ArrayList
API & Implementation

Lecture 9

Announcements

- Exam 1: Wednesday, September 30
  - Up to (and including) the lecture today (Lectures 1-9)
  - Review sessions: Fri during Lab time (9/25)
    Optional (bonus 5 pts for lab for attendance)
  - Review materials will be posted on Piazza

- Homework 5 is out: 3 Parts
  - Written and Program checkpoint first
  - Full Program due after exam as Homework 6
  - No code interview for Homework 5
  - AI log will be due for Homework 6 + code interview after HW6

Learning Objectives

- Last time: Runtime complexity, Big O formal definition

- Today:
  - Java's ArrayList class + API and methods
  - Implement the ArrayList class in Java (from scratch, as if it never existed)

ARRAYLISTS:
THE JAVA IMPLEMENTATION



The ArrayList Class

- For Java folks, an ArrayList is like an array, but:
  - It's a class, so we construct it and call methods on it.
  - It's resizable. It grows as we add elements and shrinks as we remove them (automatically, by design).

- For Python folks, an ArrayList is like a Python list, but:
  - We do not use subscript notation.

- ArrayLists (like arrays) are homogeneous.

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

ArrayList methods from `java.util.ArrayList<E>`

`boolean` **`add`**`(E obj)`
Appends `obj` to end of this list; returns `true`

`void` **`add`**`(int index, E obj)` `(0 <= index <= size)`
Inserts `obj` at position `index`

`void` **`clear`**`()`
Removes all the elements from this list.

`boolean` **`contains`**`(Object obj)`
Returns `true` if this list contains `obj`.

`E` **`get`**`(int index)`
Returns the element at position `index (0 <= index < size)`

`int` **`indexOf`**`(Object obj)`
Returns the index of the first occurrence of `obj` in this list,
or returns -1 if this list does not contain `obj`

ArrayList methods from `java.util.ArrayList<E>`

`boolean` **`isEmpty`**`()`
Returns true if the list is empty and false otherwise

`E` **`remove`**`(int index)` `(0 <= index < size)`
Removes element at position `index`;
Returns the element formerly at position `index`.

`boolean` **`remove`**`(Object obj)`
Removes the first occurrence of `obj`, if present;
Returns true if this list contained `obj`, `false` otherwise.

`E` **`set`**`(int index, E obj)` `(0 <= index < size)`
Replaces element at position `index` with `obj`;
Returns the element formerly at position `index`.

`int` **`size`**`()`
Returns the number of elements in the list.

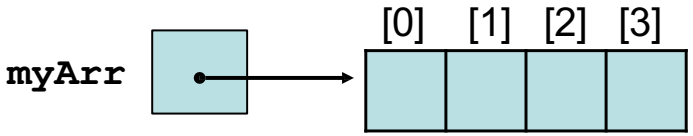
Arrays vs ArrayLists

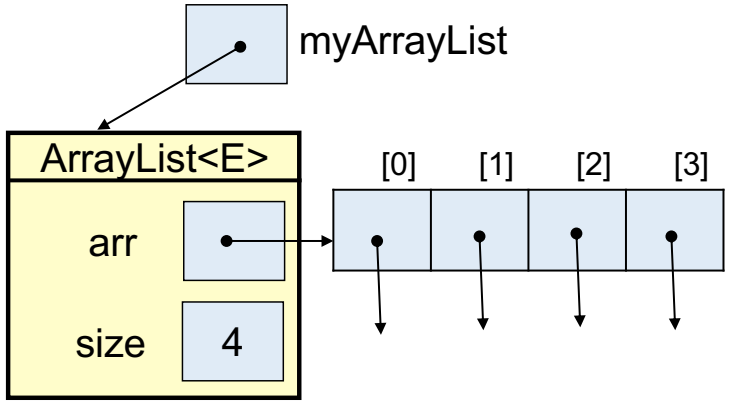
ARRAYS

- Immutable & does not expand dynamically; can't change size
- *Can* hold primitive data & object data values
- Arrays are not instances of a class
  - Special syntax `[ ]` for creating and accessing
  - Use `.length` to get size
  - Don't have methods
- All basic array operations run in O(1) time!

