Abstract Data Types

- collection of data + valid operations
- encapsulates data and operations
- hides data and operations that are not needed by clients

Encapsulation v. Info Hiding

- Encapsulation: combines data and operations
- Info Hiding: ability to make attributes/operations inaccessible to clients

Typical operations

- create an instance of the ADT (new)
- add to the data collection of the instance
- remove from the data collection
- inquire about the data collection
Instance of v. ADT

- ADT is the RULE or description (class)
- Instance (object) is an embodiment of the ADT - you can point to memory and say, “here is the object”

Implementation v. ADT

- ADT is an ABSTRACTION - only the WHAT.
- Implementation is the HOW - it has enough detail to be compiled / run
- ADT is implemented using Data Structures

Ex. List

- List is a sequence of elements
- Can be empty
- If not empty, there’s a first element, followed by the rest of the list
- can add, delete, and retrieve elements

List ADT

<table>
<thead>
<tr>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
</tr>
<tr>
<td>createList();</td>
</tr>
<tr>
<td>isEmpty();</td>
</tr>
<tr>
<td>size();</td>
</tr>
<tr>
<td>add(pos, item);</td>
</tr>
<tr>
<td>delete(pos);</td>
</tr>
<tr>
<td>deleteAll();</td>
</tr>
<tr>
<td>get(pos);</td>
</tr>
</tbody>
</table>
List ADT

```java
List aList = new List();
aList.add(1, "a");
aList.add(2, "b");
aList.add(3, "c");
aList.delete(1);
print(aList.length());
```

Specification

```java
/**
* Create an empty list.
* pre: none.
* post: empty list exists.
*/
+createList();

/**
* Determine if list is empty.
* pre: list exists.
* post: you know if list is empty.
* @return boolean true if empty
*/
+isEmpty():boolean {query}
```

```java
/**
* Determine number of items in list.
* pre: list exists.
* post: you know number of items.
* @return integer number of items in list.
*/
+size():integer; {query}
```

```java
/**
* Add elt at position pos in list.
* pre: list exists, 0<pos<=size()+1.
* post: elt is inserted before item
* at index pos. item that was at pos
* before is now at pos+1. same for all
* items at indexes greater than pos.
* @param pos integer index at which to
* insert.
* @param elt itemType item to insert.
*/
+add(in pos:integer, in elt:itemType)
```
**Specification**

```cpp
+delete(in pos:integer)
/**
 * Delete item at position pos in list.
 * pre: list exists, 0<pos<size()+1.
 * post: elt is deleted at index pos.
 * item that was at pos+1 before is now
 * at pos. same for all items at
 * indexes greater than pos. Size() is
 * now one less than before.
 * @param pos integer index at which to
 * delete.
 */
+delete(in pos:integer)
```

**Specification**

```cpp
+deleteAll()
/**
 * Remove all items from list.
 * pre: list exists.
 * post: list is empty.
 */
+deleteAll()
```

**Specification**

```cpp
+get(in pos:integer):itemType; {query}
/**
 * Get the item at position pos in list.
 * pre: list exists, 0<pos<size()+1.
 * post: you have the item.
 * @param pos integer index at which
 * to get item.
 * @return itemType item retrieved.
 */
+get(in pos:integer):itemType; {query}
```

**Using the ADT**

```cpp
/**
 * Print the list.
 * pre: list exists.
 * post: items in List are printed.
 * @param aList List to be printed.
 */
println(in aList:List) {
    for (int i = 1 to aList.size()) {
        print(aList.get(i));
    }
}
```
Using the ADT

Find elt in the list.

/**
 * Find elt in the list.
 * pre: list exists.
 * post: you know the position of elt in the list. If
 * the returned position is 0 then elt is NOT in the
 * list.
 * @param elt itemType item to find.
 * @param aList List to be searched.
 * @return int position in list of elt.
 */
find(in elt:itemType, in aList:List):integer {
    for (int i = 1 to aList.size()) {
        if(aList.get(i).equals(elt))
            return i;
    }
    return 0;
}

Homework

- Read 4.1, 4.3; Study 4.2 (not axioms)
- Answer pp. 215-217 #1, 5, 12, 18
- Due date is given in class

SortedList ADT

Just like a list, except that

for all i = 1 .. list.size()-1
and elt(i)is an element of the list:

elt(i) <= elt(i+1)

So you CAN’T insert by position!
Also, what if the item isn’t in list?

