Implementing Lists

CIS 1181 Computer Science II

The main rule for an array implementation of a list in Java is simple: **there can't be any gaps in the array**

Also, keep in mind that once an array is created, its **size cannot be changed**.

Adding an item to an array list

If the list is not full and you're adding something onto the end, it's easy…

Sean	Will	Joe	Mike	
0	1	2	3	4

theList.add("Jake");

Sean	Will	Joe	Mike	Jake
0	1	2	3	4

Adding an item to an array list

If the array is full and you're adding something onto the end of the list, you have to make the array bigger first. Usually, we double its size.

Sean	Will	Joe	Mike	Jake
0	1	2	3	4

theList.add("Brian");

First, we create the new array…

0	1	2	3	4	5	6	7	8	9

Then, we copy the items from the old array to the new one (and trash the old array)

Sean	Will	Joe	Mike	Jake					
0	1	2	3	4	5	6	7	8	9

Adding an item to an array list

Finally, we add the new item at the end, just like we did in the first case, where we had room.

Sean	Will	Joe	Mike	Jake	Brian				
0	1	2	3	4	5	6	7	8	9

What if instead of adding Brian at the end, we wanted to add him at the third spot? **theList.add("Brian", 2);** To do this, we have to make room first, by scooting everything at index 2 or after over one space.

Sean	Will		Joe	Mike	Jake				
0	1	2	3	4	5	6	7	8	9

Then we can add Brian in the correct place.

Sean	Will	Brian	Joe	Mike	Jake				
0	1	2	3	4	5	6	7	8	9

Removing an item from an array list

Sean	Will	Joe	Mike	Jake	Brian				
0	1	2	3	4	5	6	7	8	9

To remove Mike from this list, we could use either **theList.remove(3);** or **theList.remove("Mike");**

Sean	Will	Joe		Jake	Brian				
0	1	2	3	4	5	6	7	8	9

But one of the rules is that we can't have any gaps. So we need to move any items with an index greater than 3 one space to the left.

Sean	Will	Joe	Jake	Brian					
0	1	2	3	4	5	6	7	8	9

Notice that Java doesn't trim the array down, even if there is a lot of space at the end. If you want to do this manually, call **theList.trimToSize();**

A key thing to remember about a linked list is that you must **always maintain the head and tail pointers** (the head points to the beginning of the list and the tail points to the end).

Also, **links are one-directional**. You can't go backwards from a Node object to find the one that comes before it.

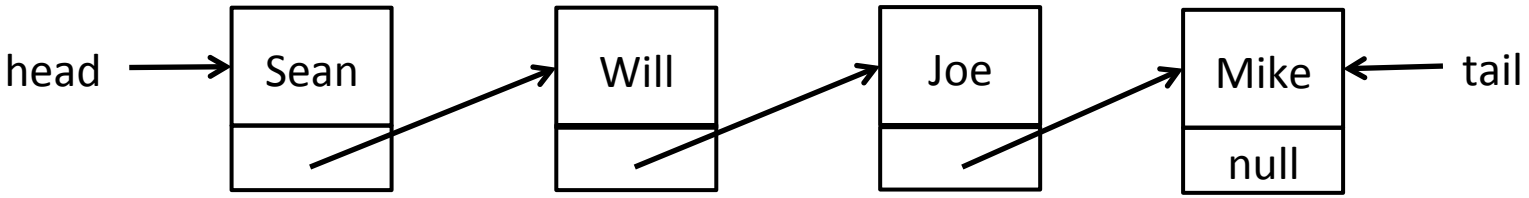
Finally, **the `next` variable of the last Node object in the list has the special value `null`**, because it doesn't point to anything.

A linked implementation of a list consists of a series of Node objects.

Each Node object contains an element (data) that the list is holding and a pointer to the next Node. The Node class uses generics so that the list can hold any type of data (Strings, Integers, FacebookUsers, etc.)

