

CS61B Lecture #17

Administrative:

- There are no Lecture #16 notes.
- Need alternative test time? Make sure you send me mail next week.
- Reminder: preliminary grading run early Tuesday (i.e., after midnight Monday).
- Reminder: Use bug-submit, *not* the newsgroup, for bugs in your code. Submit *all* your files; don't try to trim "unrelated" stuff for our benefit.

Today:

- Maps
- Generic Implementation

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

Readings for Next Topic: *Data Structures*, Chapter 4.

Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 1

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) { ... }
}
```

Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 2

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.
}
```

Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 3

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?

Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 4

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.

Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 5

Example: The java.util.AbstractList helper class

```
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 += 1) {
            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
}
```

Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 6

Example, continued: AListIterator

```
// 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 () { where += 1; return myList.get (where-1); }
    public void add (Item x) { myList.add (where, x); where += 1; }
    ... previous, remove, set, etc.
}
...
```

Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 7

Example: Using AbstractList

Problem: Want 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); }
}
```

Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 8

Aside: Another way to do AListIterator

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

```
public Iterator<Item> iterator () { return listIterator (); }
public ListIterator<Item> listIterator () { return this.new AListIterator (); }

private 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.

Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 9

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<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); }

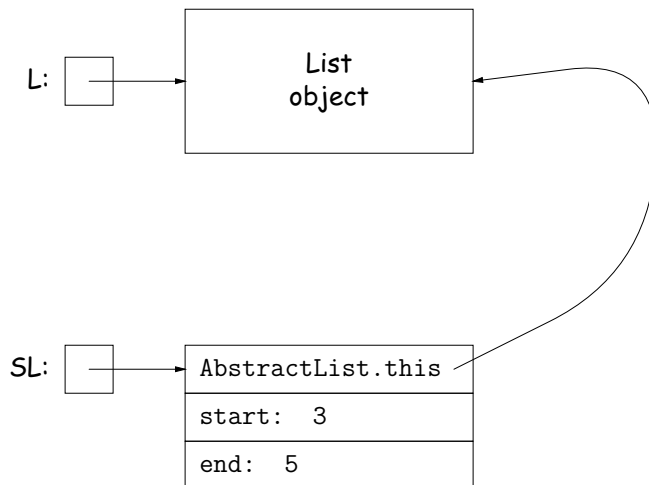
    public void add (int k, Item x) {
        { AbstractList.this.add (start+k, x); end += 1; }
        ...
    }
}
```

Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 10

What Does a Sublist Look Like?

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



Last modified: Mon Oct 11 14:21:16 2004

CS61B: Lecture #18 11