Exercise: SortedList

<table>
<thead>
<tr>
<th>SortedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
</tr>
<tr>
<td>createList();</td>
</tr>
<tr>
<td>isEmpty();</td>
</tr>
<tr>
<td>size();</td>
</tr>
<tr>
<td>add(item);</td>
</tr>
<tr>
<td>delete(pos);</td>
</tr>
<tr>
<td>delete(item);</td>
</tr>
<tr>
<td>get(pos);</td>
</tr>
<tr>
<td>find(item);</td>
</tr>
</tbody>
</table>
SortedList add

```cpp
/**
* Add item to list.
* pre: none.
* post: item is in list && size() is one more && get(find(item)) <= get(find(item)+1)
* @param item itemType to be inserted
*/
+add(item:itemType);
```

SortedList delete

```cpp
/**
* Delete item from list.
* pre: none.
* post: item no longer in list.
* @param item itemType to be deleted
*/
+delete(item:itemType);
```
Murphy’s Law

- The VAX is down.
- The VAX will crash the night the assignment is due.
- Do assignments the way Filipinos vote: early and often.

Testing object equality

- shallow: are the references equal
  ```java
  List aList, bList;
  // stuff happens
  if (aList == bList) ...
  ```
- deep: are the referred objects equal
  ```java
  List aList, bList;
  // stuff happens
  if (aList.equals(bList)) ...
  ```

Example Rational

```java
/**
 * Checks if rat is attribute-wise equal to self.
 * pre: rat is a Rational object and exists
 * post: you know if they’re deep equals.
 * @param rat Object to check
 * @return boolean true if equal
 */
public boolean equals(Object rat) {
    return ((rat instanceof Rational) &&
    (num == (Rational)rat.num) &&
    (den == (Rational)rat.den)
    );
}
```

Equality axioms

- reflexive: a == a
- symmetric: a == b iff b == a
- transitive: if a == b && b == c then a == c
```java
public boolean equals(ListAdt aList) {
    if (aList == null) {
        return false;
    }
    if (aList.size() != this.size()) {
        return false;
    }
    for (int i = 0; i < aList.size(); i++) {
        if (!aList.get(i).equals(this.get(i))) {
            return false;
        }
    }
    return true;
}
```

**List ADT via array**

```java
private final int CAPACITY = 10;
private int items[CAPACITY];
private int numItems;
```

- Array-based
- positions range from 1 .. size of list
- maximum capacity
- number of items
- elements are contiguous from 1 .. size
add(2, 6)

```
0 1 2 3 4 5 6 7 8 9
7 5 7
1 2 3 4 5 6 7 8 9 10
```

See p. 211 for code

delete(2)

```
0 1 2 3 4 5 6 7 8 9
9 6 8 7
1 2 3 4 5 6 7 8 9 10
```

See p. 212 for code

equals(aList)

```
0 1 2 3 4 5 6 7 8 9
9 6 8 7
1 2 3 4 5 6 7 8 9 10
```

What if capacities are different?

equals(aList)

```
public boolean equals(Object aList) {
    if ((aList instanceof ListAdt) &&
        (size() == aList.size())) {
        for (int i = 0; i < size(); i++) {
            if (items[i] != aList.items[i]) {
                return false;
            }
        }
        return true;
    }
    else {
        return false;
    }
}
```
equals(aList)

```java
public boolean equals(Object aList) {
    if ( !(aList instanceof ListAdt) ||
        ( size() != aList.size() ) ) {
        return false;
    }
    for (int i = 0; i < size(); i++) {
        if (items[i] != aList.items[i]) {
            return false;
        }
    }
    return true;
}
```

HW questions...

```java
/**
 * Compute sum of all items in list.
 * pre: list exists && is list of ints
 * post: you know sum of items.
 * @return int sum of items
 */
public int theSum() {
    int sum = 0;
    for (int i = 1; i < size(); i++) {
        sum += get(i);
    }
    return sum;
}
```