

---

## CSE 143 Java

### List Implementation Using Arrays

Reading: Ch. 13

2/7/2005

(c) 2001-5, University of Washington

14-1

---

## Implementing a List in Java

- Two implementation approaches are most commonly used for simple lists:
  - Arrays
  - Linked list
- Java Interface **List**
  - concrete classes **ArrayList**, **LinkedList**
  - same methods, different internals
  - **List** in turn extends (implements) **Collection**
- Our current activities:
  - Lectures on list implementations, in detail
    - **SimpleArrayList** is a class we'll develop as an example
  - Projects in which lists are used

2/7/2005

(c) 2001-5, University of Washington

14-2

---

## List Interface (review)

```
int size()
boolean isEmpty()
boolean add(Object obj)
boolean addAll(Collection other)
void clear()
Object get(int pos)
boolean set(int pos, Object obj)
int indexOf(Object obj)
boolean contains(Object obj)
Object remove(int pos)
boolean remove(Object obj)
boolean add(int pos, Object obj)
Iterator iterator()
```

2/7/2005

(c) 2001-5, University of Washington

14-3

---

## Just an Illusion?

- Key concept: *external view* (the *abstraction* visible to clients) vs. *internal view* (the *implementation*)
- **SimpleArrayList** may present an illusion to its clients
  - Appears to be a simple, unbounded list of items
- Actually may be a (more or less) complicated internal structure
- The programmer as illusionist...
- This is what abstraction is all about



2/7/2005

(c) 2001-5, University of Washington

14-4

## Java Arrays (Review)

- Key difference from other languages: declaring an array doesn't create it – it must be allocated with new

```
int[] numbers;  
numbers = new int[42]; // creates numbers[0]..numbers[41]
```

or

```
int[] numbers = new int[42];
```

- Capacity is fixed when array is allocated
- Element access: `arrayname[position]`
- Every array object can report how many items it can hold

```
int capacity = numbers.length
```

2/7/2005

(c) 2001-5, University of Washington

14-5

## Using an Array to Implement a List

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

```
// Simple version of ArrayList for CSE143 lecture example
```

```
public class SimpleArrayList implements List {
```

```
    /** variable to hold all items of the list*/
```

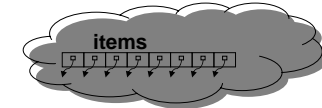
```
    private Object[] items;
```

```
    ...
```

```
}
```

- Issues:

- How big to make the array?
- Why make the array of type Object[]? Pros, cons?
- Algorithms for adding and deleting items (add and remove methods)
- Later: performance analysis of the algorithms



2/7/2005

(c) 2001-5, University of Washington

14-6

## Space Management: Size vs. Capacity

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

2/7/2005

(c) 2001-5, University of Washington

14-7

## List Representation

```
public class SimpleArrayList implements List {
```

```
    // instance variables
```

```
    private Object[] items; // items stored in items[0..size-1]
```

```
    private int size; // size: # of items currently in the list
```

```
    // capacity ?? Why no capacity variable??
```

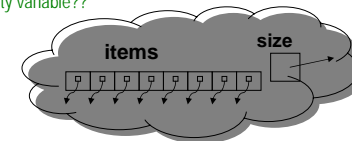
```
    // default capacity
```

```
    private static final int defaultCapacity = 10;
```

```
    ...
```

```
}
```

Review: what does "static final" mean?



2/7/2005

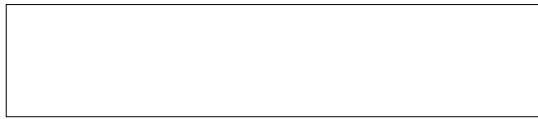
(c) 2001-5, University of Washington

14-8

## Constructors

- We'll provide two constructors:

```
/** Construct new list with specified capacity */  
public SimpleArrayList(int capacity) {
```



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

- Review: `this(...)`

means what? can be used where? why do we want it here?

2/7/2005

(c) 2001-5, University of Washington

14-9

## size, isEmpty: Code

- size:

```
/** Return size of this list */
```

```
public int size() {
```



```
}
```

- isEmpty:

```
/** Return whether the list is empty (has no items) */
```

```
public boolean isEmpty() {
```

```
    return size() == 0;        // OR return size == 0;
```

```
}
```

- Each choice has pros and cons: what are they?

