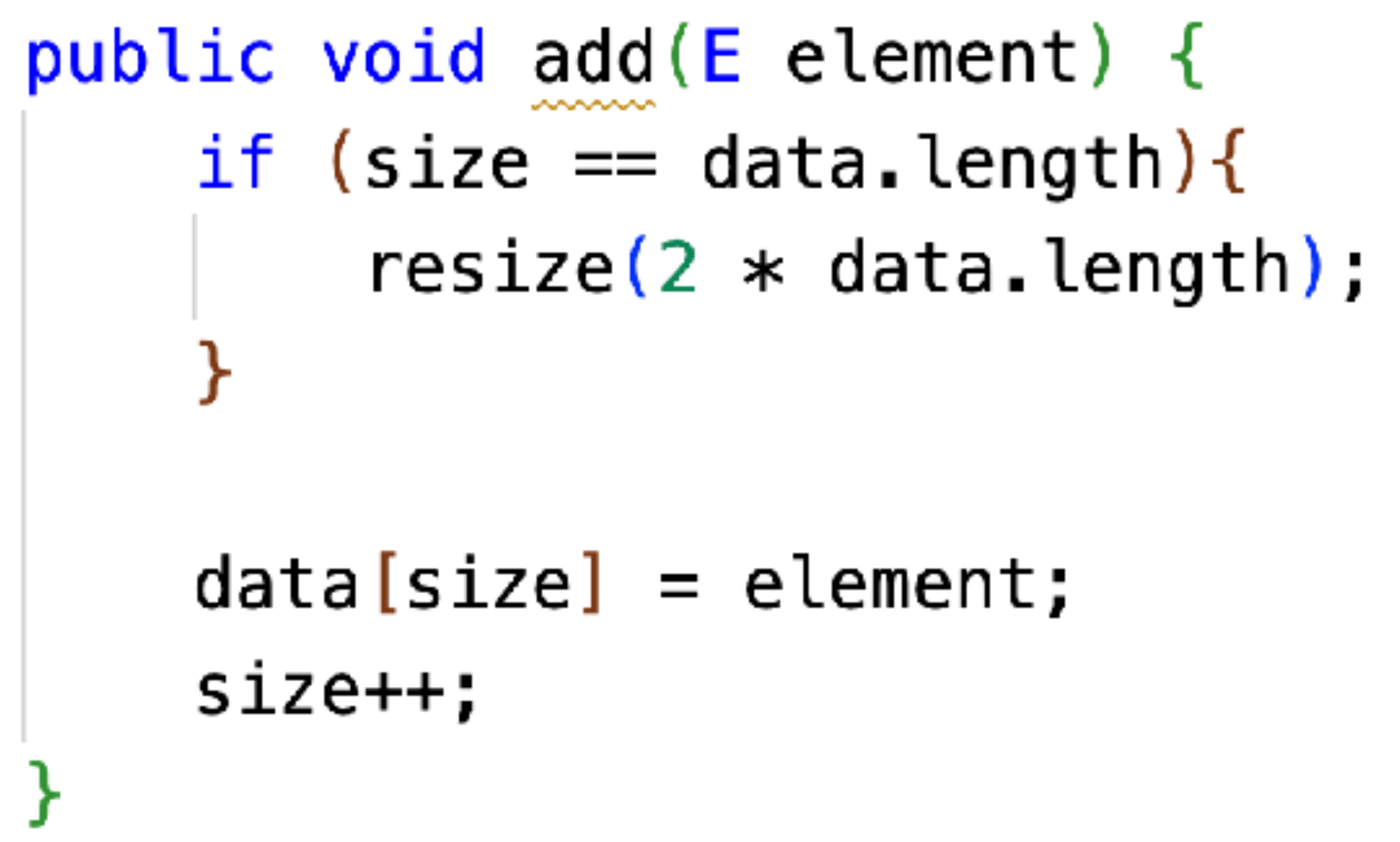
Resizing ArrayList

Takes 1 argument - the new capacity



```java
/**
 * Resizes the ArrayList's capacity to the specified capacity.
 */
@SuppressWarnings("unchecked")
private void resize(int capacity) {
    //reserve a new temporary array of Es with the provided capacity
    E[] temp = (E[]) new Object[capacity];

    //copy all elements from old array (data) to temp array
    for (int i = 0; i < size; i++){
        temp[i] = data[i];
    }

    //point data to the new temp array
    data = temp;
}
```