ARRAYLISTS

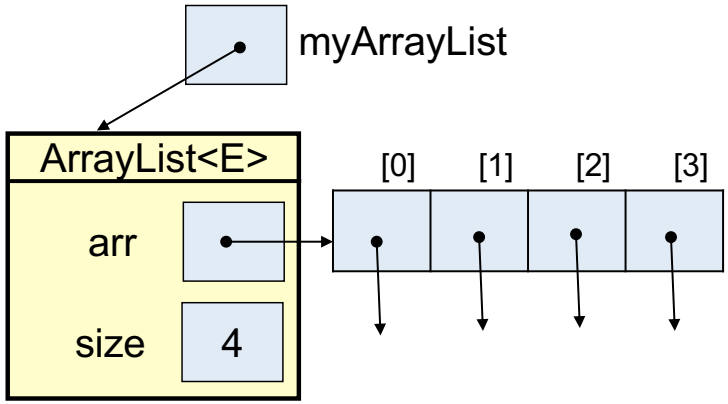
- Dynamic (contains a "dynamic" array as an underlying structure)
- Can **only** hold objects
- ArrayLists are instances of a class
  - Use a variety of methods for different operations
- Runtime depended on operation (i.e., do we need to shift when adding or removing elements)





For a given input size of n :

Method	Worst	Best
`int size()`	O(1)	
`get(int index)`	O(1)	
`set(int index, E obj)`	O(1)	
`add(E obj)`	O(1)*	
`add(int index, E obj)`	O(n)	O(1)*
`remove(int index)`	O(n)	O(1)
`remove(Object obj)`	O(n)	?
`contains(Object obj)`	O(n)	O(1)
`indexOf(Object obj)`	O(n)	O(1)
`clear()`	O(1)	
`isEmpty()`	O(1)	



* amortized

WRAPPER CLASSES



Wrapper Classes

- ArrayLists can **only** store references to objects, not primitive values.
- All primitive types have a corresponding object type (wrapper class).
- Example: `Integer`, `Double`, `Boolean`,...
- These Wrapper classes essentially create object classes for the primitives so they can act like other objects.

Java does conversion between primitive and wrapper types

```
ArrayList<Integer> intList = new ArrayList<Integer>();

intList.add(3);          // converts int to Integer
int n = intList.get(0); // converts Integer to int
```

```
Integer a = i++;     // auto-boxing
Integer b = j + 2;

int k = a + b;       // auto-unboxing

System.out.println(a.toString() + b.toString()); // concat
System.out.println(a + b);     // unboxed add
```

PRACTICE EXAMPLES

Example: countNamesWithLetter

```
// Returns number of names in given list, with given number of letters
public static int count(__ ArrayList<String> _ names, int numLetters) {
   int count = 0;
   for (int i = 0; i < names.size() ; i++) {
      if ( names.get(i).length() == numLetters    ____) {
         count++;
      }
   }
   return count;
}
```

Example: getNamesOfLength

```
// Returns a list of names in the given list that have the given number of letters
public static ArrayList<String> getNamesOfLength(ArrayList<String> names,
                                                 int numLetters) {

        ArrayList<String> result;

        result = new ArrayList<String>() ;

        for (int i = 0; i < names.size() ; i++) {

            if (names.get(i).length() == numLetters) {

                result.add(names.get(i)) ;
      }
   }
   return result;
}
```

Example: `removeNamesOfLength`

```
// Removes all names in list that have the given number of letters

public static void removeNamesOfLength(ArrayList<String> names, int numLetters) {

   for (int i = 0; i < names.size(); i++) {

      if (names.get(i).length() == numLetters) {

         names.remove(i);

      }
   }
}
```

Oops! When doesn't this code work correctly? It won't remove 2 consecutive names.