2/7/2005

(c) 2001-5, University of Washington

14-10

## Method add: simple version

- Assuming there is unused capacity ...

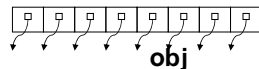
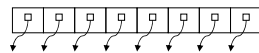
```
/** Add object obj to the end of this list.
```

```
@return true iff the object was added successfully.
```

```
This implementation always returns true. */
```

```
public boolean add(Object obj) {
```

?



```
    return true;
```

```
}
```

- addAll(array or list) left as an exercise – try it at home!

2/7/2005

(c) 2001-5, University of Washington

14-11

## Method add: simple version

- Assuming there is unused capacity ...

```
/** Add object obj to the end of this list
```

```
@return true, since list is always changed by an add */
```

```
public boolean add(Object obj) {
```

```
    if (size < items.length) {
```



```
    } else {
```

```
        // Already full – what can we do here? here's a temporary measure....
```

```
        throw new RuntimeException(
```

```
            "no room – automatic expansion not implemented yet!");
```

```
    }
```

```
    return true;
```

```
}
```

2/7/2005

(c) 2001-5, University of Washington

14-12

## *clear*

- Logically, all we need to do is set `size = 0`
- But it's good practice to null out all of the object references in the list.

- Why?

```
/** Empty this list */  
public void clear() {
```



```
}
```

2/7/2005

(c) 2001-5, University of Washington

14-13

## *Method get*

```
/** Return object at position pos of this list
```

```
* The list is unchanged
```

```
*/
```

```
public Object get(int pos) {  
    return items[pos];  
}
```

- Anything wrong with this?

Hint: what are the preconditions?

2/7/2005

(c) 2001-5, University of Washington

14-14

## *A Better get Implementation*

- We want to catch out-of-bounds arguments, including ones that reference unused parts of array items

```
/** Return object at position pos of this list.  
 * 0 <= pos < size(), or IndexOutOfBoundsException is thrown */  
public Object get(int pos) {
```



```
    return items[pos];
```

```
}
```



- Question: is a "throws" clause required?
- Exercise: write out the preconditions more fully
- Exercise: specify and implement the `set` method

2/7/2005

(c) 2001-5, University of Washington

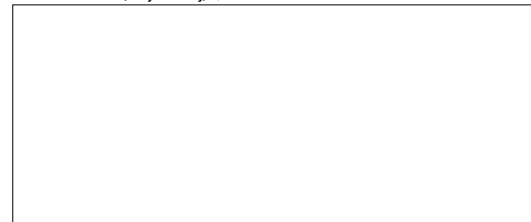
14-15

## *Method indexOf*

- Sequential search for first "equal" object

```
/** return first location of object obj in this list if found, otherwise return -1 */
```

```
public int indexOf(Object obj) {
```



```
}
```

- Exercise: write postconditions

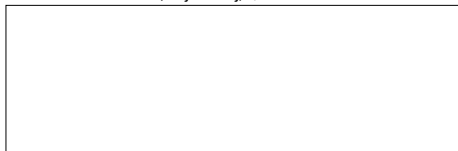
2/7/2005

(c) 2001-5, University of Washington

14-16

## Method *contains*

```
/** return true if this list contains object obj, otherwise false */  
public boolean contains(Object obj) {
```



```
}
```

- Do we need a search loop here? Tradeoffs?
- Exercise: define "this list contains object obj" more rigorously

2/7/2005

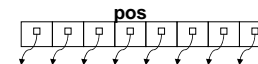
(c) 2001-5, University of Washington

14-17

## *remove(pos)*: Specification

```
/** Remove the object at position pos from this list. Return the removed element.  
0 <= pos < size(), or IndexOutOfBoundsException is thrown */
```

```
public Object remove(int pos) {  
...  
return removedElem;  
}
```



- Postconditions: a bit more complicated this time...
  - Try writing them out!
- Key observation for implementation:
  - we need to compact the array after removing something in the middle; slide all later items left one position

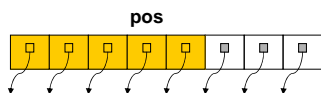
2/7/2005

(c) 2001-5, University of Washington

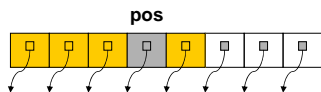
14-18

## Array Before and After *remove*

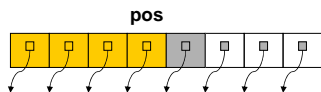
- Before



- After – Wrong!