Calling resize in add means we should double the capacity



```java
public void add(E element) {
    if (size == data.length){
        resize(2 * data.length);
    }

    data[size] = element;
    size++;
}
```

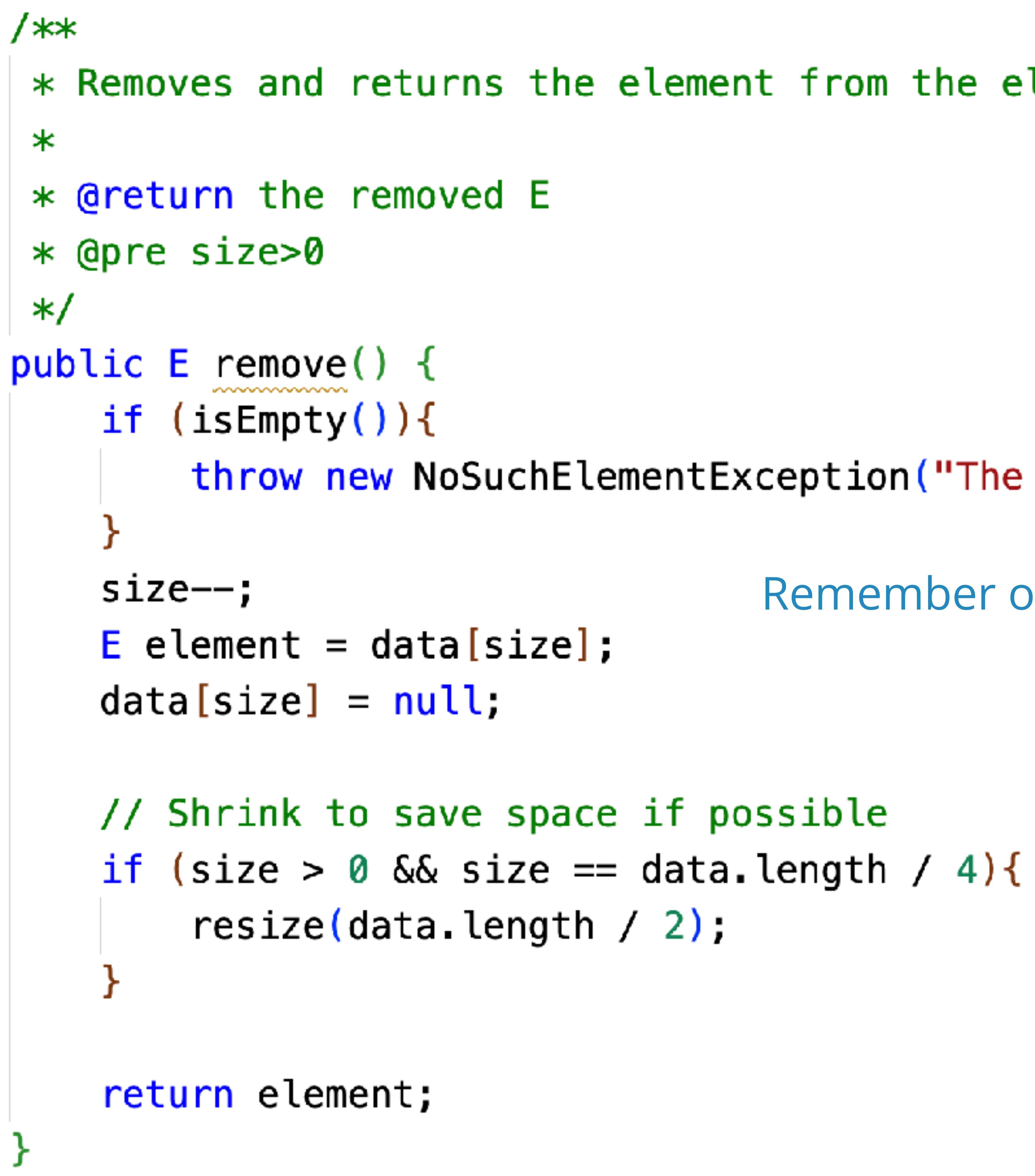
Worksheet time!

