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
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.

```java
public class Person {
    public String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}
```

**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.
- **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(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?

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

Try adding a Blob of every color-shape combination
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

[Diagram showing the hierarchy of interfaces and implementing classes]

What is a collection?

```java
public interface Collection extends Iterable,
```

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 JRE does not provide any direct implementations of this interface; it provides implementations of more specific interfaces like **Set** and **List**. This interface is typically used to pass collections around and manipulate them when 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 {
   public static void main(String[] args) {
      // List cannot be resolved to a type
      ListInterface li = new ListInterface();
   }
}

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

Multiple Implementations of the Same Interface

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().
Array List 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.
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**

\[ 1 + 2 + 4 + \cdots + N = \sum_{i=0}^{\log_2 N} 2^i \approx 2N \]

**Arithmetic growth**

\[ 1 + 101 + 201 + \cdots + N = \sum_{i=0}^{N/100} 1 + 100i \approx \frac{N^2}{200} \]

---

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
int n = 100;
int numIterations = 0;
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
        numIterations += 1; // 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

```java
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.

```
15 12
```

ArrayList contains revisited

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

```
list.contains(33)
```

```
15 12 21 33
```

33.equals(33) return true