Outline

1. Understand why it is okay that the fields of ListNode are public
2. Get more familiarity with changing LinkedLists
3. Write more methods in the LinkedList class
4. Protecting Against NullPointerExceptions

ListNode Fields are public?

ListNode Class

```java
public class ListNode {
    public int data;
    public ListNode next;
}
```

This is our ONLY exception to the "make all fields private" rule

Why is this okay?
- Do we need them to be public?
  - Yes; we access data and next directly from LinkedIntList.
- Will our client be using ListNode?
  - The point of LinkedIntList is to handle manipulation of ListNodes for our client. Likely, they won’t touch ListNode.

A client of a LinkedList already knows that it’s made of ListNodes. We don’t expect them to use ListNode, but it’s okay if they do.

A New LinkedList Constructor

First Attempt

```java
public LinkedList(int n) {
    /* Current State */
    ListNode current = this.front;
    for (int i = 1; i <= n; i++) {
        current = new ListNode(i);
        current = current.next;
    }
}
```

What kind of loop should we use?
- A for loop, because we have numbers we want to put in the list.

What cases should we worry about?
- We’re creating the list; so, there aren’t really ‘cases’.

Remember, to edit a LinkedList, we MUST edit one of the following:
- front, or
- node.next (for some ListNode node)

In our code above, we edit current, which is neither.
A New LinkedList Constructor 4

Second Attempt

```java
public LinkedList(int n) {
    /* Current State */
    if (n > 0) {
        // is at least 1...
        this.front = new ListNode(i);
    }
    ListNode current = this.front;
    for (int i = 1; i <= n; i++) {
        current.next = new ListNode(i);
        current = current.next;
    }
}
```

This other solution works by going backwards. Before, we were editing the next fields. Here, we edit the front field instead:

```
public LinkedList(int n) {
    /* Current State */
    for (int i = n; i > 0; i--) {
        ListNode next = this.front;
        this.front = new ListNode(i, next);
    }
}
```

Implementing addSorted 6

```java
addSorted
Write a method addSorted(int value) that adds value to a sorted LinkedList and keeps it sorted. For example, if we call addSorted(10) on the following LinkedList, we would get:

```
    8   10   32   35
```

As always, we should approach this by considering the separate cases (and then drawing pictures):

- We’re supposed to insert at the front
- We’re supposed to insert in the middle
- We’re supposed to insert at the back

Case: Middle 7

An Incorrect Solution

```java
public void addSorted(int value) { //Say value = 10...
    ListNode current = this.front;
    while (current.next.data < value) {
        current = current.next;
    }
    current.next = new ListNode(value);
}
```

Uh Oh! We went too far! We needed the next field BEFORE us.

Case: End 9

Adding At The End?

```java
public void addSorted(int value) { //Say value = 48...
    ListNode current = this.front;
    while (current.next.data < value) {
        current = current.next;
    }
    current.next = new ListNode(value);
}
```

We fell off the end of the LinkedList. Idea: Make sure current.next exists.
public void addSorted(int value) {
    ListNode current = this.front;
    /* If we are making a check for current.next, we must */
    /* be sure that current is not null. */
    while (current.next != null && current.next.data < value) {
        current = current.next;
    }
    ListNode next = current.next;
    current.next = new ListNode(value, next);
}

A Fix?
public void addSorted(int value) {
    ListNode current = this.front;
    /* The extra check here is useless...we’ve already checked */
    /* current.next by the time we get to it. */
    while (current.next != null && current.next.data < value) {
        current = current.next;
    }
    ListNode next = current.next;
    current.next = new ListNode(value, next);
}

A Real Fix!
public void addSorted(int value) {
    ListNode current = this.front;
    while (current.next != null && current.next.data < value) {
        current = current.next;
    }
    current.next = new ListNode(value, next);
}

Case: Beginning
Our current code only sets current to a new ListNode. Importantly,
this never updates front; so, we lose the new node.

Adding At The Beginning?
public void addSorted(int value) { //Say value = -10...
    if (front == null) {
        front = new ListNode(value, null);
    } else {
        while (current.next != null && current.next.data < value) {
            current = current.next;
        }
        current = current.next;
        current.next = new ListNode(value, next);
    }
}

Case: End
Adding At The End?
public void addSorted(int value) {
    if (value < front.data) {
        front = new ListNode(value, next);
    } else {
        current = current.next;
    }
}

Have we covered all of our cases now?

Protect All Of Your Conditionals! Make sure that nothing can
accidentally be null.

When protecting your conditionals, make sure the less complicated
check goes first.

With LinkedList code, every time we make a test (if, while, etc.), we
need to make sure we’re protected. Our current code is:

We’re “protected” if we know we won’t get a NullPointerException
when trying the test. So, consider our tests:
- value < front.data
- current.next != null & current.next.data < value

So, Are We Protected?

Protection Our Tests!

Some LinkedList Tips!

- Make sure to try all the cases:
  - Empty List
  - Front of Non-empty List
  - Middle of Non-empty List
  - Back of Non-empty List

- To edit a LinkedList, the assignment must look like:
  - this.front = <something> ; or
  - node.next = <something> ; (for some ListNode node in the list)

- Protect All Of Your Conditionals! Make sure that nothing can
  accidentally be null.

- When protecting your conditionals, make sure the less complicated
  check goes first.