Object Oriented Programming and Design in Java

Session 19

Instructor: Bert Huang

Announcements

- Homework 4 due **MONDAY**. Apr. 19
  - No multithreading in programming part

Review

- Deadlocks and the Dining Philosophers Problem
- More on Threads in Java
  - Thread, Runnable, Object javadoc
  - Keywords synchronized and volatile
  - ReentrantLock
- Programming by contract and threads

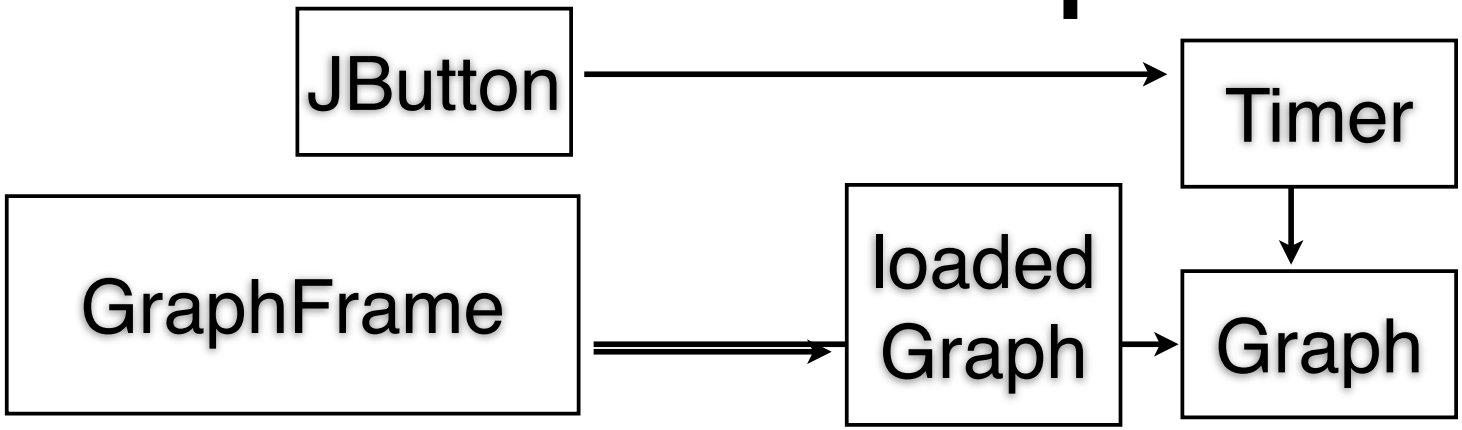
Today's Plan

- Homework tips
- Data Structures
  - Lists, Stacks, Queues
  - Sets, HashSet
  - Maps, HashMap

Homework Tips: Main Program

- Remember that the framework will do a huge portion of the work; make a non-animating version first
- Create the button that toggles the timer on and off. Test the timer and the toggle button by having actionListener print to console
- Each time Timer ticks, compute the new position and velocity for each node
- Force = k * lengthOfEdge, Accel. = Force / Mass
- Velocity = Velocity + Accel., Position = oldPosition + Velocity
- After moving nodes, call repaint() on the GraphFrame

Homework Tips



- Serialization will disconnect animation logic (Timer etc) because framework encapsulates Graph
  - Main can connect Graph and Timer to "Start Animation" JButton, but loaded Graph is a private reference
  - One solution: adding accessor for Graph in GraphFrame, and having the Timer call frame.getGraph().animate()

Abstract Data Types

- Data structures implement abstract data types (ADT), analogous to interfaces
- Algorithms for efficient data manipulation can be complicated; encapsulate them!
- Vast library of well-studied ADTs. Don't reinvent the wheel, don't reinvent the hash table
- ADTs include: Lists, Sets, Maps

ADTs and Interfaces

- It's good practice to treat all your data structures through their interfaces
- Only the constructor knows the actual type; changing implementation is easy
- Makes your code more reusable
- (but be careful about being too general)

Efficiency

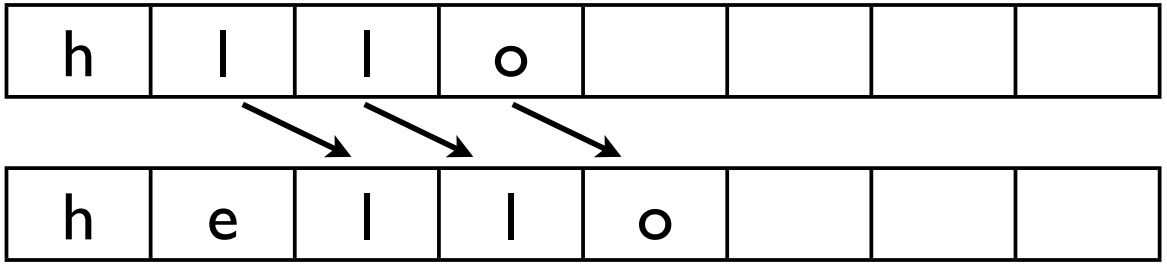
- Abstract Data Types usually have limited functionality
  - ideally optimized for the limited functionality
- The more limited the functionality, the faster the operations should be
- Design efficient programs by using the most limited ADT that will do the job

