public class MyLinkedList {
    private class Node {
        int val;
        Node next;

        public Node (int val) {
            this.val = val; //this.val: the instance variable;  val: the parameter
            next = null; //next: the instance variable
        }
    }

    private Node head; // the first node and access point of the linked list

    //constructor, initialize an empty linked list
    public MyLinkedList() {
        head = null;
    }

    public void addAtHead(int val) {...} //done in the last lecture

    public static void main(String[] args) {

        //use addAtHead( ...) repeatedly to create the linked list shown in the figure at the top
        MyLinkedList list1 = new MyLinkedList();
        list1.addAtHead(15);
        list1.addAtHead(6);
        list1.addAtHead(23);
        list1.addAtHead(9);

        MyLinkedList list2 = list1; //link2 and link1 point to the same linked list

    }
**How to add a node at the tail:**

Main ideas:
- create a node, called `newTail`, with the given `val` and its next set to null
- if the linked list is not empty, start at the head, keep on traversing the linked list until we reach the current tail, make the current tail’s next to be the `newTail`
- If the list is empty, with `head` equal to null: let `head` to be the `newTail`

How do we know if a node, called `curr`, is the tail?
Answer: `curr` is the tail if it does not have a node after it, or equivalently, `curr.next == null` is true

//does the following program works for empty list, 1 node list, 2-node list, general list? Yes
public void addAtTail(int val) {
    Node newTail = new Node(val); //Theta(1) time
    if (head == null) //Theta(1) time
        head = newTail; //Theta(1) time
    else{
        Node curr = head; //Theta(1) time
        while (curr.next != null) //the loop is executed n times: Theta(n) total time
            curr = curr.next; //loop body: Theta(1) each time

        //at this point, curr is the old tail node
        curr.next = newTail; //Theta(1) time
    }
}

Given a linked list of length `n`, what is the **running time of the addAtHead(val) method** above?
Answer: Theta(n). See the running time comments in the code above.
Starting with an empty list, how to use addAtTail repeatedly to create the linked list above?

... main (...) {
    MyLinkedList list2 = new MyLinkedList();
    list2.addAtTail(9);
    list2.addAtTail(23);
    list2.addAtTail(6);
    list2.addAtTail(15);

    MyLinkedList list1 = list2;
}

Starting with an empty list, use addAtTail repeatedly to create a list of n nodes.

**What is the total running time, using our current addAtTail method given on the previous page?**

**Analysis:**
Create an empty list: Theta(1) time
Add the first node: Theta(1) time
Add the second node: Theta(1) time, traverse one node
Add the 3rd node: Theta(1) time, traverse two nodes
...
Add the (n/2)th node: traverse the existing (n/2 - 1) nodes, Theta(n/2 - 1) = Theta(n/2) = Theta(n) time
...
Add the (n-1)th node: traverse the existing (n-2) nodes, running time Theta(n-2) = Theta(n) 
Add the nth node: traverse the existing (n-1) nodes, running time Theta(n-1) = Theta(n)

- For the second half of the linked list, each new tail takes Theta(n) time to add. There are (n/2) nodes in this half. The total time to create the second half of the list is 
  Theta(n/2 * n) = Theta(n^2 / 2) = Theta(n^2)
- The first half of the linked list takes less time to create
- For the Theta notation, the total running time is determined by the running time of the second half of the list, which is **Theta (n^2)**. This is a quadratic time algorithm – inefficient

**Insight:** to have an efficient algorithm for addAtTail, it’d be helpful to have access to the tail directly, without traversing the list.
Alternative LinkedList Class Design and the Concepts of Invariants, pre-conditions and post-conditions

Intuitively:
- invariants: the properties that need to be maintained, for example on instance variables
- for each method:
  - pre-conditions: what are assumed to be true at the beginning of the method, e.g. with respect to the instance variables, parameters, etc..
  - post-conditions: what should be true at the end of the method, e.g. with respect to the instance variables, output, parameters, etc.

```java
public class MyLinkedList {

    // The following properties of the instance variables can be considered their invariants.
    // Also, if the list is empty, head and tail are null, length is 0.
    private Node head; // the first node and access point of the linked list
    private Node tail; // the last node of the linked list
    private int length; // the number of the nodes in the linked list

    // pre-conditions: head and tail point at the right nodes, and length has the right value
    // post-conditions: the invariants are still satisfied, after the instance variables have been updated

    public void addAtTail (int val) {
        Node newTail = new Node(val);
        if (head == null){
            head = newTail;
            tail = newTail;
        } else {
            tail.next = newTail;
            tail = newTail;
        }
        length
    }

    // Time Analysis related to the new addAtTail(...)
    - using the new addAtTail (..), each new tail is added in Theta(1) time, regardless of the list length.
    - starting with an empty list, repeatedly use addAtTail(...) to create a list of n nodes: the total time is Theta(n), much better than the old algorithm’s Theta(n^2) time.

    // Memory Analysis of MyLinkedList class
    - instance variable(s): Theta(1)
    - deep memory, including the list: Theta(n)
```