- Try writing adding an element at a specific index.
  - `void add(int i, E element) // inserting element at specific index`

Worksheet answer: add with index

```java
/**
 * Inserts the element at the specified index. Shifts existing elements to the
 * right and doubles its capacity if necessary.
 *
 * @param index the index to insert the element
 * @param element the element to be inserted
 * @pre: 0 <= index <= size
 */
public void add(int index, E element) {
    //check whether index is in range
    if (index > size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }

    //if full double in size
    if (size == data.length){
        resize(2 * data.length);
    }

    // shift elements to the right
    for (int i = size; i > index; i--){
        data[i] = data[i - 1];
    }
    //increase number of elements
    size++;
    //put the element at the right index in data
    data[index] = element;
}
```

Removing (and returning) the last element



```
/**
 * Removes and returns the element from the elementnd of the ArrayList.
 *
 * @return the removed E
 * @pre size>0
 */
public E remove() {
    if (isEmpty()){
        throw new NoSuchElementException("The list is empty");
    }
    size--;
    E element = data[size];
    data[size] = null;

    // Shrink to save space if possible
    if (size > 0 && size == data.length / 4){
        resize(data.length / 2);
    }

    return element;
}
```

Checking pre-condition

Remember our invariant that the last element is going to be at size - 1

Q: Why size == data.length / 4? Why not size <= data.length / 4?

A: Because we can only remove one element at a time, so it's guaranteed to eventually be equal

Clearing the ArrayList

```java
/**
 * Clears the ArrayList of all elements.
 */
public void clear() {

    // Help garbage collector.
    for (int i = 0; i < size; i++){
        data[i] = null;
    }

    size = 0;
}
```

Note that we don’t need to call remove() many times - let’s avoid unnecessary computation.

Iterate through the underlying Array and set everything to null - prevent “loitering”

