**Linked List Practice**

* A pointer to a ListNode is different from the ListNode object itself. A pointer stores an address, while the ListNode object itself stores the value and another pointer.

ListNode temp; ListNode temp = new ListNode(“X”, null);

* Your name is a pointer. You and your body is the object. Your name points to, or references, you.

Person angela = new Person();

* When we work with ListNodes, we often need to check if a ListNode is pointing to the same memory location as another ListNode. E.g., if( temp == front )

1. The LList class below has a private pointer list that provides access to a linked list of ListNode objects. Inside the bigger box, draw a picture showing a typical state of this class. Inside the smaller box, draw a picture showing a special case/state for this class.

State 1 of LList

State 2 of LList

Of course, when writing code, you must be careful to take care of all the possible *states* of this object. The code below must work for either of the states above.

**public class** LList

{

**private** ListNode list; // points to the first node in list if

// the list is not empty; otherwise null

//constructor not shown

//Prints the contents of the linked list using a while-loop. The //linked list is not changed.

**public void** printWhile()

{

}

//Prints the contents of the linked list using a for-loop. The   
//linked list is not changed.

**public void** printFor()

{

}

//other methods not shown

}

**2.** This LList2 class has two private fields, a pointer to the front and a pointer to the rear of a regular ListNode linked list. The programmer has to update both the front and the rear, according to the state of the class. One possible state is when both front and rear are null. Draw the other two possible states:

class LList

list null

State 3 of LList2

front null rear null

State 2 of LList2

State 1 of LList2

**public class** LList2

{

**private** ListNode front; //points to first node in the list if list

//is not empty; null otherwise

**private** ListNode rear; //points to last node in the list if list

//is not empty; null otherwise

//constructor not shown

//Adds a new node to the front of the list. front must be updated. // rear may or may not be updated.

**public void** addFirst(Object obj)

{

}

//Adds a new node to the end of the list. rear must be updated.

// front may or may not be updated.

**public void** addLast(Object obj)

{

}

}

**3.** Complete the oddEven method for some ***client class***. Method oddEven should add even Integer values to the front of the LList2 object aList and add odd Integer values to the rear of the same object. Assume the methods you wrote above actually work, regardless of what you wrote.

**public static void** oddEven(LList2 aList, Integer val)

{

}