ADT Abstract Data Type

For collections, use with generics

class = Attributes (variables, nouns) + Behaviors (methods, verbs)
    inheritance (is-a, can use), reuse of type, # protected accessibility
    Java has single inheritance of classes
    abstract (can-have), reuse of type, can't instantiate
    composition( has-a), use of type

enum reuse w/ composition symbolic constants,

interface must-have, must-use: constants, enums, abstract methods
    collections of interfaces
    restriction of implementor's functionality
    specific interfaces used with multi-implementor classes
    java has multiple implementations (behavioral inheritance)

generics
    declare collections w/ actual types specified at instantiation
    implementation of Comparable: int compareTo(E other)
UML Diagrams

Abstract Class

- privateVariable : type
  # protectedVariable : type
  + publicVariable : type

+ Constructor(arguments)
- privateMethod(args) : type
  # protectedMethod(args) : type
  + publicMethod(args) : type

AncestorClass

composition

has – a

array, collections

must override inherited

abstract methods

DescendantClass

inheritance

is – a

super

factoring

inheritance

is – a

super

factoring

Class Name
Linked List UML

A singly linked list ADT
allocates space as needed
has 1 "head" SLLList<SLNode>
has 0 or many "nodes" SLLNode<E>

```
SLLList<E>
- name : String
- size : int
- next : SingleLinkedNode
+ SLLList()
+ insert(E e) : void
+ remove(E e) : E
+ show() : void
```

```
SLLNode<E>
    implements Comparable<E>
- element : E
- next : SLLNode<E>
+ SLLNode(E anE)
+ compareTo(SLLNode<E> other) : int
+ get() : E
```

```
aList
| size is 4 |
| next is b |
```

```
<table>
<thead>
<tr>
<th>aList</th>
<th>b</th>
<th>m</th>
<th>h</th>
<th>c</th>
<th>87</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>size is 4</td>
<td>20</td>
<td>43</td>
<td>46</td>
<td>87</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>
```
3 insertion and 4 deletion cases

create SLLNode
   SLLNode newNode ← new SLLNode(E)

1. insert at "front" of list,
2. insert within in the list
3. insert at the end of the list

Deletion
1. remove from empty list    SSList.next == null, return null
2. remove "front" of list,
   traverse list looking for target, stop when found, or end of list
3. remove from list
4. target "e" not in list
BSTree UML

Element is an example class type for <E>
traversal

BSTNode {
  Integer key
  BSTNode right, left;
  BSTNode(Integer k) {
    key = k
    right = null
    left = null
  }
  Integer get() {
    return key
  }
  int compareTo(...) {
  }
}

BSTNode root;    // "n"

inorder(BSTNode n) { // iterator
  if (n == null) return;
  inorder(n.left)
  visit() // print?
  inorder(n.right)
}

Write w/ generics <Element>

Change order of
  "go left"
  "visit"
  "go right"
Insertions and Deletions

Assumes no duplicate Elements allowed in BST

All insertions are on leaves
All insertions result in binary search tree

void insert (BSTNode newNode)
1. Empty tree
2. Non-empty tree

Deletion

All "actual deletions" are at leaves
All deletions result in binary search tree

deadNode ← BSTNode delete(BSTNode target)
  find target, mark for replacement,
  find immediate successor (replacement), replace,
  delete replacement (always a leaf).
Operations on a stack are:

- **Stack\(<E>\)(\())** create a new Stack that can hold E values
- **\(<E>\) top()** return a reference to the top of the stack, "peek" at the stack
- **push(\(<E>\))** put an E on top of the stack, top references E
- **pop(\(<E>\))** remove an E from the top of the stack, top has a new reference
- **boolean isEmpty()** returns true if stack has no E values else false
UML Queue

```
Queue<E>
- count : int
- front : QueueNode<E>
- rear : QueueNode<E>
+ Queue<E>()
+ empty() : boolean
+ enqueue(E) : void
+ dequeue() : E
+ peek() : E
+ size() : int
```

```
QNode<E>
- E element
- next : QNode<E>
+ QNode(E)
+ getElement() : E
+ getNext() : QNode<E>
+ setNext(QNode<E> node) : void
```
Array based implementation

Consider a statically allocated (fixed size) array Queue having a: count, front, and rear int variables.

Count is the size of the Queue, front and rear are indexes (subscripts) into the array.

The front of the Queue is not always at index 0.

