

Poll Everywhere

PollEv.com/javabear

text javabear to 22333



Is the ID class immutable?

```
public class ID {  
    private final String[] name;  
    private int age;  
  
    public ID(String[] name, int age) {  
        this.name = name; this.age = age;  
    }  
  
    public String firstName() {  
        return name[0];  
    }  
  
    public int age() { return age; }  
}
```

"rep exposure" 😞

Yes: state is fixed after constructor. **(A)**

No: age field is not marked final. **(B)**

No: because of the constructor

(C)

No: because of firstName()

(D)



Lecture 12: Collections and Generics

CS 2110

October 1, 2026

Today's Learning Outcomes

- 48. Describe the differences between *data structures* and *abstract data types*.
- 49. Implement a generic class or method with one or more generic type parameters. Use generic classes in client code.
- 50. Describe the semantics of *auto-boxing* and *auto-unboxing* and identify where they happen in a code snippet.
- 52. Compare and contrast the behaviors of ordinary Java arrays and dynamic array types such as Java's ArrayList.

Arrays as Collections

A **collection** is a type that groups together objects of another type.

First Example: Arrays

`String[]` is a collection of Strings

State: contiguous "block" of memory cells



Behaviors:

- query length $O(1)$

- read and write cell values
using index notation `a[i]`

$O(1)$ ~ does not depend on
`a.length` or `i`

"Random Access Guarantee"
of arrays

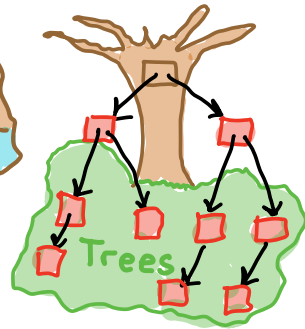
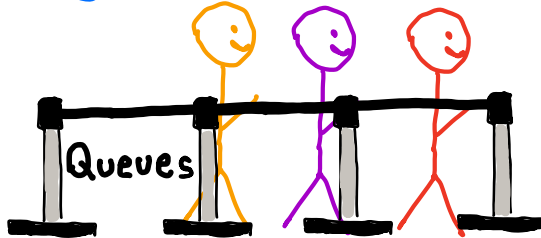
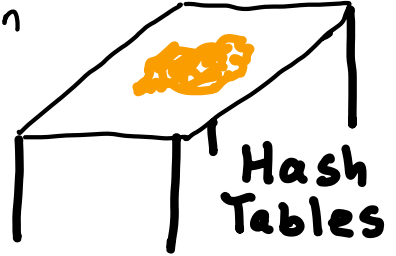
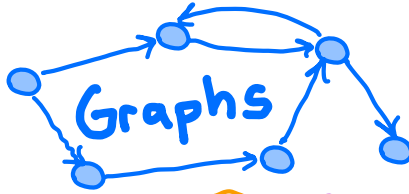
runtime guarantees

Other Collections



Each has its own

- structure/state representation
- behaviors / invariants
- performance guarantees



Abstract Data Types vs. Data Structures

The client and implementer interact with collections in different ways.
(i.e., collections impose an *abstraction barrier*)

Client:

- What operations are supported

How is the collection

- modified (add, remove, etc.)

- queried

- (runtime guarantees for behaviors)

ADT models behaviors

Implementer:

How do we

- represent data in memory

- define promised operations

- meet runtime guarantees

Data Structures answer
these "behind the scenes" ?s

Poll Everywhere

PollEv.com/javabear

text `javabear` to 22333



Which of the following is the most natural way to model an Abstract Data Type in Java?

guarantee behaviors, don't fix state

Abstract Class

(A)

(Concrete) Class

(B)

Record Class

(C)

Interface

(D)

The List ADT



A **list** is an ordered collection that grows to accommodate an arbitrary number of elements. Its elements are accessed by index.

Behaviors:

Accessing

- size
- get (at index)
- contains (element)
- indexOf (element)

Mutating

- add (at end)
- insert (at index)
- set (particular index)
- remove (at index)
- delete (element)



Coding Demo: `StringList` Interface



Using StringList as a Client

Use the StringList methods to complete the definition of the following method:

```
/** Replaces all instances of the given `word`  
 * with "****" in these `lyrics`. */  
static void censor(StringList lyrics, String word) {  
  
  
  
  
  
  
}
```

StringList

```
add(String elem): void  
size(): int  
get(int index): String  
contains(String elem): boolean  
indexOf(String elem): int  
set(int index, String elem): void  
remove(int index): String  
delete(String elem): void
```

Poll Everywhere

PollEv.com/javabear

text **javabear** to 22333



Which `StringList` method(s) did you use?

```
/** Replaces all instances of the given `word`  
 * with "****" in these `lyrics`. */  
static void censor(StringList lyrics, String word) {  
    while (lyrics.contains(word)) {  
        int i = lyrics.indexOf(word);  
        lyrics.set(i, "****");  
    }  
}
```

StringList

`add(String elem): void`
`size(): int`
`get(int index): String`
`contains(String elem): boolean`
`indexOf(String elem): int`
`set(int index, String elem): void`
`remove(int index): String`
`delete(String elem): void`

Generic Classes

What if we want to make a list of something other than Strings?

- Accounts, Points, other lists...

We'd need to define a new ADT interface
- lots of duplicated code ☹

Instead: Can we write one ADT to handle all data?
A new type of polymorphism (parametric)

Idea: Just like how we can parameterize a method on variables, we can parameterize a class/interface on a type called a generic type.

New angle-bracket < > Syntax



Coding Demo: Generic CS2110List



Using Generic Types

Generic types can take the place of a type name almost everywhere in a class.