- After – Right!



2/7/2005

(c) 2001-5, University of Washington

14-19

## *remove(pos)*

```
/** Remove the object at position pos from this list. Return the removed element.  
0 <= pos < size(), or IndexOutOfBoundsException is thrown */
```

```
public Object remove(int pos) {
```



```
}
```


2/7/2005

(c) 2001-5, University of Washington

14-20

## 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
 * @param pos Position to check
 * @throws IndexOutOfBoundsException if position not valid, otherwise returns silently */
private void checkPosition(int pos) {
    
}
```

- Question: why **private**?

2/7/2005

(c) 2001-5, University of Washington

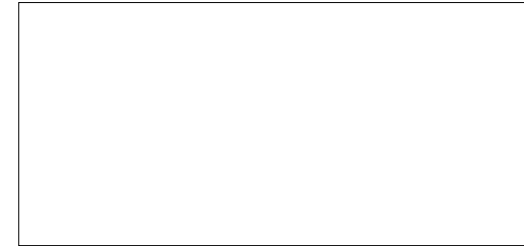
14-21

## *remove*(Object)

```
/** Remove the first occurrence of object obj from this list, if present.
```

```
 @return true if list altered, false if not */
```

```
 public boolean remove(Object obj) {
```



```
 }
```

- Pre- and postconditions are not quite the same as **remove(pos)**

2/7/2005

(c) 2001-5, University of Washington

14-22

## *add* Object at position

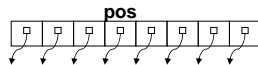
```
/** Add object obj at position pos in this list. List changes, so return true
```

```
 0 <= pos < size(), or IndexOutOfBoundsException is thrown */
```

```
 public boolean add(int pos, Object obj) {
```

```
 ...
```

- Key implementation idea:
  - we need to make space in the middle; slide all later items right one position
- Pre- and postconditions?



2/7/2005

(c) 2001-5, University of Washington

14-23

## *add*(pos, obj): Code

```
/** Add object obj at position pos in this list. List changes, so return true
```

```
 0 <= pos < size(), or IndexOutOfBoundsException is thrown */
```

```
 public boolean add(int pos, Object obj) {
```

```
     checkPosition(pos);
```

```
     if (size >= items.length) {
```

```
         // out of room – for now, ...
```

```
         throw new RuntimeException(
```

```
             "no room – automatic expansion not implemented yet");
```

```
     }
```

```
     ... continued on next slide ...
```

2/7/2005

(c) 2001-5, University of Washington

14-24

## *add(pos, obj)* (continued)

```
...
// preconditions have been met
// first create a space
for (int k = size - 1; k >= pos; k --) { // must count down!
    items[k+1] = items[k]; // slide k'th element right by one index
}
size ++;

// now store object in the space opened up
items[pos] = obj;
return true;
}
```

2/7/2005

(c) 2001-5, University of Washington

14-25

## add Revisited – Dynamic Allocation

- Our original version of `add` checked for the case when adding an object to a list with no spare capacity
  - But did not handle it gracefully: threw an exception
- Better handling: "grow" the array
- Problem: Java arrays are fixed size – can't grow or shrink
- Solution: Make a new array of needed capacity, copy contents of old array to new, then add new item
- This is *dynamic allocation*

2/7/2005

(c) 2001-5, University of Washington

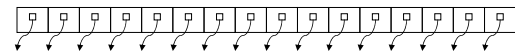
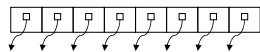
14-26

## Dynamic Allocation Algorithm

### Algorithm

1. **allocate** a new array with larger capacity,
2. **copy** the items from the old array to the new array, and
3. **replace** the old array with the new one

i.e., make the array name refer to the new array



- Issue: How big should the new array be?

2/7/2005

(c) 2001-5, University of Washington

14-27

## Method *add* with Dynamic Allocation

- This implementation has the dynamic allocation hidden away...

