CompSci 201, L4: Interfaces and Implementations, ArrayList
Logistics, Coming up

• Wednesday 9/7 (Today)
  • APT1 due, complete at least 4 for full credit

• Friday 9/9
  • APTs, Working with Sets, Strings, Git

• Monday 9/12
  • Project 0: Person201 due
  • Studying Sets and Maps
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 13.3

• 3 most important ADTs we study are all interfaces in Java!
  
  • **List**: An ordered sequence of values
  • **Set**: An unordered collection of *unique* values
  • **Map**: A collection that associates keys and values
The Java Collection Hierarchy

**Collection**
- **List**
  - **ArrayList**
  - **LinkedList**
- **Set**
  - **HashSet**
  - **TreeSet**
- **Map**
  - **HashMap**
  - **TreeMap**
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

Interfaces need an *implementing class* specified at creation.

```java
public class InterfaceExample {
    public static void main(String[] args) {
        List<String> strList = new List<>();
    }
}
```

What is an implementation? Can have any instance variables. Must override and implement *all* methods.

```java
import java.util.List;
public class DIYList implements List {
    // TODO Auto-generated method stub
    public int size() {
        return 0;
    }
}
```
Multiple Implementations of the Same Interface

2.4.1: List ADT using array and linked lists data structures.

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

Print result: 55, 88, 66

Array-based implementation

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>88</td>
<td>66</td>
</tr>
</tbody>
</table>

length: 3

Linked list-based implementation

head

| 55 | 88 | 66 |

length: 3

tail

A list ADT is commonly implemented using array and linked list data structures. But, a programmer need not have knowledge of which data structure is used to use the list ADT.
Implementations all have (at least) the same methods as the Interface

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

```java
public static List<String> inBothLists(List<String> aList, List<String> bList) {
    List<String> retList = new ArrayList<>();
    for (String s : aList) {
        if (bList.contains(s)) {
            retList.add(s);
        }
    }
    return retList;
}
```

Method doesn’t even “know” how aList and bList are implemented.

Since retList is an ArrayList which implements List, it is a valid return.
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.
DIY (do it yourself) ArrayList

Live Coding
WOTO

Go to duke.is/ggtfc

Not graded for correctness, just participation.

Try to answer without looking back at slides and notes.

But do talk to your neighbors!
How efficient is \textbf{ArrayList} \texttt{add}? 

For an \textbf{ArrayList} with \( N \) values, 2 cases:

1. Space left – One Array assignment statement, \textit{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 \textit{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

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

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

$\approx 2N$

Geometric series formula:

$\sum_{i=0}^{n} a r^i = a \frac{1 - r^{n+1}}{1 - r}$

Arithmetic growth

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

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

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

Arithmetic series formula:

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

9/7/22

Comp3ci 201, Spring 2022, Interfaces, ArrayList
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!

```java
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
Which version is more efficient? Small N?

Total number of values copied while growing ArrayList with different growth patterns

- Double when full (2N)
- Increase size by 100 when full $N^2/200$
Which version is more efficient? Larger N?

Total number of values copied while growing ArrayList with different growth patterns

- Double when full (2N)
- Increase size by 100 when full $N^2/200$
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

```
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)

15.equals(33) False, continue
12.equals(33) false, continue
21.equals(33) false, continue
33.equals(33) return true