Update size

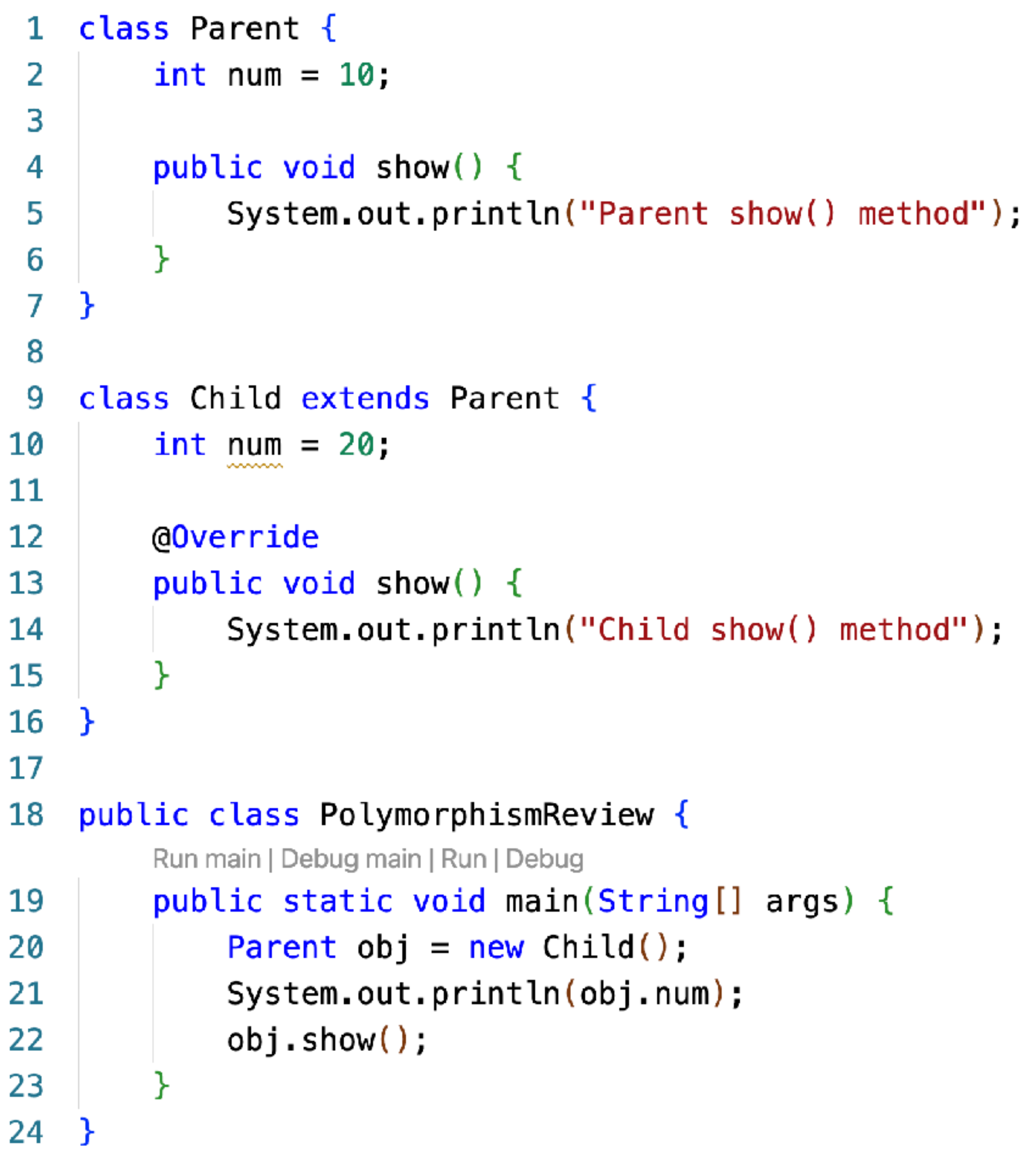
Lecture 4 wrap-up

- Abstract data types, often specified as interfaces, tell us what should happen but not how.
- An ArrayList is a re-sizable Array and implements the List<E> interface. “<E>” means it is a List of generic E types.
- HW2: Flippycard due Tues 11:59pm
- Lab this week isn’t coding based, but on learning Git
    - We may ditch Github classroom if problems persist

Resources

- Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
- ArrayLists: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- Resizable arrays in our textbook: https://algs4.cs.princeton.edu/13stacks/