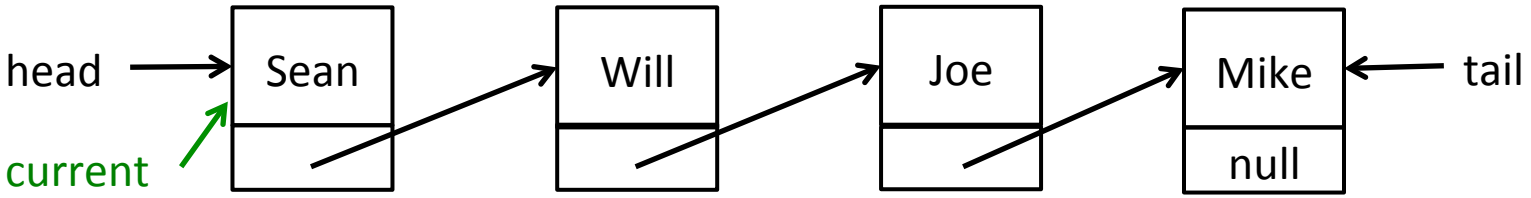
```
class Node<E> {
     E element;
     Node<E> next;

     public Node(E e) {
          element = e;
     }
}
```

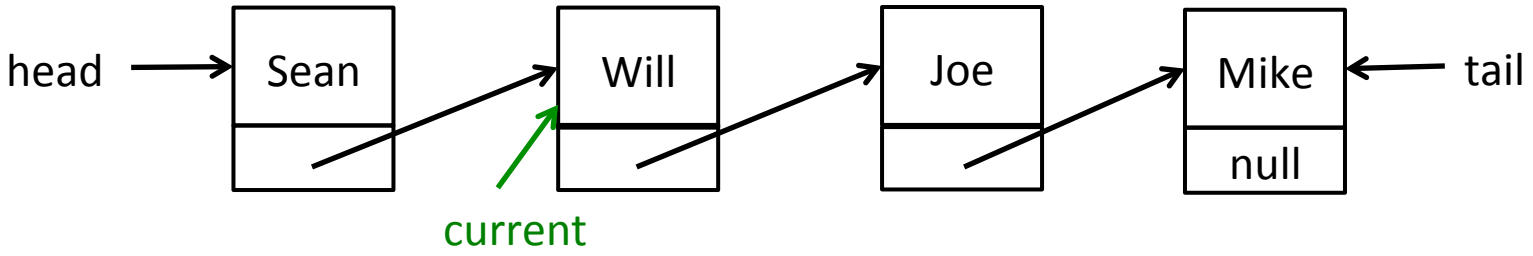


How would you display all of the items in a linked list?

```
1: Node current = head;
2: while (current != null) {
3:     System.out.println(current.element);
4:     current = current.next;
5: }
```

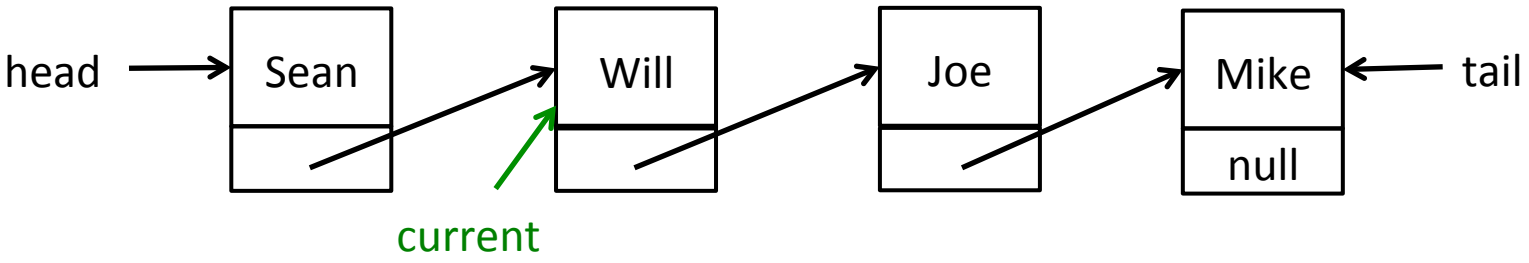


First `current` equals `head` (line 1). Because `current` isn't `null` (line 2), we print out its element, which is Sean (line 3). Then we update `current` by following its `next` pointer (line 4).