As items arrive to the non-empty queue they are inserted into position 

\((\text{rear} + 1) \mod \text{capacity}\)

As items depart from the queue the value of front must be incremented

\(\text{front} \gets (\text{front} + 1) \mod \text{capacity}\)
JFC Deque as a Queue

<table>
<thead>
<tr>
<th>Queue Method</th>
<th>Equivalent Deque Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(e)</td>
<td>addLast(e)</td>
</tr>
<tr>
<td>offer(e)</td>
<td>offerLast(e)</td>
</tr>
<tr>
<td>remove()</td>
<td>removeFirst()</td>
</tr>
<tr>
<td>poll()</td>
<td>pollFirst()</td>
</tr>
<tr>
<td>element()</td>
<td>getFirst()</td>
</tr>
<tr>
<td>peek()</td>
<td>peekFirst()</td>
</tr>
</tbody>
</table>

Deque<E> aQueue = new LinkedList<E>();
aQueue is-a LinkedList object that can only be accessed with Deque's interface methods.
Deques can also be used as LIFO (Last-In-First-Out) stacks.

<table>
<thead>
<tr>
<th>Stack Method</th>
<th>Equivalent Deque Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(e)</td>
<td>addFirst(e)</td>
</tr>
<tr>
<td>pop()</td>
<td>removeFirst()</td>
</tr>
<tr>
<td>peek()</td>
<td>peekFirst()</td>
</tr>
</tbody>
</table>

This interface often used in preference to the legacy JCF Stack class.

Stack class extends Vector, a dynamic array based collection.

Oracle's documentation:
A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class.

```java
Deque<E> stack = new ArrayDeque<E>();
```
Binary Tree, Stack, Queue applications

Linked List
Small linear search, or static / ordered collections: binary search
dynamically sized collections Vs ArrayList (or Vector)

Binary Tree
Large dynamically updated search collections
Huffman coding, frequency based
Expression trees: infix → postfix, prefix
Heaps, implemented ArrayList tree, JCF Priority Queue, min heap

Stacks
Expression evaluation
Implementation of method calls (recursive algo w/o recursion)

Queues
Transmission (communication) buffers
   computer i/o stream: keyboard, hard disk, internet packets
digital media: streaming media audio, video
simulation: next event lists
Big O

Big O is "on order" analysis, efficiency of algorithms estimate, statement of algorithms efficiency growth rate

Time & Memory $\leftarrow$ fn(critical operations, frequency of operations)

O(...) $\leftarrow$ simplification of fn(critical operation, frequency of operation)

remove constants, assume critical operation(s) are constant

ignore constants $\quad O(n) \leftarrow O(n - 1)$

ignore low order terms $\quad O(n^3) \leftarrow O(n^3 + n^2 + n)$

ignore multiplicative constants $\quad O(n^2) \leftarrow O(7 \times n^2)$

combine growth rates $\quad O(n^2 + n) \leftarrow O(n^2) + O(n)$
### Big O values

| $O(1)$ | constant | index into an array, hash functions, pop/push stack, add/remove queue |
| $O(n)$ | linear | search unsorted array or linked list array, list, or BST traversal |
| $O(n^2)$ | quadratic | simple sorts: exchange, selection 2 nested loop through n items: duplicates ! sort |
| $O(\log n)$ | logarithmic | Binary search in a sorted list BST insert, find, Heap insert, remove |
| $O(n \log n)$ | "$n \log n$" | merge, quicksort |
| $O(a^n)$ | exponential | tower of Hanoi, recursive fibonacci, permutations where $a > 1$ |
algorithm evaluation

formal
- Big O, for initial design considerations

experimental
- measure representative cases (probabilistic variations), for time / space measurements: performance || system

software metrics
- KLOC (thousands lines of code)
- complexity measures (graphs, ...), for maintenance / modification, understandability

qualitative
- code review / walkthroughs
  for maintenance / modification, understandability

Size of problem determines degree of algorithm evaluation required.
P1 View2D

Interfaces are ADTs can be used for collections
Abstract class, inheritance, and polymorphism
Generic type
UML class diagrams

P2 Binary Tree Experiment, experience with BST

BinaryTree, BinaryTreeNode class design, implementation, testing
Big O, O log n expectations for BST
Experimental measuring of algorithm performance
    BST max and average levels, time

P3 Curious and Hungry Robot

Interfaces and JCF  Deque <E> memory = new LinkedList<E>();
Application design and states
Application as state transistion

State diagrams can help understand / design problem
Generics and Interfaces

Interfaces can be used to restrict the definition of implementing classes

Deque <E> memory = new LinkedList<E>();

Generic <E extends InterfaceADT>

Allows compiler to test method signatures of implemented (shared) methods without specifying actual generic class.
What to do next?

Comp 282:
- hashing: map functions, map classes: JCF HashMap, HashSet
- Self balancing search trees: AVL, red-black, B-Tree
  - JCF TreeMap, TreeSet
- graphs: \{nodes, edges\} adjacency, search, path finding
- multi-access collections
  - primary key access + non-key access (inverted)
- databases: design / normalization, query
  - SQLite, MySQL

Algorithms
- search: radix, heap sorts

Collection libraries other APIs
- Microsoft's .NET
- C++ STL (Standard Template Library)