CS61B Lecture #17

Administrative:
- Need alternative test time? Make sure you send me mail.

Topics
- Overview of standard Java Collections classes.
  - Iterators, ListIterators
  - Containers and maps in the abstract
  - Views
- Generic Implementation
- Array vs. linked: tradeoffs
- Sentinels
- Specialized sequences: stacks, queues, deques
- Circular buffering
- Recursion and stacks
- Adapters

Readings: Data Structures, Chapter 2, 3 (for today), and 4 (Friday).

Data Types in the Abstract
- Most of the time, should not worry about implementation of data structures, search, etc.
- What they do for us—their specification—is important.
- Java has several standard types (in java.util) to represent collections of objects
  - Six interfaces:
    * Collection: General collections of items.
    * List: Indexed sequences with duplication
    * Set, SortedSet: Collections without duplication
    * Map, SortedMap: Dictionaries (key $\mapsto$ value)
  - Concrete classes that provide actual instances: LinkedList, ArrayList, HashSet, TreeSet.
  - To make change easier, purists would use the concrete types only for new, interfaces for parameter types, local variables.

Collection Structures in java.util
The Collection Interface

- **Collection interface. Main functions promised:**
  - Membership tests: contains ($\in$), containsAll ($\subseteq$)
  - Other queries: size, isEmpty
  - Retrieval: iterator, toArray
  - Optional modifiers: add, addAll, clear, remove, removeAll (set difference), retainAll (intersect)

- **Design point (a side trip): Optional operations may throw UnsupportedOperationException**

- **An alternative design would have separate interfaces:**
  - interface Collection { contains, containsAll, size, iterator, ... }
  - interface Expandable { add, addAll }
  - interface Shrinkable { remove, removeAll, difference, ... }
  - interface ModifiableCollection extends Collection, Expandable, Shrinkable { }

  You'd soon have lots of interfaces. Perhaps that's why they didn't do it that way.

The List Interface

- **Extends Collection**
- **Intended to represent indexed sequences (generalized arrays)**
- **Adds new methods to those of Collection:**
  - Membership tests: indexOf, lastIndexOf.
  - Retrieval: get($i$), listIterator($i$), sublist($B, E$).
  - Modifiers: add and addAll with additional index to say where to add. Likewise for removal operations. set operation to go with get.

- **Type ListIterator$<$Item$>$ extends Iterator$<$Item$>$:**
  - Adds previous and hasNext.
  - add, remove, and set allow one to iterate through a list, inserting, removing, or changing as you go.
  - **Important Question:** What advantage is there to saying List $L$ rather than LinkedList $L$ or ArrayList $L$?

Views

**New Concept:** A view is an alternative presentation of (interface to) an existing object.

- For example, the sublist method is supposed to yield a "view of" part of an existing list:

  List<String> $L = \text{new ArrayList<String> ();}$
  $L.$add("at"); $L.$add("ax"); ...
  List<String> $SL = L.$sublist$(1,4);$

- Example: after $L.$set$(2, "bag")$, value of $SL.$get$(1)$ is "bag", and after $SL.$set$(1,"bad")$, value of $L.$get$(2)$ is "bad".

- Example: after $SL.$clear()$, $L$ will contain only "at" and "cat".

- Small challenge: "How do they do that?!"

Maps

**A Map is a kind of "modifiable function:"

```java
package java.util;
public interface Map<Key,Value> {
    Value get (Object key); // Value at KEY.
    Object put (Key key, Value value); // Set get(KEY) -- VALUE ...
}
```

```java
Map<String,String> $f =$\text{new TreeMap<String,String> ();}$
$f.$put("Paul", "George"); $f.$put("George", "Martin");
$f.$put("Dana", "John");
// Now $f.$get("Paul").equals("George")
// $f.$get("Dana").equals("John")
// $f.$get("Tom") == null
```
Map Views

public interface Map<Key, Value> { // Continuation
    /* VIEWS */
    Set<Key> keySet();
    Collection<Value> values();
    Set<Map.Entry<Key, Value>> entrySet();
}

Using example from previous slide:

for (Iterator<String> i = f.keySet().iterator(); i.hasNext();)
    i.next() ===> Dana, George, Paul