```
/** Add object obj to the end of this list
 * @return true, since list is always changed by an add */
public boolean add(Object obj) {
    ensureExtraCapacity(1);
    items[size] = obj;
    size ++;
    return true;
}

/** Ensure that items has at least extraCapacity free space,
 * growing items if needed */
private void ensureExtraCapacity(int extraCapacity) {
    ... magic here ...
}
```

2/7/2005

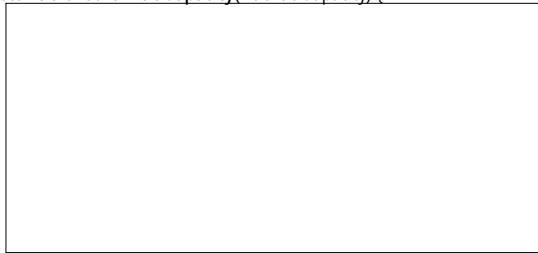
(c) 2001-5, University of Washington

14-28

## *ensureExtraCapacity*

```
/** Ensure that items[] has at least extraCapacity free space,  
growing items[] if needed */
```

```
private void ensureExtraCapacity(int extraCapacity) {
```



```
}
```

- Pre- and Post- conditions?

2/7/2005

(c) 2001-5, University of Washington

14-29

## How Much Extra Capacity?

- **Observation: Growing the array is expensive**
  - “new” is relatively expensive
  - Have to copy all the old entries
- If we increase the capacity by 1 or 2 at a time –
  - Every addition to the list is expensive
- **Common heuristic: when the list fills up, double the capacity when adding a new element**
  - Makes average cost of adding a new element essentially the same as before
  - More about this when we get to efficiency (complexity theory)

2/7/2005

(c) 2001-5, University of Washington

14-30

## Method *iterator*

- Collection interface specifies a method *iterator()* that returns a suitable Iterator for objects of that class
  - Key Iterator methods: boolean hasNext(), Object next()
  - Method remove() is optional for Iterator in general, but expected to be implemented for lists. [left as an exercise]
- **Idea: Iterator object holds...**
  - a reference to the list it is traversing and
  - the current position in that list.
- Can be used for any List, not just ArrayList!
- Except for remove(), iterator operations should never modify the underlying list

2/7/2005

(c) 2001-5, University of Washington

14-31

## Method *iterator*

- In class SimpleArrayList

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

2/7/2005

(c) 2001-5, University of Washington

14-32



## Class SimpleListIterator (1)

```
/** Iterator helper class for lists */
class SimpleListIterator implements Iterator {
    // instance variables
    private List list;           // the list we are traversing
    private int nextItemPos;     // position of next element to visit (if any left)
    // invariant: 0 <= nextItemPos <= list.size()

    /** construct iterator object */
    public SimpleListIterator(List list) {
        list = list;
        nextItemPos = 0;
    }

    ...
}
```

2/7/2005

(c) 2001-5, University of Washington

14-33

## Class SimpleListIterator (2)

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

/** return next item in this iteration and advance.
Note: changes the state of the Iterator but not of the List
@throws NoSuchElementException if iteration has no more items */
public Object next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    [redacted]
}
}
```

2/7/2005

(c) 2001-5, University of Washington

14-34

## Design Questions



- Why create a separate Iterator object?
- Couldn't the list itself have..
  - ...operations for iteration?  
hasNext()  
next()  
reset() //start iterating again from the beginning
  - ...private instance variable for nextPos?
- Would it have been better to implement the iterator as a nested class inside the simple list class?
  - Yes, probably

2/7/2005

(c) 2001-5, University of Washington

14-35

## Summary

- SimpleArrayList presents an illusion to its clients
  - Appears to be a simple, unbounded list of items
  - Actually a more complicated array-based implementation 
- Key implementation ideas:
  - capacity vs. size
  - sliding items to implement (inserting) add and remove
  - growing to increase capacity when needed  
growing should be transparent to client
- Caution: Frequent sliding and growing is likely to be expensive.... 

2/7/2005

(c) 2001-5, University of Washington

14-36