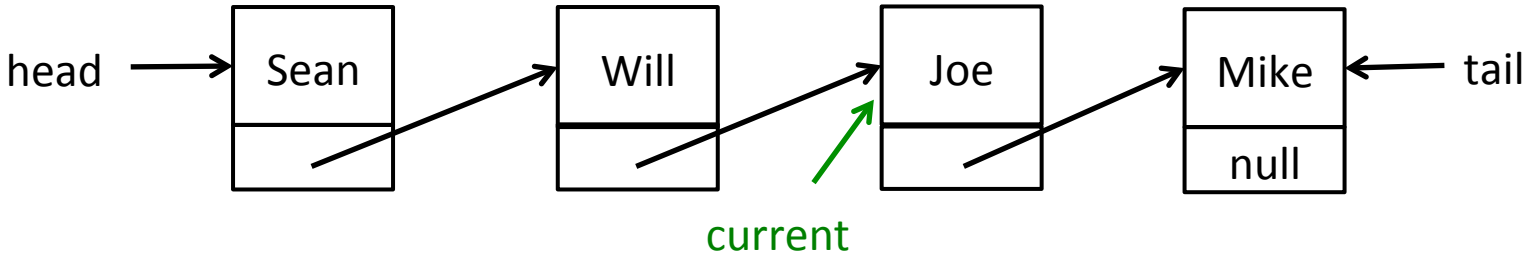


Now we go back to the beginning of the loop (line 2). `current` isn't null (it points to Will), so we enter the loop again.

```
Node current = head;
while (current != null) {
    System.out.println(current.element);
    current = current.next;
}
```

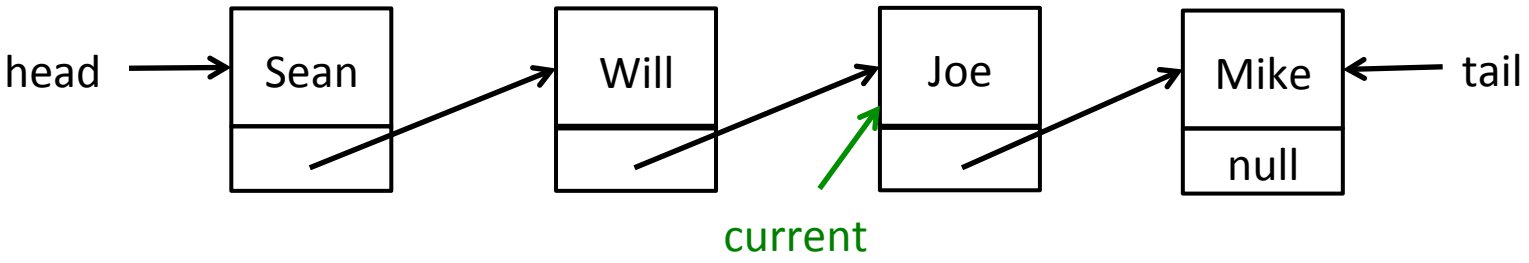


Inside the loop, we print out `current`'s element, which is Will, and once again update `current` by following its `next` pointer.

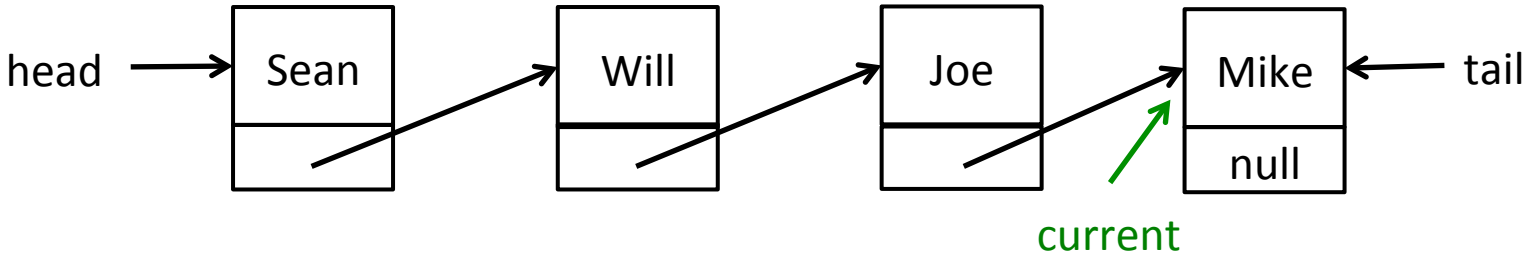


Back at the beginning of the loop once again, we see that `current` is still not `null` (it is pointing at Joe now), so we enter the loop.

```
Node current = head;
while (current != null) {
    System.out.println(current.element);
    current = current.next;
}
```

