CSC 143 Java

List Implementation using Arrays
Includes Generics

Implementing a List in Java

- Two implementation approaches are most commonly used for simple lists:
  - arrays
  - linked list
- Java Interface List<E>
  - List<E> extends Collection<E>
  - concrete classes ArrayList<E>, LinkedList<E>
  - same methods, different implementations, different advantages

Just an Illusion?

- Key concept: *external view* (the *abstraction* visible to clients) vs. *internal view* (the *implementation*)
- SimpleArrayList<E> may present an illusion to its clients
  - Appears to be a simple, unbounded list of elements
  - Actually may be a complicated internal structure
- The programmer as illusionist...

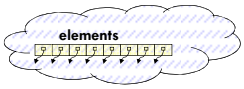
- This is what abstraction is all about

Using an Array to Implement a List

Review Java Arrays…

- Idea: store the list elements in an array instance variable

```
// Simple version of ArrayList for lecture example
public class SimpleArrayList<E> implements List<E> {
    /** variable to hold all elements of the list*/
    private E[ ] elements;
    …
}
```



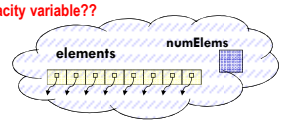
- Issues:
  - How big to make the array?
  - How to use E?
  - Algorithms for adding and deleting elements (add and remove methods)
  - Later: performance analysis of the algorithms

Space Management: Size vs. Capacity

- Idea: allocate space in the array,
  - possibly more than is actually needed at a given time
- Definitions
  - *size*: the number of elements currently in the list, from the client's view
  - *capacity*: the length of the array (the maximum size)
  - invariant: 0 <= *size* <= *capacity*
- When list object created, create an array of some initial maximum capacity
  - What happens if we try to add more elements than the initial capacity? We'll get to that…

List Representation

```
public class SimpleArrayList<E> implements List<E> {
    // instance variables
    private E[ ] elements;     // elements stored in elements[0..numElems-1]
    private int numElems;      // # of elements currently in the list
    // capacity ?? Why no capacity variable??
```




```
    // default capacity
    private static final int defaultCapacity = 10;
    …
}
```

Constructors

- We'll provide two constructors: one where user specifies initial capacity, the other that uses a default initial capacity

```
/** Construct new list with specified capacity */
public SimpleArrayList(int capacity) {
    [                                        ]
}
/** Construct new list with default capacity */
public SimpleArrayList( ) {
    this(defaultCapacity);
}
```

- Review: this(…)
- How about **size()** and **isEmpty()** ?

clear(), get()

void clear()

- Logically, all we need to do is set numElems to 0
  Why?
- It's good practice to null out all of the object references in the list.

E get(int pos)

- We want to handle out-of-bounds argument.
  What is out of bounds? How to handle?
- Otherwise, return the requested object.

indexOf, contains

int indexOf(Object o)

- Sequential search for first "equal" object
- Return first location of object o if found, otherwise return –1
- == or equals()?
- What if we had allowed null items in our list?

boolean contains(Object o)

- return true if this list contains object o, otherwise false
- == or equals()

Interlude: Validate Position

- We've written the code to validate the position and throw an exception twice now – suggests refactoring that code into a separate method

```
/* Validate that a position references an actual element in the list
 * pos: Position to check
 * throws IndexOutOfBoundsException if position invalid, otherwise returns silently */
private void checkPosition(int pos) {
    [                                        ]
}
```


remove at position

Trick here is that we need to compact the array after removing something in the middle

```
/** Remove the object at position pos from this list.  Return removed element.
 *  precondition: 0 <= pos < size() or exception thrown  */
public E remove(int pos) {
    [                                        ]
    return removedElem;
}
```

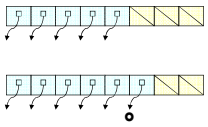

remove Object

```
/** Remove the first occurrence of object o from this list, if present.
 *  @return true if list altered, false if not */
public boolean remove(Object o) {
    [                                        ]
}
```

Method add (simple version)

Assuming there is unused capacity …

```
/** Add object o to the end of this list
 * precondition: o is non-null
  * @return true if list altered by this operation */
public boolean add(E o) {
    if ( o == null )
    if (numElems < elements.length) {
    } else {
       wait for later slide
    }
    return true;
}
```



Dynamic Allocation

- This version of add does not handle the case when adding an object to a list with no spare capacity

Problem: Java arrays are fixed size – can't grow or shrink

Strategy: If the array is already full, allocate a new array with enough capacity, copy the old array, and replace the old array with the new one. Known as **dynamic allocation**

- Question: How big should the new array be?
- Answer:

ensureExtraCapacity

```
/** Ensure that elements[ ] has at least extraCapacity free space,
    growing elements[ ] if needed */
private void ensureExtraCapacity(int extraCapacity) {

}
```

- Pre- and Post- conditions?
- How would we complete the add(Object o) method?

Exercise: add at position

```
/** Add object o at position pos in this list.
  *  precondition: 0 <= pos <= size() or exception thrown */
public void add(int pos, E o) {

}
```


Method iterator()

- Collection interface specifies a method iterator( ) that returns a suitable Iterator<E> for objects of that class
  - Core interface: boolean hasNext( ), E next( )
  - also support remove( ) [left as an exercise]
- Idea: Iterator object holds
  - a reference to the list it is traversing, and
  - the current position in that list

Method iterator

- In class SimpleArrayList

```
/** Return a suitable iterator for this list */
public Iterator<E> iterator( ) {
    return new SimpleListIterator<E>(this );
}
```

- Where do we define SimpleListIterator?
  - Separate top-level, public class?
  - Declare SimpleListIterator as a nested inner class inside class SimpleArrayList?

Class SimpleListIterator (1)

- Inside class SimpleArrayList. It is a *member* of SimpleArrayList

```
// Iterator class for ArrayLists.
private class SimpleListIterator<E> implements Iterator<E> {
    // instance variables
     private SimpleArrayList<E> list; // the List the iterator is traversing
     private int nextItemPos;          // position of next element not yet visited
                                       //  by this iterator
     // invariant: 0 <= nextItemPos <= list.size()

    /** construct iterator object */
    public SimpleArrayListIterator(SimpleArrayList<E> l) {
          nextItemPos = 0;
           list = l;
    }
```


Class SimpleListIterator (2)

```
/* return true if more objects remain in this iteration */
public boolean hasNext( ) {

}
/* return next item in this iteration and advance.
 * @throws NoSuchElementException if iteration has no more elements */
public E next( ) {

}
public void remove(){ throw new UnsupportedOperationException(); }
```


Getting a copy

- Worst case: items field of the new SimpleArrayList copy is bound to the original array -- draw the picture
  - known as shallow copy
  - States can get out of whack really fast!
- Best case: get an entirely unique array and entirely unique elements -- draw the picture
  - Known as deep - copy
- The best we can do: somewhere in the middle. Unique array, but references (elements) of the 2 arrays are bound to the same objects -- draw the picture. Why is this the best we can do?

Copy Constructor

Add a constructor that is passed in a Collection and uses its data to create a new SimpleArrayList

```
public SimpleArrayList ( Collection<? Extends E> c) {

}
```


Summary

- SimpleArrayList presents an illusion to its clients
  - Appears to be a simple, unbounded list of elements
  - Actually a more complicated array-based implementation

- Key implementation ideas:
  - capacity vs. size/numElems
  - sliding elements to implement (inserting) add and remove
  - growing to increase capacity when needed

    growing is transparent to client
- Caution: Frequent sliding and growing is likely to be expensive….