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

• Today, Wednesday, 1/25
  • APT 1 due (need to do 4 for full credit)

• This Friday, 1/27
  • Discussion 2

• Next Monday 1/30
  • Prjoect 0: Person201 due (warmup project)

• Next Wednesday 2/1
  • APT 2 due
Reminder: Java4Python Resource

Might find the **Java4Python** extended intro to Java for Python programmers helpful for translating.

Contents:

- 1. Java for Python Programmers
  - 1.1. Preface
  - 1.2. Introduction
  - 1.3. Why Learn another programming Language?
    - 1.3.1. Why Learn Java? Why not C or C++?
  - 1.4. Lets look at a Java Program
  - 1.5. Java Data Types
    - 1.5.1. Numeric
      - 1.5.1.1. Import
      - 1.5.1.2. Declaring Variables
      - 1.5.1.3. Input / Output / Scanner
    - 1.5.2. String
    - 1.5.3. List
    - 1.5.4. Arrays
    - 1.5.5. Dictionary
  - 1.6. Conditionals
    - 1.6.1. Simple if
    - 1.6.2. if else
    - 1.6.3. elif
    - 1.6.4. switch
    - 1.6.5. Boolean Operators
  - 1.7. Loops and Iteration
    - 1.7.1. Definite Loop
    - 1.7.2. Indefinite Loops
  - 1.8. Defining Classes in Java
    - 1.8.1. Writing a constructor
    - 1.8.2. Methods
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 Record {
    public String displayName;
    private int uniqueID;
    public Record(String name, int id) {
        displayName = name;
        uniqueID = id;
    }
}
```

```java
public class PublicPrivate {
    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
(Im)mutability

• An object is **immutable** if you cannot change it after creation. Methods that change objects are called **mutators**.

• Java Strings are immutable, even though you can “append” to them. Creates a new String and copies all characters every time!

```java
String s = "Hello";
s += " World";  // More like
String sOld = "Hello";
String sNew = "" + sOld + " World";
```

(and then get rid of sOld)
Static belongs to the class

- Regular instance variables and methods are called on an object.
- Static methods are called on the class, do not use any instance variables. Often utility “functions”

```java
public class StaticExample {
    public static void main(String[] args) {
        String s = "Hello World!";
        System.out.println(s.split(" ")[0]);
        System.out.println(Math.sqrt(4.0));
    }
}
```

Note that `split` is called on a String object.

Whereas `sqrt` is called on the Math class.
APT and OOP, making a PSVM method

Suppose you’re working on the SandwichBar APT.

```java
public class SandwichBar {
    public int whichOrder(String[] available, String[] orders){
        // fill in code here
        return 0;
    }
}
```

Remember what you know about Java OOP:
• whichOrder is a regular method, need to call on an object of the SandwichBar class.
• whichOrder has parameters, need to supply those.
• All java programs must begin in a PSVM method.
APT and OOP, making a PSVM method

```java
public class SandwichBar {
    public int whichOrder(String[] available, String[] orders) {
        // fill in code here
        return 0;
    }
}

public static void main(String[] args) {
    String[] testAvailable = {"ham", "cheese", "mustard"};
    String[] testOrders = {"ham cheese"};
    SandwichBar testInstance = new SandwichBar();
    int testResult = testInstance.whichOrder(testAvailable, testOrders);
    System.out.println(testResult);
}
```

PSVM method can be in the same class or in a separate “driver” class in the same directory.

Creating test parameters, using example from APT site.

Make a SandwichBar object

Call the method
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

Interfaces

- List
  - ArrayList
  - LinkedList

- Set
  - HashSet
  - TreeSet

- Map
  - HashMap
  - TreeMap

Implementing Classes
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
1 public class InterfaceExample {

2 public static void main(String[] args) {

3 List<String> strList = new List<>();

4 }
```

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

```java
6 public class DIYList implements List {

7 @Override
8 public int size() {
9     // TODO Auto-generated method stub
10     return 0;
11 }
12 }
```
Multiple Implementations of the Same Interface

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

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

Print result: 55, 88, 66

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.
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

<table>
<thead>
<tr>
<th>15</th>
<th>12</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>12</td>
<td>21</td>
</tr>
</tbody>
</table>
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

\[ 1 + 2 + 4 + \cdots + N \approx \log_2 N \]

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

Arithmetic growth

\[ 1 + 101 + 201 + \cdots + N \approx \frac{N}{100} \]

\[ = \sum_{i=0}^{\frac{N}{100}} (1 + 100i) \approx \frac{N^2}{200} \]
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<i; j++) {
        numIterations += 1;
    }
}
```

```
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

```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.
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