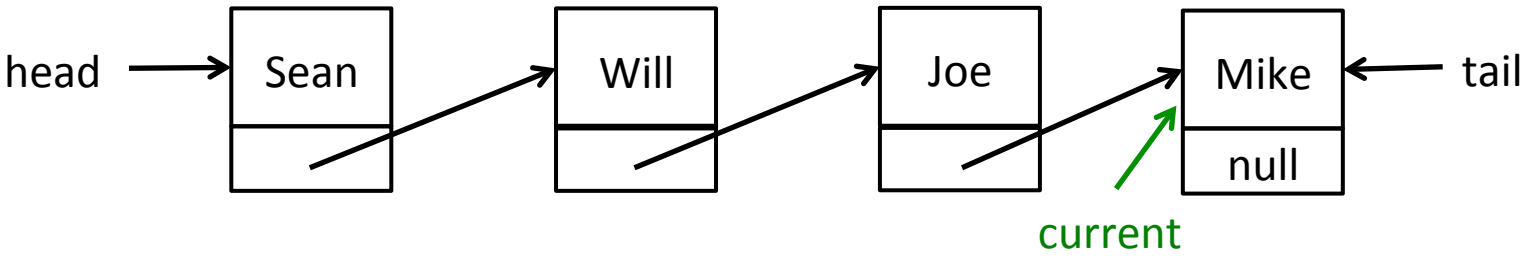


Now we print out Joe and follow the pointer (loops are repetitive, go figure…)

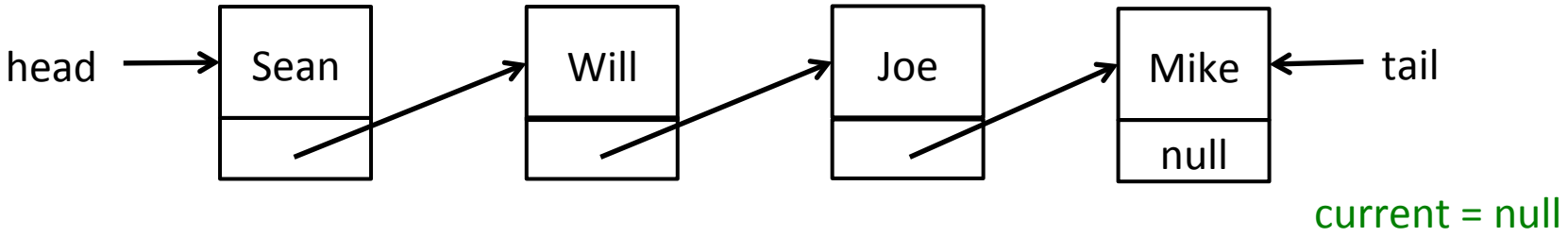


`current` still isn't `null` quite yet, so we need to enter the loop again.

```
Node current = head;
while (current != null) {
    System.out.println(current.element);
    current = current.next;
}
```



Now we print out Mike. When we try to update `current` by following the `next` pointer though, `current` gets the value `null`.

