

CS61B Lecture #18

Administrative:

- Need alternative test time? Make sure you send me mail.
- Lab5 is on-line in the usual place.

Today:

- Maps
- Generic Implementation

Readings for Today: *Data Structures*, Chapter 3.

Readings for Next Topic: *DS(IJ)*, Chapter 4, Goodrich and Tamassia, Chapter 4.

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) { ... }
}
```

Simple Banking II: Banks

```
class Bank {
    SortedMap accounts = new TreeMap ();
    SortedMap names = new TreeMap ();

    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 = (Account) accounts.get (number);
        if (acc == null) ERROR(...);
        acc.balance += amount;
    }

    // Likewise for withdraw.
    // Continues...
}
```

Simple Banking III: Iterating

Printing out Account Data

```
/** Print out all accounts sorted by number on STR. */
void printByAccount (PrintStream str) {
    for (Iterator i = accounts.values ().iterator ());
        i.hasNext ())
        ((Account) i.next ()).print (str);
}
```

```
/** Print out all bank acconts sorted by name on STR. */
void printByName (PrintStream str) {
    for (Iterator i = names.values ().iterator ());
        i.hasNext ())
        ((Account) i.next ()).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: AbstractList

- java.util.AbstractList "helper class"

```
public abstract class AbstractList implements List {
    /** Inherited from List */
    // public abstract int size ();
    // public abstract Object get (int k);
    public boolean contains (Object x) {
        for (int i = 0; i < size (); i += 1) {
            if ((x == null && get (i) == null) ||
                (x != null && x.equals (get (i))))
                return true;
        }
        return false;
    }
    public Iterator iterator () { return listIterator (); }
    public ListIterator listIterator () { return new AListIterator (); }
    /* OPTIONAL (By default, throw exception) */
    Object void add (int k, Object x) {
        throw new UnsupportedOperationException ();
    } ...
}
```

Example, continued: AListIterator

```
public abstract class AbstractList implements List {
    // NOTE: No error checking shown
    ...
    private class AListIterator implements ListIterator {
        /** Current position in our list. */
        int where = 0;
        public boolean hasNext () {
            return where < AbstractList.this.size ();
            // or return where < size (); for short
        }
        public Object next () {
            where += 1;
            return AbstractList.this.get (where-1);
        }
        public void add (Object x) {
            AbstractList.this.add (where, x);
            where += 1;
        }
        ...
    }
    ...
}
```

Example: Using AbstractList

Problem: Want to create a *reversed view* of an existing List (same elements in reverse order).

```
public class ReverseList extends AbstractList {
    private final List L;

    public ReverseList (List L) { this.L = L; }

    public int size () { return L.size (); }

    public Object get (int k) { return L.get (L.size ()-k-1); }

    public void add (int k, Object x)
        { L.add (L.size ()-k, x); }

    public Object set (int k, Object x)
        { return L.set (L.size ()-k-1, x); }

    public Object remove (int k)
        { return L.remove (L.size () - k - 1); }
}
```

Small Side Trip about Reflection: toArray

- **Types** Class and `reflect.Array` in `java.lang` provide way of talking about Java constructs in Java.

```
/** An array containing the elements of THIS in order. Use A if big
 * enough, otherwise a new array of same dynamic type. */
public Object[] toArray (Object[] A) {
    Object[] result;
    if (A.length >= size ())
        result = A;
    else {
        Class eltType =
            A.getClass ().getComponentType ();
        result =
            (Object[]) Array.newInstance (eltType, size ());
    }
    Iterator i = iterator ();
    for (int j = 0; j < A.length; j += 1)
        if (i.hasNext ())
            result[j] = i.next ();
        else
            result[j] = null;
    return result;
}
```

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`:

```
List sublist (int start, int end) {
    return new Sublist (start, end);
}

private class Sublist extends AbstractList {
    // NOTE: Error checks not shown
    private int start, end;
    Sublist (int start, int end) { obvious }

    public int size () { return end-start; }

    public int get (int k)
        { return AbstractList.this.get (start+k); }

    public void add (int k, Object x) {
        { AbstractList.this.add (start+k, x); end += 1; }
        ...
    }
}
```

What Does a Sublist Look Like?

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

