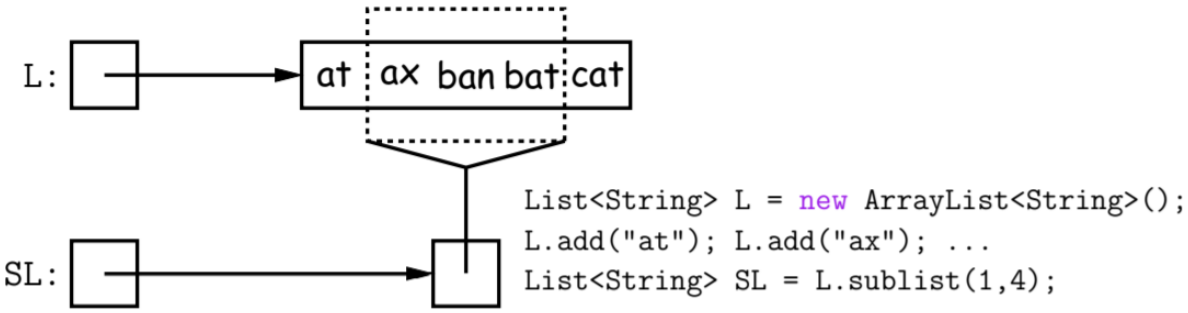
CS61B Lecture 18

Wednesday, March 4, 2020

Today, we will be discussing various assorted topics of Java.

Views

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:



This is different from Python, which simply gives you a copy of the list. Any changes made to either list, will be reflected in the other.

While we will not discuss its implementation today, it may be a good idea for you to think about.

Maps

A map is a kind of modifiable function:

```
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
    ...
}
```

```
Map<String,String> f = 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 of Maps */
    /** The set of all keys. */
    Set<Key> keySet();
    /** The multiset of all values that can be returned by get.
    * (A multiset is a collection that may have duplicwates). */
    Collection<Value> values();
    /** The set of all(key, value) pairs */
    Set<Map.Entry<Key,Value>> entrySet();
}
```

Using the example from the previous slides:

```
Map<String,String> f = new TreeMap<String,String>();
f.put("Paul", "George"); f.put("George", "Martin");
f.put("Dana", "John");
```

We can take various views of `f`:

```
for (Iterator<String> i = f.keySet().iterator(); i.hasNext();)
   i.next()  ===>  Dana, George, Paul
// or, more succinctly:
for (String name : f.keySet())
   name  ===>  Dana, George, Paul

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

for (Map.Entry<String,String> pair : f.entrySet())
   pair   ===>   (Dana,John), (George,Martin), (Paul,George)

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

Banking

We want a simple banking system that can look up accounts by name or number, deposit or withdraw, print. Here's some structure of how we intend to achieve that:

Account

```
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(PrintStream str) { ... }
}
```

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.
```

Printing

```
/** 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`), the Java library also provides abstract classes such as `AbstractList`. The idea is to take advantage of the fact that operations are related to each other.

For 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`

```
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: Throws exception; override to do more. */
    void add (int k, Item x)
        throw new UnsupportedOperationException();
    }
}
// Likewise for remove, set
```

`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.
}
...
```

Notice this is a private class, but we can still access it because they implement an interface that is available publicly, so we can still access its methods. Often, you'll find that when you call iterator classes, you will get an object whose type you cannot mention, but because it is an iterator, we know it has `.next()` and `.hasNext()`.

Another approach to `AListIterator`

It is 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.
}
...
```

Using `AbstractList`

What if we wanted to create a reversed view of an existing `List`, with the same elements but in reverse order. We want any changes to one list to affect the other:

```
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(0-k-1, x));
    }
    public Item remove (int k) { return L.remove(L.size() - k - 1); }
}
```

Sublists

`L.sublist(start, end)` is a List that gives a view of part of an existing list. Changes in one must affect the other. How?

```
// Continuation of class AbstractList. Error checks not shown.
List<Item> sublist(int start, int end) {
    return this.new Sublist(start, end);
}
private class Sublist extends AbstractList<Item> {
    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; }
    ...
}
```