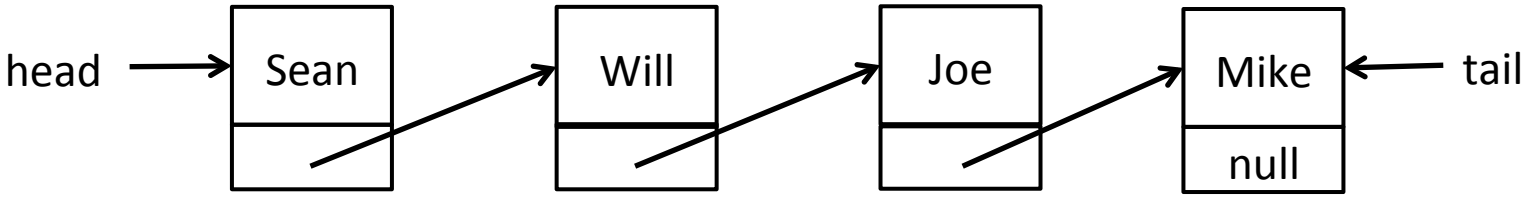


Back at the beginning of the loop, we see the `current` equals `null` and the loop is finished. This general pattern (using a current pointer to walk from the head of a linked list to its tail) will come up a lot in this chapter.

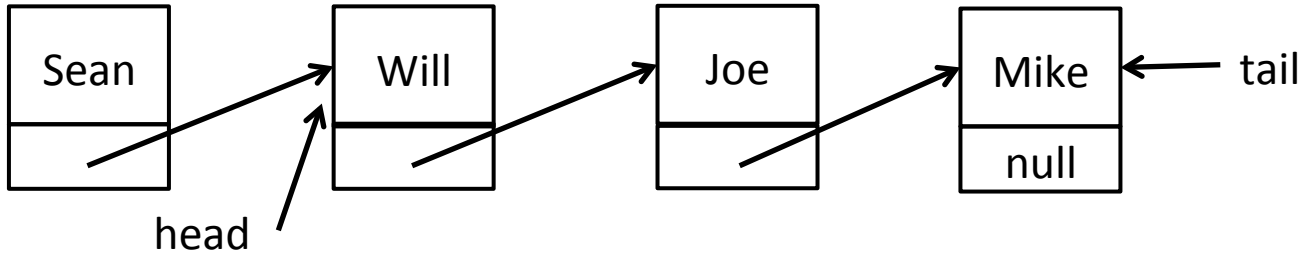
Why do we need the current pointer?

If we try to use the head pointer instead of creating a separate current pointer to walk through the list, terrible things happen…

```
while (head!= null) {
    System.out.println(head.element);
    head = head.next;
}
```

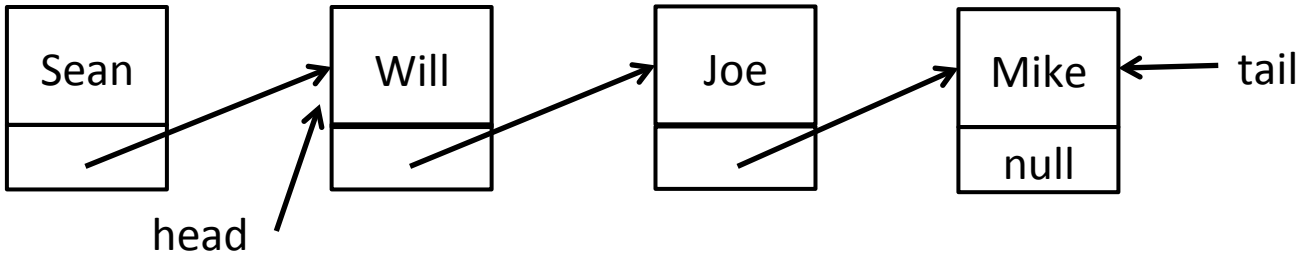


At first, everything starts off the same. `head` isn't `null`, so we print out its element (Sean) and update it by following its next pointer.



Why do we need the current pointer?

```
while (head!= null) {
    System.out.println(head.element);
    head = head.next;
}
```

