CS61B Lecture #7

Reminder:

- Discussion section 114 (3-4 Th) moves to 3102 Etch., starting tomorrow.

Today:

- Java Library Classes for lists.
- Iterators, ListIterators

Abstracting "Listness"

- So far, we've seen fairly primitive types for representing lists of things.
- Arrays:
  - Good: *random access* to items.
  - Bad: hard to expand, insert items, or delete items.
- Linked lists (e.g., IntList):
  - Good: easy to expand, insert, delete.
  - Bad: must access in sequence, pointer manipulation can be tricky.
- Both used to represent same thing (sequence of things), but syntax for using very different,
- So hard to switch from one to the other if you change your mind.

Useful Classes from the Java Library

- Java library has types to represent collections of objects, including
  - *Lists* (sequences) of objects: `ArrayList`, `LinkedList`.
  - *Sets* of objects: `TreeSet`, `HashSet`.
  - *Maps* (dictionaries): `TreeMap`, `HashMap`.
- These types are "in the package `java.util`."
  - *Package = Set of classes and subpackages.*
  - Notation: `java.util.ArrayList`: "The class named `ArrayList` in the (sub)package named `util` in the package named `java`.
- Names of these classes reflect implementations, but they "publicize" very similar *interfaces* to the outside.
- Thus, easy to change from using ArrayList to LinkedList, e.g.

Lists

- The list classes `ArrayList` and `LinkedList` both share many public methods, including:
  - size(), isEmpty(): Number of items, test for 0 items.
  - get(k): Get item #k, where $0 \leq k <$ `size()`.
  - remove(k): Remove item #k.
  - clear(): Make the list empty.
  - set(k, x): Set item #k to x.
  - add(x), add(k,x): Add item to end, or a position k.
  - contains (x): True iff there is an item that equals `x` (according to `.equals` method).
  - indexOf (x): Gives the position ($0 \leq \cdot <$size()) of the first item that `.equals` x, or -1 if there is none.
- Both expand as needed (automatically).
- A few methods specialized to one or the other (e.g. `LinkedList.removeFirst()`.

Example: Read and interleave two lists

```
/** Read the sequence of words on INPUT, and print on
 *  OUTPUT in reverse order. */
static void readAndReverse (Scanner input, PrintStream output) {
    ArrayList<String> L = new ArrayList<String> ();
    while (input.hasNext ())
        L.add (input.next ());
    for (k = L.size ()-1; k >= 0; k -= 1)
        output.printf ("%s ", L.get (k))
}
```

- Not shown: `import java.util.ArrayList;` before class.
- Could also use a `LinkedList<String>`. What problem might there be with that?

Iterators

- **Problem:** Indexing as for arrays (via `.get`) not always best (fastest) way to get items.
- **Problem:** But would like to use same interface (same methods, same text) for `ArrayList` and `LinkedList`.
- Abstraction to the rescue: the library has class called `Iterator`, which acts like a "moving finger" through a collection of objects.

```
static void printList (ArrayList<String> L) {
   System.out.printf ("{%n");
   for (Iterator<String> place = L.iterator (); place.hasNext (); )
       System.out.printf ("   %s%n", place.next ());
}
```

So common, Java 1.5 introduced shorthand:

```
for (String s : L)
  System.out.printf ("   %s%n", s);
```


ListIterator

- Library also has type `ListIterator`
- These have both `.previous()` and `.next()` methods.
- Also allow insertion.
- Look at reversal again:

```
/** Read the sequence of words on INPUT, and print on
 *  OUTPUT in reverse order. */
static void readAndReverse (Scanner input, PrintStream output) {
    ArrayList<String> L = new ArrayList<String> ();
    ListIterator<String> place = L.listIterator ();

    while (input.hasNext ())
      place.add (input.next ());
    while (place.hasPrevious ())
      System.out.printf ("%s ", place.previous ());
}
```


Primitive Types and Wrappers

- ArrayLists and the like can only take elements that are pointers, no `int`s, `double`s, `boolean`s, etc.
- So, Java library contains corresponding *wrapper classes:* `Integer`, `Double`, `Boolean`, etc.—all pointed-to objects
- So, `new Integer(3)` is → 3 . The `intValue()` method retrieves the 3.
- All very tedious, so Java 1.5 converts int ⇔ Integer automatically—*boxes* 3 to make an `Integer`, *unboxes* to get 3 back.
- So we can do things like this:

```
ArrayList<Double> sqrts = new ArrayList<Double>();
while (inp.hasNext ())
   sqrts.add (Math.sqrt (inp.nextDouble ()));
double sum = 0;
for (double x : sqrts)
   sum += x;
```

- Almost painless, but, alas, expensive.