**Simple Banking I: Accounts**

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

**Account Structure**

```java
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**

```java
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;
    }
}
```

**Banks (continued): Iterating**

**Printing out Account Data**

```java
    /** 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 += 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();
    }
    ... previous, remove, set, etc.
}
```

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

Example: Using `AbstractList`

```java
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 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); }
}
```

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

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

```java
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; }
    ...
}
```

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.

Getting a View: Sublists

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

```java
private class Sublist extends AbstractList<Item> {
    // NOTE: Error checks not shown
    private int start, end;

    Sublist (int start, int end) {
        this.start = start;
        this.end = end;
    }

    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; }
    ...
}
```

What Does a Sublist Look Like?

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

```
<table>
<thead>
<tr>
<th>L:</th>
<th>List object</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL:</td>
<td>AbstractList.this</td>
</tr>
<tr>
<td></td>
<td>start: 3</td>
</tr>
<tr>
<td></td>
<td>end: 5</td>
</tr>
</tbody>
</table>
```

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.
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:
    ```java
    S.push ("X");
    S.push ("Y");
    S.push ("Z");
    ```
  - To allow growth at either end, use circular buffering:
    ```java
    F  I  H  G
    add here
    ```
  - 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:
  ```java
  // To delete list node at p:
  p.next.prev = p.prev;
  p.prev.next = p;
  // To add new node N before p:
  N.prev = p.prev; N.next = p;
  p.prev.next = N;
  ```

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 dequeue) 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.”

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

• Could instead have delegated to a field:
  ```java
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:
  ```java
  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); } ... }
  ```
  ```java
class ArrayStack<Item> extends StackAdapter<Item> {
    ArrayStack () { super (new ArrayList<Item> ()); }
  }
  ```