// or, just:
for (String name : f.keySet())
    name ===> Dana, George, Paul

for (String parent : f.values())
    parent ===> John, Martin, George

f.keySet().remove("Dana"); // Now f.get("Dana") == null

Simple Banking II: Banks

class Bank {
    /* These variables maintain mappings of String -> Account. They keep */
    /* the set of keys (Strings) in "compareTo" order, and the set of */
    /* values (Accounts) is ordered according to the corresponding keys. */
    SortedMap<String, Account> accounts = new TreeMap<String, Account>();
    SortedMap<String, Account> names = new TreeMap<String, Account>();

    void openAccount(String name, int initBalance) {
        Account acc =
            new Account (name, chooseNumber(), initBalance);
        accounts.put(acc.number, acc);
        names.put(name, acc);
    }

    void deposit(String number, int amount) {
        Account acc = accounts.get(number);
        if (acc == null) ERROR(...);
        acc.balance += amount;
    }

    // Likewise for withdraw.
}

Simple Banking I: Accounts

Problem: Want a simple banking system. Can look up accounts by name or number, deposit or withdraw, print.

Account Structure

class Account {
    Account (String name, String number, int init) {
        this.name = name; this.number = number;
        this.balance = init;
    }
    /** Account-holder’s name */
    final String name;
    /** Account number */
    final String number;
    /** Current balance */
    int balance;

    /** Print THIS on STR in some useful format. */
    void print (PrintWriter str) { ... }
}

Banks (continued): Iterating

Printing out Account Data

/** Print out all accounts sorted by number on STR. */
void printByAccount (PrintStream str) {
    // accounts.values () is the set of mapped-to values. Its
    // iterator produces elements in order of the corresponding keys.
    for (Account account : accounts.values ())
        account.print (str);
}

/** Print out all bank accounts sorted by name on STR. */
void printByName (PrintStream str) {
    for (Account account : names.values ())
        account.print (str);
}

A Design Question: What would be an appropriate representation for keeping a record of all transactions (deposits and withdrawals) against each account?
Partial Implementations

- Besides interfaces (like `List`) and concrete types (like `LinkedList`), Java library provides abstract classes such as `AbstractList`.
- Idea is to take advantage of the fact that operations are related to each other.
- Example: once you know how to do `get(k)` and `size()` for an implementation of `List`, you can implement all the other methods needed for a read-only list (and its iterators).
- Now throw in `add(k,x)` and you have all you need for the additional operations of a growable list.
- Add `set(k,x)` and `remove(k)` and you can implement everything else.

Example: The `java.util.AbstractList` helper class

```java
public abstract class AbstractList<Item> implements List<Item> {
    /** Inherited from List */
    public abstract int size();
    public abstract Item get(int k);
    public boolean contains(Object x) {
        for (int i = 0; i < size(); i++) {
            if ((x == null && get(i) == null) ||
                (x != null && x.equals(get(i))))
                return true;
        }
        return false;
    }
    /* OPTIONAL: By default, throw exception; override to do more. */
    void add(int k, Item x) {
        throw new UnsupportedOperationException();
    }
    // Likewise for remove, set
}
```

Example, continued: `AListIterator`

```java
// Continuing abstract class AbstractList<Item>: public Iterator<Item> iterator() { return listIterator(); }
public ListIterator<Item> listIterator() { return new AListIterator(this); }

private static class AListIterator implements ListIterator<Item> {
    AbstractList<Item> myList;
    AListIterator(AbstractList<Item> L) { myList = L; }
    /* Current position in our list. */
    int where = 0;

    public boolean hasNext() { return where < myList.size(); }
    public Item next() { return myList.get(where); where += 1; }
    private void add(Item x) { myList.add(where, x); where += 1; }
    ... previous, remove, set, etc.
    }
    ...
}
```

Example: Using AbstractList

```java
Problem: Was to create a reversed view of an existing List (same elements in reverse order).

public class ReverseList<Item> extends AbstractList<Item> {
    private final List<Item> L;

    public ReverseList(List<Item> L) { this.L = L; }
    public int size() { return L.size(); }
    public Item get(int k) { return L.get(L.size()-k-1); }
    public void add(int k, Item x) {
        L.add(L.size()-k, x); }
    public Item set(int k, Item x) {
        return L.set(L.size()-k-1, x); }
    public Item remove(int k) {
        return L.remove(L.size() - k - 1); }
}
```
Aside: Another way to do AListIterator

It's also possible to make the nested class non-static:

```java
public class AListIterator implements ListIterator<Item> {
    /** Current position in our list. */
    int where = 0;

    public boolean hasNext () { return where < AbstractList.this.size (); }
    public Item next () { where += 1; return AbstractList.this.get (where-1); }
    public void add (Item x) { AbstractList.this.add (where, x); where += 1; }
    ... previous, remove, set, etc.
}
```

- Here, AbstractList.this means "the AbstractList I am attached to" and X.new AListIterator means "create a new AListIterator that is attached to X."
- In this case you can abbreviate this.new as new and can leave off the AbstractList.this parts, since meaning is unambiguous.