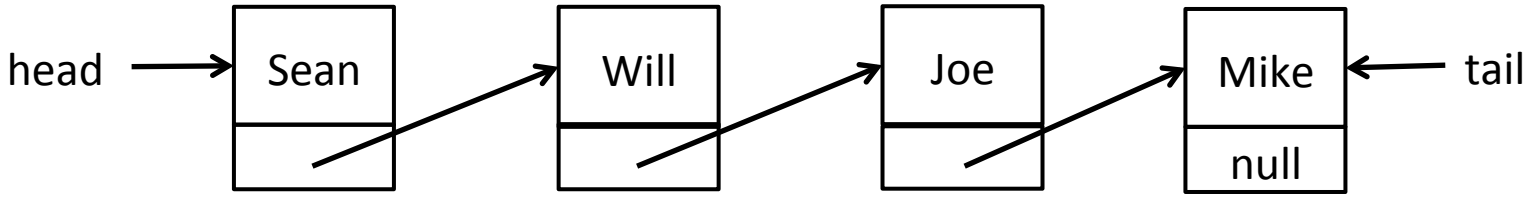


Can you see the problem here? We've lost Sean! Links in this type of linked list only work in one direction – there is no way to access the Node containing Sean ever again, because we don't have anything that points to it.

The next section of slides explains how to add and remove items from a linked list. Notice how every piece of code makes certain to update the head and tail pointers of the list correctly, so that no data is lost.

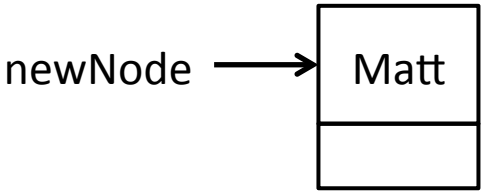
Adding an item to the beginning of a linked list

```
theList.addFirst("Matt");
```



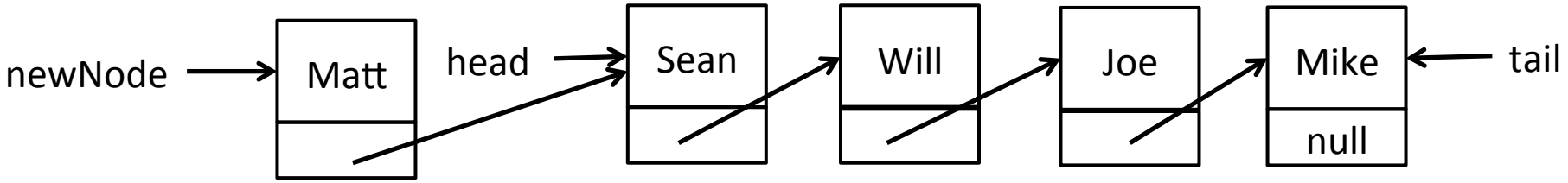
Step 1: Create the new node

```
Node<String> newNode = new Node<>("Matt");
```

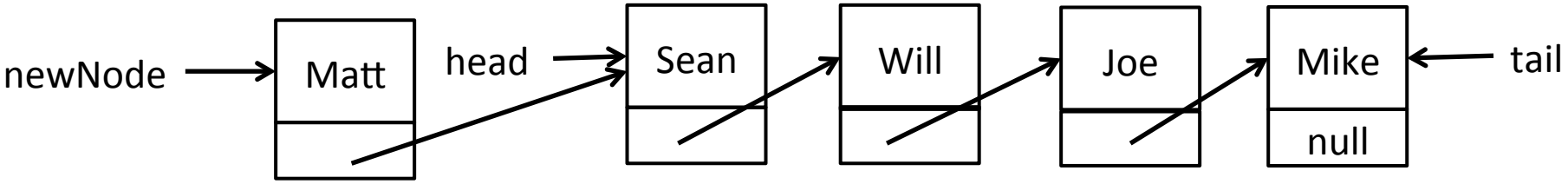


Step 2: Set the next pointer of the new node to the current head of the list

```
newNode.setNext(head);
```

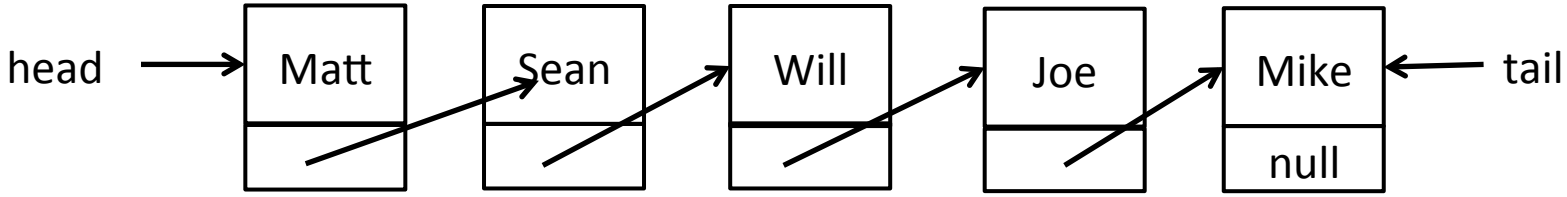


Adding an item to the beginning of a linked list



Step 3: Update the head pointer to point to the new node

```
head = newNode;
```