- History of ArrayLists (made just for you all!) https://cs.pomona.edu/classes/cs62/history/arraylists/
- More practice problems behind this slide

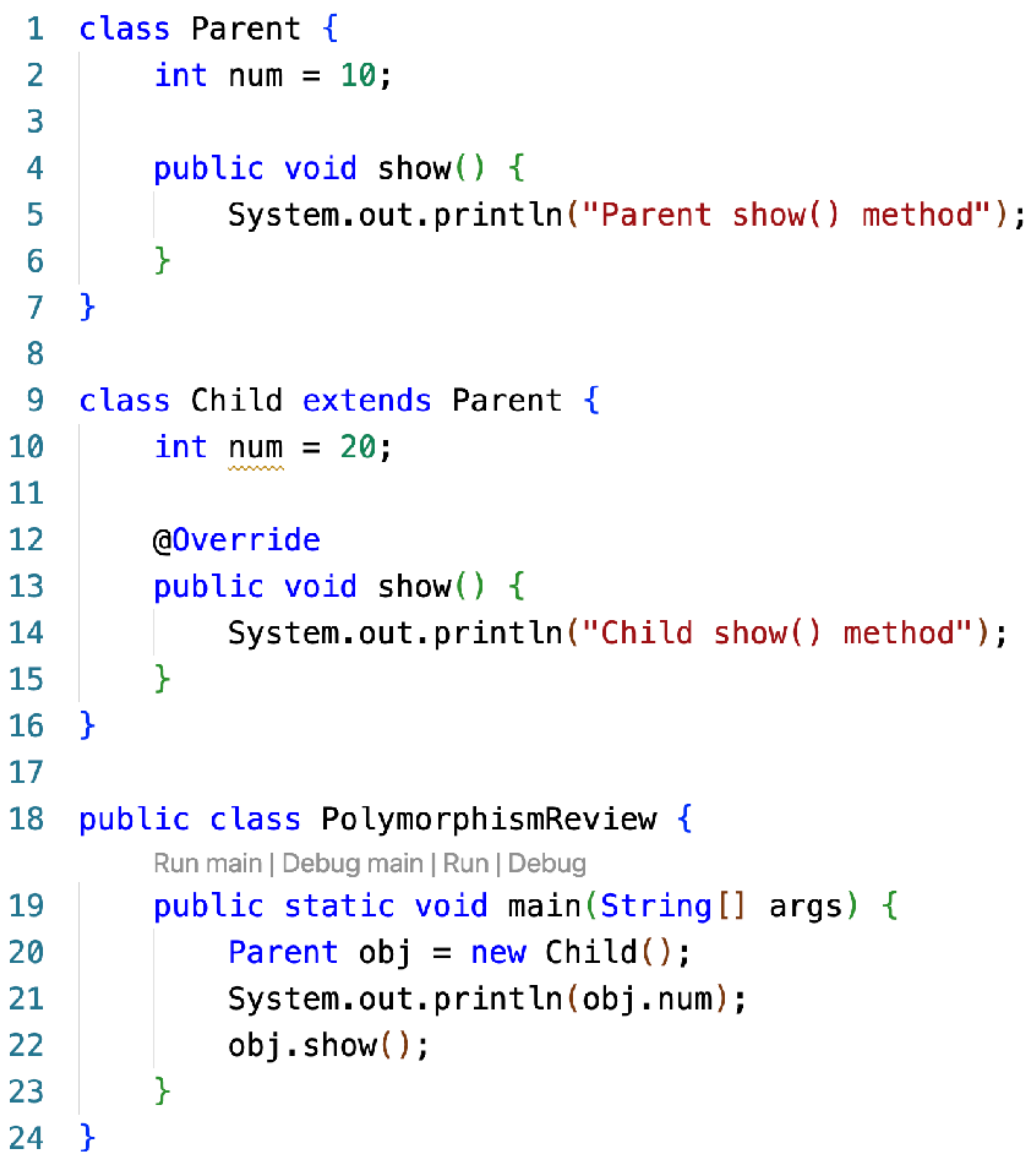
Review: polymorphism



```java
class Parent {
    int num = 10;

    public void show() {
        System.out.println("Parent show() method");
    }
}

class Child extends Parent {
    int num = 20;

    @Override
    public void show() {
        System.out.println("Child show() method");
    }
}

public class PolymorphismReview {
    public static void main(String[] args) {
        Parent obj = new Child();
        System.out.println(obj.num);
        obj.show();
    }
}
```