What Does a Sublist Look Like?

- Consider SL = L.sublist (3, 5);

Arrays and Links

- Two main ways to represent a sequence: array and linked list
- In Java Library: ArrayList and Vector vs. LinkedList.
- Array:
  - Advantages: compact, fast ($\Theta(1)$) random access (indexing).
  - Disadvantages: insertion, deletion can be slow ($\Theta(N)$)
- Linked list:
  - Advantages: insertion, deletion fast once position found.
  - Disadvantages: space (link overhead), random access slow.

Getting a View: Sublists

Problem: L.sublist(start, end) is a full-blown List that gives a view of part of an existing list. Changes in one must affect the other.

How? Here's part of AbstractList:

```java
private class Sublist extends AbstractList<Item> {
    // NOTE: Error checks not shown
    private int start, end;
    Sublist (int start, int end) { obvious }
    public int size () { return end-start; }
    public Item get (int k) {
        return AbstractList.this.get (start+k);
    }
    public void add (int k, Item x) {
        AbstractList.this.add (start+k, x); end += 1; }
    ...}
```

L: List object

SL: AbstractList.this

start: 3
end: 5
Implementing with Arrays

• Biggest problem using arrays is insertion/deletion in the middle of a list (must shove things over).
• Adding/deleting from ends can be made fast:
  - Double array size to grow; amortized cost constant (Lecture #15).
  - Growth at one end really easy; classical stack implementation:

\[
\text{S.push ("X");} \\
\text{S.push ("Y");} \\
\text{S.push ("Z");}
\]

- To allow growth at either end, use circular buffering:

\[
\begin{array}{c}
\text{add here} \\
\text{F} \\
\text{H} \\
\text{I} \\
\text{G}
\end{array}
\]

- Random access still fast.

Clever trick: Sentinels

• A sentinel is a dummy object containing no useful data except links.
• Used to eliminate special cases and to provide a fixed object to point to in order to access a data structure.
• Avoids special cases (if statements) by ensuring that the first and last item of a list always have (non-null) nodes—possibly sentinels—before and after them:

\[
\text{p.next.prev = p.prev;} \\
\text{p.prev.next = p.next;}
\]

L: axolotl kludge xerophyte

Specialization

• Traditional special cases of general list:
  - Stack: Add and delete from one end (LIFO).
  - Queue: Add at end, delete from front (FIFO).
  - Dequeue: Add or delete at either end.
• All of these easily representable by either array (with circular buffering for queue or deque) or linked list.
• Java has the List types, which can act like any of these (although with non-traditional names for some of the operations).
• Also has java.util.Stack, a subtype of List, which gives traditional names ("push", "pop") to its operations. There is, however, no "stack" interface.
Stacks and Recursion

- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become “push current variables and parameters, set parameters to new values, and loop.”
  - Return becomes “pop to restore variables and parameters.”

```
findExit(start):
  if isExit(start)
    FOUND
  else if (! isCrumb(start))
    leave crumb at start; for each square, x, adjacent to start:
      if legalPlace(x)
        findExit(x)

findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (! isCrumb(start))
      leave crumb at start; for each square, x, adjacent to start (in reverse):
        if legalPlace(x)
          push x on S
```

Call: findExit(0)
Exit: 16

Design Choices: Extension, Delegation, Adaptation

- The standard java.util.Stack type extends Vector:
  ```
  class Stack<Item> extends Vector<Item> { void push (Item x) { add (x); } ... }
  ```

- Could instead have delegated to a field:
  ```
  class ArrayStack<Item> {
    private ArrayList<Item> repl = new ArrayList<Item> ();
    void push (Item x) { repl.add (x); } ...
  }
  ```

- Or, could generalize, and define an adapter: a class used to make objects of one kind behave as another:
  ```
  public class StackAdapter<Item> {
    private List repl;
    /** A stack that uses REPL for its storage. */
    public StackAdapter (List<Item> repl) { this.repl = repl; }
    public void push (Item x) { repl.add (x); } ...
  }
  ```

  class ArrayStack<Item> extends StackAdapter<Item> {
    ArrayStack () { super (new ArrayList<Item> ()); }
  }
```