Example: `removeNamesOfLength`

```
// Removes all names in list that have the given number of letters

public static void removeNamesOfLength(ArrayList<String> names, int numLetters) {

    for (int i = 0; i < names.size(); i++) {

        if (names.get(i).length() == numLetters) {

            names.remove(i);

            i--;
        }
    }
}
```

Solution 1:
Decrement `i` after each removal. Ugly

Example: removeNamesOfLength

```
// Removes all names in list that have the given number of letters
public static void removeNamesOfLength(ArrayList<String> names, int numLetters) {
   int i = 0;
   while (i < names.size()) {
      if (names.get(i).length() == numLetters)
         names.remove(i);
      else
         i++;
   }
}
```

Solution 2:
Increment only when we don't remove.

Example: removeNamesOfLength

```
// Removes all names in list that have the given number of letters

public static void removeNamesOfLength(ArrayList<String> names, int numLetters) {

    for (int i = names.size()-1; i >= 0; i--) {

        if (names.get(i).length() == numLetters) {

            names.remove(i);

        }
    }
}
```

Solution 3:
Loop backward. Move only the elements you are keeping. Sweet.

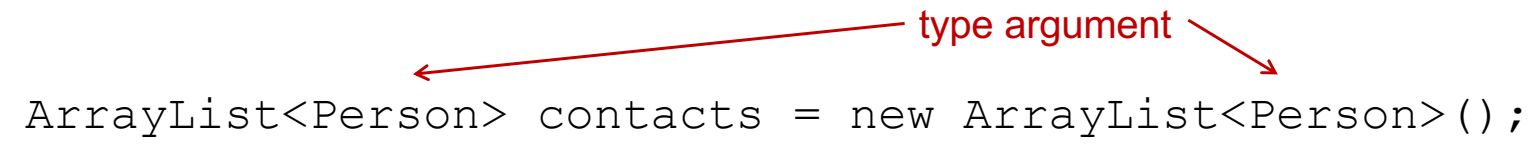
ARRAYLIST IMPLEMENTATION



Analyzing ArrayLists

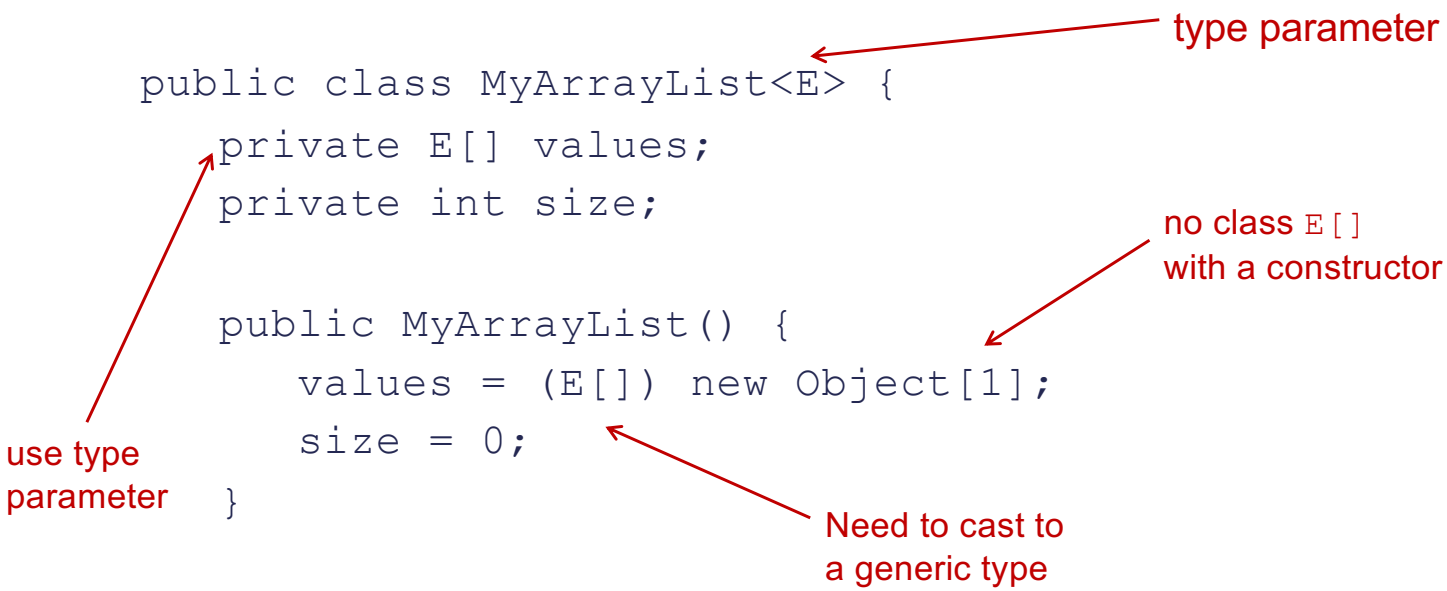
- We can use an array to implement an ArrayList, just like we did in the `ContactList` application.
- Because arrays underlie ArrayLists, the runtime for `size` / `get` / `set` in ArrayLists is also O(1).
- But `add` / `remove` at index are O(n) in the worst case.
- Let's see better why by implementing an ArrayList.