Lists

- An ordered series of objects
- Each object has a previous and next
  - Except **first** has no prev.,
    last has no next
- We can insert an object (at location *k*)
- We can remove an object (at location *k*)
- We can read an object (from location *k*)

ArrayList

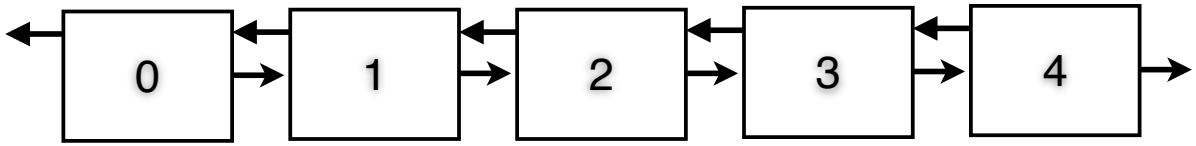
- Essentially a wrapper for an array
- Store elements in array, but handles list operations by shifting elements



- If array is full, copies into a new larger array
- O(1) get, O(N) insert/remove
  - O(1) insert/remove at the end of list

LinkedList

- Stores elements in Link objects
- Each link has reference to next (and prev)
  - prev links only in *doubly-linked* list
- Navigate by following next() references
- O(1) insert/remove with reference
  - But need O(N) to find (get) reference



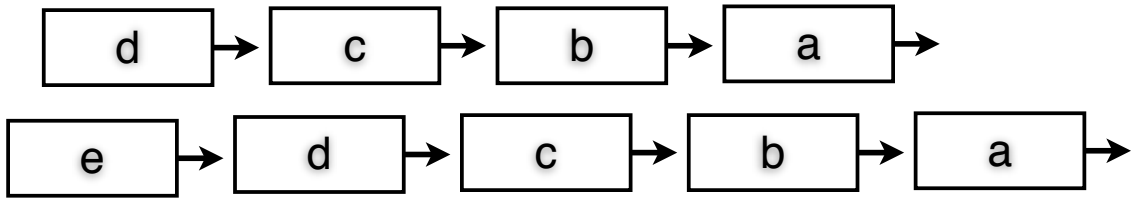
Stacks and Queues

image from
http://bwog.net/2006/05/03/tray-spotting

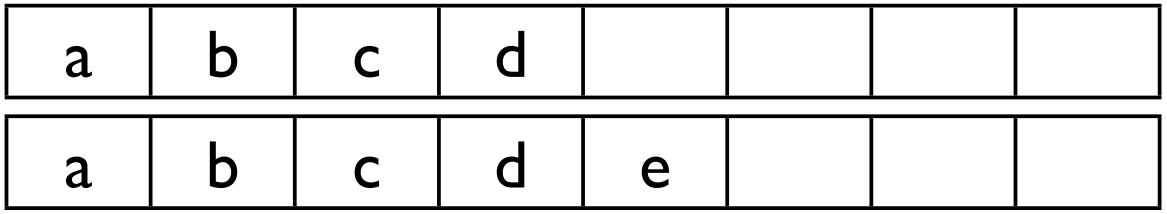
- Stack - Last in first out
  - push() - add element to top of stack
  - pop() - remove element from top
- Queue - First in first out
  - enqueue (offer) - add element at back of line
  - dequeue (poll) - remove from front of line

Stack Implementation

- Must be as fast as possible: O(1)
- Singly Linked List: add to beginning, remove from beginning



- Array List: add to end, remove from end



Queue Implementation

- Doubly-linked list: add at beginning, remove from end

- Array: "circular array"
  - mark beginning and end, wrap around when either exceeds array length
  - add at end, remove from beginning but don't shift

d	e				a	b	c
d	e	f				b	c

Hierarchy

- Should Stack or Queue implement the Collections Interface type?
- Should Stack or Queue implement the List Interface type?
- java.util.Stack extends Vector, which implements both
- java.util.Queue is subinterface of Collection, but not List

Sets

- An unordered collection
- No duplicate entries
- We can insert an object
- We can check for an object – contains()
- We can remove an object

HashSet

- Uses hashCode() to index into an array
- Collisions occur when distinct elements *hash* into the same index
- Collisions resolved by trying empty spots in a systematic way

Maps

- Maps are collections of objects "indexed" by other objects
- key types map to value types
- No duplicate keys, duplicate values allowed
- aka "associative array"

HashMap

- `Map<String, Double> costs = new HashMap<String, Double>();`
- `myMap.put("Big Mac", 2.99);`
- `myMap.get("Big Mac");`
- index by the key's hashCode()
  - but insert value instead of key

Sets, Maps, Collections

- Recall that Set is a subinterface of Collections that has no new methods
- HashMap doesn't implement Collection
- Has methods
  - `Set<K> keyset()`
  - `Collection<V> values()`

Reading

- Might be worth reviewing parts of previous reading:
  - Lists: Horstmann 1.11
  - Queues: Horstmann p. 42
  - Stacks: Horstmann p. 256-257
  - More discussion in section on Collections Framework, Section 8.3