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 gruesome detail
    SimpleArrayList is a class we’ll develop as an example
  - Projects in which lists are used

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()

Java Arrays (Review)

- Key difference from other languages: declaring an array doesn’t create it – it must be allocated with new
  int[] numbers;  // creates numbers[0], numbers[1], numbers[2],...
  or
  int[] numbers = new int[42];
- Size is fixed when array is allocated
- Element access: arrayname[position]
- Every array object can report how many items it contains
  int capacity = numbers.length

Using an Array to Implement a List

- Idea: store the list contents in an array instance variable
  ```java
  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
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 <= size <= 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...

List Representation

public class SimpleArrayList implements List {

    // instance variables
    private Object[] items; // items stored in items[0..numItems-1]
    private int numItems; // size: # of items currently in the list

    // default capacity
    private static final int defaultCapacity = 10;

    ...}

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?

Method 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 numItems == 0;

  }

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

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. */
  public boolean add(Object obj) {

    if (numItems < items.length) {

    } else {

        // Already full – what can we do here? here's a temporary measure....
        throw new RuntimeException("list capacity exceeded");

    }

    return true;

  }

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

- Logically, all we need to do is set `numItems = 0`
- But it's good practice to null out all of the object references in the list.
  
  ```java
  public void clear() {
  }
  ```

**Method get**

/** Return object at position pos of this list
 * The list is unchanged
 */
```java
public Object get(int pos) {
    return items[pos];
}
```

- Why?
- Anything wrong with this?
  
  Hint: what are the preconditions?

**A Better get Implementation**

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

**Method indexOf**

- Sequential search for first "equal" object
  
  ```java
  public int indexOf(Object obj) {
  }
  ```

- Exercise: write postconditions

**Method contains**

- Return true if this list contains object obj, otherwise false
  
  ```java
  public boolean contains(Object obj) {
  }
  ```

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

**remove(pos): Specification**

/** Remove the object at position pos from this list. Return the removed element.
 * 0 <= pos < size(), or IndexOutOfBoundsException is thrown
 */
```java
public Object remove(int pos) {
    return removedElem;
}
```

- Postconditions: quite 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
Array Before and After remove

- Before
  ![Array Before](image)
- After – Wrong!
  ![Array After Wrong](image)
- After – Right!
  ![Array After Right](image)

remove(pos)

- Remove the object at position pos from this list. Return the removed element.
  - \(0 \leq pos < \text{size}()\), or IndexOutOfBoundsException is thrown
  - \(\text{public Object \(\text{remove}(\text{int} \ pos)\)}\)

remove(Object)

- Remove the first occurrence of object obj from this list, if present.
- \(\text{@return true if list altered, false if not}\)
- \(\text{public boolean \(\text{remove}(\text{Object} \ \text{obj})\)}\)

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
- \(\text{public void \(\text{checkPosition}(\text{int} \ pos)\)}\)

add(Object at position)

- Add object obj at position pos in this list. List changes, so return true
  - \(0 \leq pos < \text{size}()\), or IndexOutOfBoundsException is thrown
  - \(\text{public boolean \(\text{add}(\text{int} \ pos, \text{Object} \ \text{obj})\)}\)

  - Key implementation idea:
    - we need to make space in the middle; slide all later items right one position
  - Pre- and postconditions?
    - \(\text{pos}\)

add(pos, obj): Code

- Add object obj at position pos in this list. List changes, so return true
  - \(0 \leq pos < \text{size}()\), or IndexOutOfBoundsException is thrown
  - \(\text{public boolean \(\text{add}(\text{int} \ pos, \text{Object} \ \text{obj})\)}\)
    - \(\text{checkPosition}(\text{pos})\)
    - \(\text{if (numItems >= items.length)}\)
      - \(\text{throw new RuntimeException("list capacity exceeded")}\)
    - ... continued on next slide ...
add(pos, obj) (continued)

// preconditions have been met
// first create a space
for (int k = numItems - 1; k >= pos; k --) {    // must count down!
    items[k+1] = items[k]; // slide k'th element right by one index
}    
numItems ++;
// now store object in the space opened up
items[pos] = obj;
return true;

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 size & copy
    contents of old array to new, then add
  • This is dynamic allocation

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
               3 5 7 9 11
             /  /  /  /  /
               1 3 5 7 9
   • Issue: How big should the new array be?

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

ensureExtraCapacity

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

Pre- and Post- conditions?
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

Class SimpleListIterator (1)

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

Class SimpleListIterator (2)

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

**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();
  }
}

Design Questions

- Why create a separate Iterator object?
- Couldn’t the list itself have...
  - ...operations for iteration?
    - hasNext()
    - next()
  - ...private instance variable for nextPos?

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/numItems
  - sliding items 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...