- What's printed? Why?
- Why does .show() behave differently from num?
- What keyword can be used to access the value of num in the parent class from the child class?
- How can we modify the code so we can print out 20 for num in main(), while keeping the type of obj as Parent?

Review: polymorphism



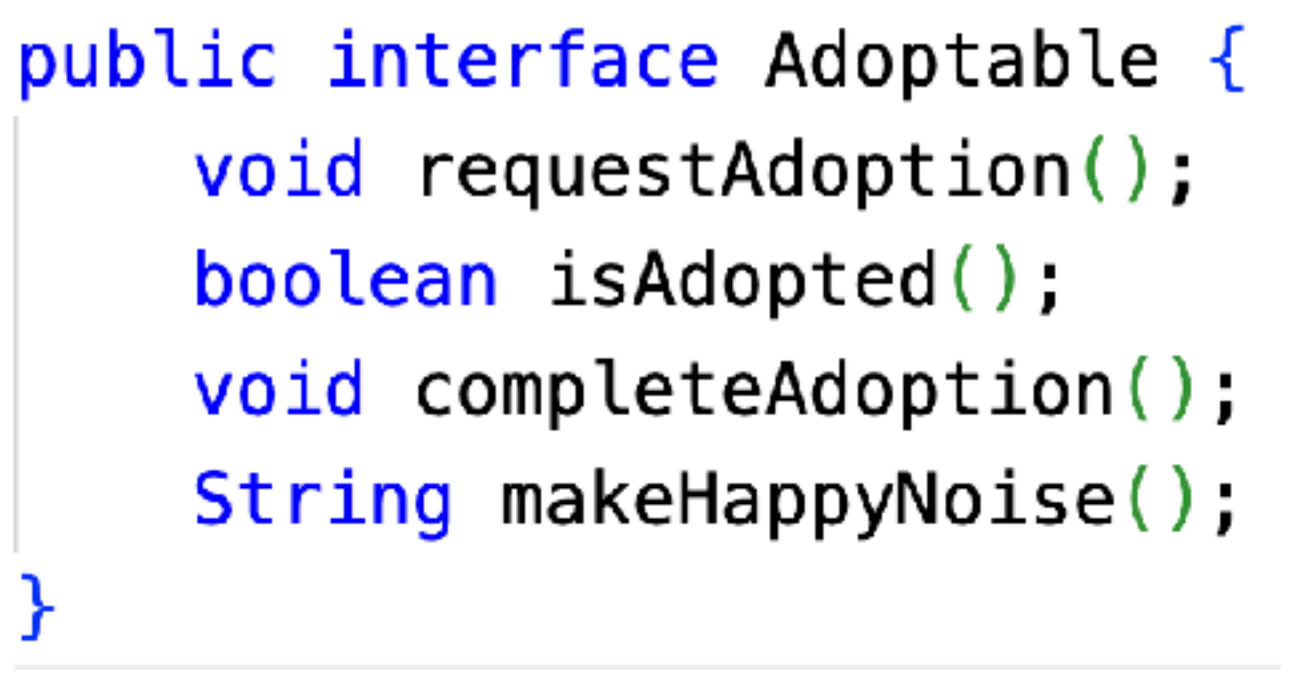
```java
class Parent {
    int num = 10;

    public void show() {
        System.out.println("Parent show() method");
    }
}

class Child extends Parent {
    int num = 20;

    @Override
    public void show() {
        System.out.println("Child show() method");
    }
}

public class PolymorphismReview {
    public static void main(String[] args) {
        Parent obj = new Child();
        System.out.println(obj.num);
        obj.show();
    }
}
```