- Fields

```
private T elem;  
private T[] storage;
```

- Local variables

- Parameters

```
public void add(T elem) {}
```

- Return types

```
public T get(int index) {}
```

cannot:

- invoke methods on objects of type T (for now)

can't enforce CTRR

- construct new T objects
don't know constructor args

- construct new T[]s
need weird hack...

Brief Aside: Auto-(un)boxing

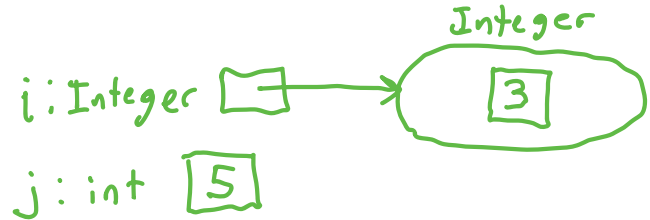
Generic types parameters can only be assigned **reference** types, not primitive types.

What if we want a list of ints?

Solution: Java has wrapper classes for each primitive type. E.g. Boolean for boolean, Integer for int ...

Conversion between primitives and wrapper class objects happens automatically

```
Integer i = 3; // auto-boxing  
int j = i + 2; // auto-unboxing
```



Poll Everywhere

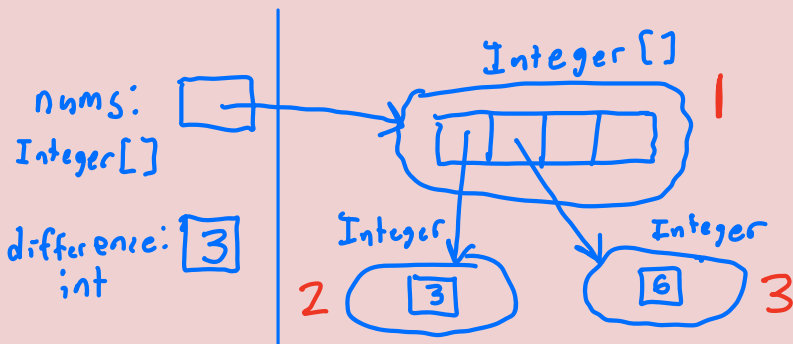
PollEv.com/javabear

text javabear to 22333



How many objects are constructed by this code?

```
1 Integer[] nums = new Integer[4];  
2 nums[0] = 3; } autoboxing  
3 nums[1] = 6; } auto-unboxing  
4 int difference = nums[1] - nums[0];
```



1

(A)

3

(B)

4

(C)

5

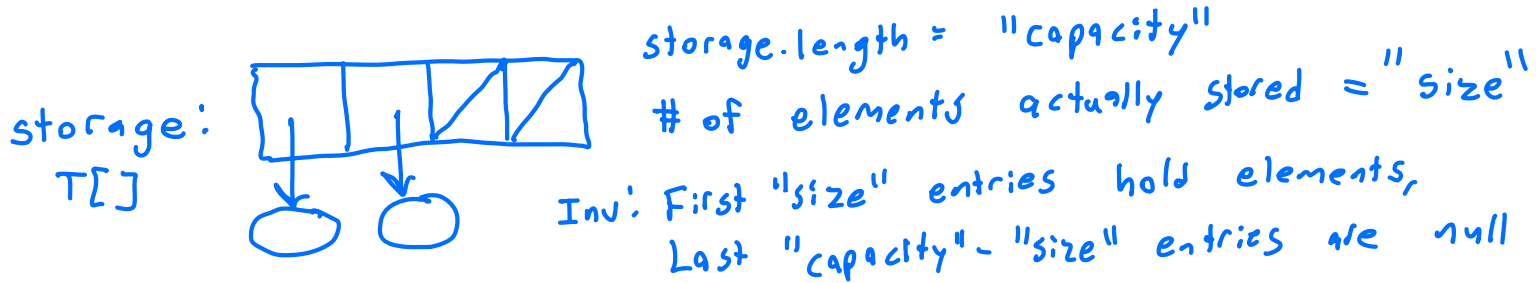
(D)

The Dynamic Array Data Structure

How can we use an array to represent the state of a list?

Need to handle unbounded size carefully.

Idea: When array becomes full, replace with new array and copy entries over.



Resizing strategy: add()/insert() when "size" = "capacity" $\Rightarrow$ double the capacity



Coding Demo: `DynamicArrayList` Design



invariantSatisfied() Methods

Data structures often rely on intricate class invariants to ensure correctness and achieve good performance.

Recall: The class invariant must hold at the start/end of every public method call.

`invariantSatisfied()` methods are a good development tool

- packages up checks for class invariant into one boolean method
- `assert invariantSatisfied()` before returning from any public mutating methods
- *"defensive programming against yourself"*



Coding Demo: `DynamicArrayList` Methods



Space Complexity of ArrayList

- size takes up $O(1)$ space
 - storage includes $O(N)$ "full" cells and $O(N)$ empty cells
(since it will always* be $\gg$ half full after resizing)
 - don't count space of elements, since we didn't construct them
- Overall: $O(N)$

Methods:

Most use $O(1)$ space

increaseCapacity() + add() / insert() allocate second $O(N)$ array during resizing copy

Time Complexity of DynamicArrayList

DynamicArrayList

insert(int index, T elem): void

remove(int index): T

size(): int

get(int index): T

set(int index, T elem): void

contains(T elem): boolean

indexOf(T elem): int

delete(T elem): void

add(T elem): void

$\{ O(N)$, need to shift elems when index=0

$O(1)$

$\{ O(1)$ random access guarantee

$\{ O(N)$ find() does linear search

$O(N)$ because of resize