Working with ArrayList

CS187 Data Structures
February 3, 2010
Chris Vigorito
ArrayList

• Implementation of a dynamically allocated array
• **Class in** `java.util` **package** (must import to use)
• **Implements** the `List` interface
• **Standard list operations** - implementation hidden
List Interface

• Abstract specification for an indexed list

• Extends more general Collection interface

• Describes methods you must write to implement List

• Includes standard list operations:
  • insertion - add()
  • deletion - remove()
  • search - contains(), indexOf()
List Interface

- Other useful methods:
  - `get(int index)` - returns object at position `index` in the list
  - `set(int index, Object o)` - assigns object `o` to be the element at position `index`
  - `size()` - returns the number of elements in the list (not the capacity, which is hidden)
List Interface

- `isEMPTY()` - returns `true` if list contains no elements, `false` otherwise
- `clear()` - removes all elements from list
- ...and more - check out the API docs
ArrayList

- Wrapper class for arrays that implements List
- Actual array is private, accessed through public methods
- Capacity can be set via constructor
- Default initial capacity is 10
Capacity

• Dynamically allocated - grows as elements are added

• Not usually something you need to worry about, but you do have some control over it:
  - Constructor sets initial capacity
  - `trimToSize()` forces capacity to match number of elements
  - `ensureCapacity(int c)` makes sure the capacity is at least c
Capacity

- If you don’t interfere, the array is resized when you enter new elements
- Resized so that insertion has constant amortized cost (long-term average)
- $N$ insertions take $O(N)$ time
- Any given insertion may take time linear in the number of elements, though
Time Complexity

• Insertion - constant \(O(1)\) amortized
• Deletion - linear \(O(n)\)
• Search - linear \(O(n)\)
• Indexing (get/set) - constant \(O(1)\)
Searching **ArrayList**

- `contains(Object o)` - returns **true** if `o` is contained in array, **false** otherwise
- `indexOf(Object o)` - returns the index of `o` in the array, `-1` if `o` is not found
- Both are $O(n)$ operations
Comparisons with Collections

• Comparisons are made using the `equals()` method

• Must be overridden if you are writing a class

  • If you don’t, then just compares references (same as `==`)

  • If you do, must also override `hashCode()` appropriately (equal objects must have equal hash codes)
Generics

• Since Java 5 you can specify the type of objects you want a list/collection to hold

• You specify the element type when declaring and initializing using $<E>$:

  ArrayList<Integer> ints;
  ints = new ArrayList<Integer>();

• Type must be Object or extend Object (no primitive types)

• Default is Object if unspecified
Generics Examples

- Provide a way to ensure type safety
- What happens here?

```java
ArrayList doubles = new ArrayList();
doubles.add(new Double(5.0));
Double d = new Double(doubles.get(0));
```

- Compiler error - Double constructor requires type double or Double, not Object
Generics Examples

• What about here?

```java
ArrayList doubles = new ArrayList();
doubles.add(new Double(5.0));
Integer i = new Integer((Integer) doubles.get(0));
```

• Runtime error - `ClassCastException`
Generics Examples

• And with generics?

```java
ArrayList<Double> doubles = new ArrayList<Double>();
doubles.add(new Double(5.0));
Integer i = new Integer((Integer) doubles.get(0));
```

• Compiler error - “inconvertible types”
Auto(un)boxing

- Very convenient - can save a lot of typing when using collections
- Automatically converts between primitive types and associated wrapper classes
- Under the hood, but you should be aware that it’s going on
Auto(un)boxing

ArrayList<Integer> ints =
    new ArrayList<Integer>();
ints.add(new Integer(6));
ints.add(7);
Integer i = ints.get(0);
int j = ints.get(1);
double d = ints.get(1);
Iterators

- Allow you to traverse a list/collection
- Generally used with `while` loops

**Iterator** interface requires three methods:

- `hasNext()` - returns true if there are tokens left to process
- `next()` - returns the next token
- `remove()` - removes (safely) the last token returned by the iterator from the collection
Iterators

ArrayList<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
ints.add(3);
Iterator<Integer> it = ints.iterator();
while (it.hasNext())
    System.out.println(it.next());
Concurrent Modification Exceptions

• If you try to change the contents of a collection while traversing it (e.g., with for each or an iterator), you will cause a ConcurrentModificationException.

• The only way to do this safely is to use an iterator and use the `remove()` method from the iterator object (not the collection itself).

• This means you can’t add to a collection while traversing it.
Concurrent Modification
Exceptions

ArrayList<Integer> ints =
    new ArrayList<Integer>();
ints.add(1);
ints.add(2);
ints.add(3);
for (Integer i : ints)
    if (i.intValue() == 3)
        ints.remove(i);
Concurrent Modification
Exceptions

ArrayList<Integer> ints =
    new ArrayList<Integer>();
ints.add(1);
ints.add(2);
ints.add(3);
Iterator<Integer> it = ints.iterator();
while(it.hasNext())
    if(it.next().intValue() == 3)
        it.remove();
Quiz

- Fill in your name and student ID
- 10 questions (two sides)
- Drop off in front on your way out