Syntax Recall: When we **declare** and **create** an instance of an ArrayList, we specify its type.



Implementing ArrayList

`MyArrayList` is a *generic* class, which specifies a type parameter.



Implementing ArrayList `size` & `get`

```
// Returns the number of elements in the list
public int size() {
   return size;
}

// Returns the element at position index
public E get(int index) {
   if (index < 0 || index >= size)
      throw new ArrayIndexOutOfBoundsException(index);

   else
      return values[index];
}
```

constructor creating an error

Note: "**`throw`**" is a Java statement, but a method header declares that it "**`throws`**".

Implementing ArrayList `set`

```
// Replaces element at position index with newValue.
// Returns the element previously at position index.

public  E  set(int index, E element) {
   if (index < 0 || index >= size)
      throw new ArrayIndexOutOfBoundsException(index);

   E  oldvalue = values[index];  // save old
   values[index] = element;
   return oldvalue;
}
```

Implementing ArrayList `add`

```
// Appends obj to the end of the list; returns true
public boolean add(E element) {
   growIfFull();

   values[size] = element;
   size++;
   return true;
}
```

Why return a boolean?

Header specified by the `Collection` Interface; `ArrayList` is a subclass

Why always return true? When would you return false?

when a `Collection` does not support duplicates (e.g. `Set`).

Implementing ArrayList `growIfFull`

```
// Helper method to double the length of values
private void growIfFull(){

   if (size == values.length) {
      E[] bigger = (E[]) new Object[size * 2];

      for (int i = 0; i < values.length; i++) {
         bigger[i] = values[i];
      }
      values = bigger;
   }
}
```

Just as in the constructor

Implementing ArrayList `add` at index

```
// Insert obj at position index
public void add(int index, E element) {
   if (index < 0 || index > size )
      throw new ArrayIndexOutOfBoundsException(index);

   growIfFull();

   for (int from = size-1; from >= index; from--) {
    values[from+1] = values[from];
   }

   values[index] = element;
   size++;
}
```

Can add at end

`from` is where moving data "from"

Implementing ArrayList `add` at index (alternate)

```
// Insert obj at position index
public void add(int index, E element) {
   if (index < 0 || index > size)
      throw new ArrayIndexOutOfBoundsException(index);

   growIfFull();

   for (int to = size ; to > index ; to--) {
    values[ to ] = values[ to-1 ];
   }

   values[index] = element;
   size++;
}
```

`to` is where moving data "to"

Exercises: You do!

Write the following remove method for `MyArrayList` class:

```
public E remove (int index) {









}
```

Exercises: You do!

Write the following remove method for `MyArrayList` class:

```
public int removeEvery (E element) {




}
```