- What's printed? Why?
  - 20, Child show() method
- Why does .show() behave differently from num?
  - .show() is overridden in the child class, while num hides the parent value
- What keyword can be used to access the value of num in the parent class from the child class?
  - super
- How can we modify the code so we can print out 20 for num in main(), while keeping the type of obj as Parent?
  - create a getter (getNum()) so overriding happens
- **SUMMARY**: instance methods get overridden, but variables (and static methods) are hidden

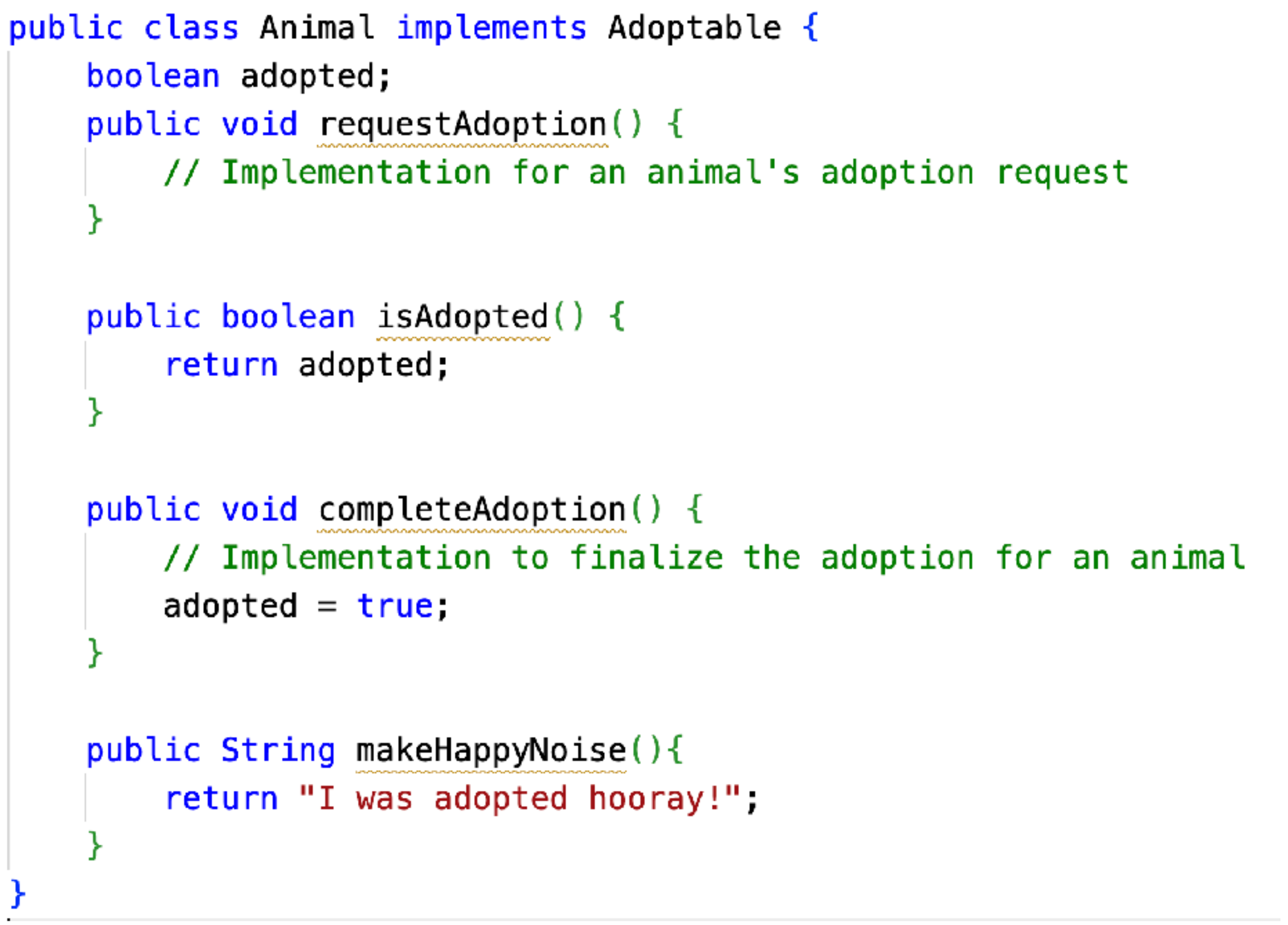
Bonus problem

- Create an interface called `Adoptable` that contains four abstract methods: a `void requestAdoption()`, `boolean isAdopted()`, `void completeAdoption(), and String makeHappyNoise()`.
- Have the class `Animal` implement the interface. You can provide some very minimal implementation of the methods so that you don't receive a compile-time error.
- Override the `makeHappyNoise()` in the `Cat` and `Dog` subclasses.

Bonus answers



```java
public interface Adoptable {
    void requestAdoption();
    boolean isAdopted();
    void completeAdoption();
    String makeHappyNoise();
}
```



```java
public class Animal implements Adoptable {
    boolean adopted;
    public void requestAdoption() {
        // Implementation for an animal's adoption request
    }

    public boolean isAdopted() {
        return adopted;
    }

    public void completeAdoption() {
        // Implementation to finalize the adoption for an animal
        adopted = true;
    }

    public String makeHappyNoise(){
        return "I was adopted hooray!";
    }
}
```

Bonus answers

```java
public class Cat extends Animal{

    private String fur;
    private static int catCounter;

    public Cat(String name, int age, int daysInRescue, String fur){
        super(name, age, daysInRescue);
        this.fur = fur;
        catCounter++;
    }
    public String getFur(){
        return fur;
    }
    protected void setFur(String fur){
        this.fur =  fur;
    }

    public String toString(){
        return super.toString() + "Cat fur: " + fur + "\n";
    }

    public String makeHappyNoise(){
        return "I am a happy cat!";
    }
}
```

```java
public class Dog extends Animal{

    private String breed;
    private static int dogCounter;

    public Dog(String name, int age, int daysInRescue, String breed){
        super(name, age, daysInRescue);
        this.breed = breed;
        dogCounter++;
    }

    public String toString(){
        return super.toString()+ "Dog breed: " + breed + "\n";
    }

    public String makeHappyNoise(){
        return "I am a happy dog!";
    }
}
```

Bonus problem

- Try writing removing an element at a specific index instead of the last element. Pay attention to how it resembles the remove last element method, or add at a specific index method.
  - `E remove(int i) // remove and return element at specific index`

Bonus answer: remove with index

```java
/**
 * Removes and returns the element at the specified index.
 *
 * @param index the index of the element to be removed
 * @return the removed element
 * @pre: 0<=index<size
 */
public E remove(int index) {
    //check if index is in range
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }
    //retrieve element at index
    E element = data[index];
    //reduce number of elements by 1
    size--;
    //shift all elements from the index until the end left 1
    for (int i = index; i < size; i++){
        data[i] = data[i + 1];
    }
    //set last element to null (since they've been shifted)
    data[size] = null;

    // shrink to save space if necessary
    if (size > 0 && size == data.length / 4){
        resize(data.length / 2);
    }
    //return removed element
    return element;
}
```

main difference from add at index - shift left instead of right