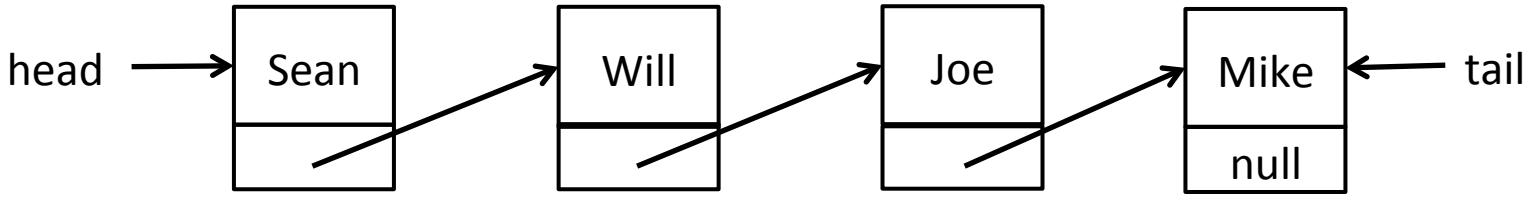


Step 4: If the list now contains only one item, set the tail equal to the head

```
If (tail == null) {
   tail = head;
}
```

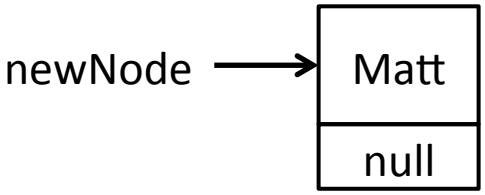
Adding an item to the end of a linked list

```
theList.addEnd("Matt");
```



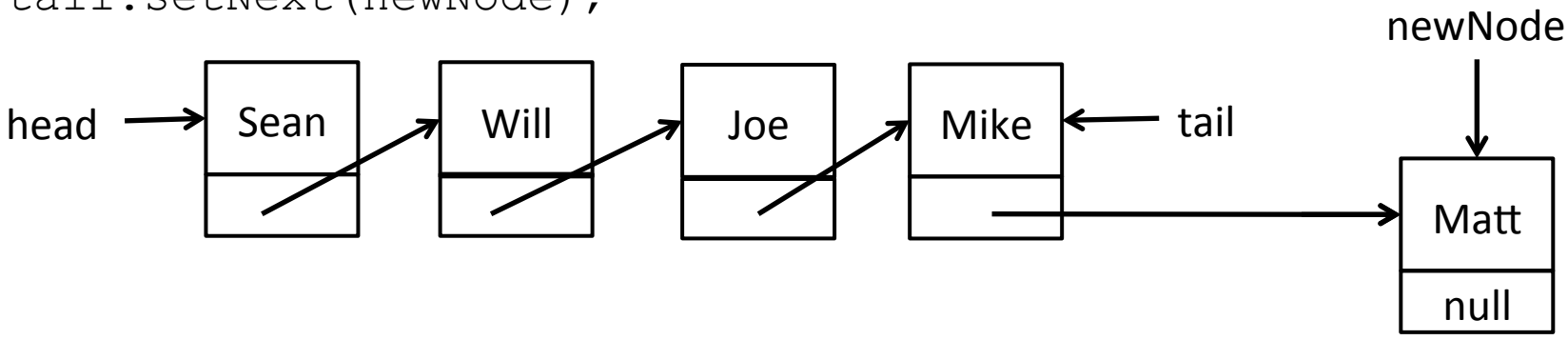
Step 1: Create the new node

```
Node<String> newNode = new Node<>("Matt");
```

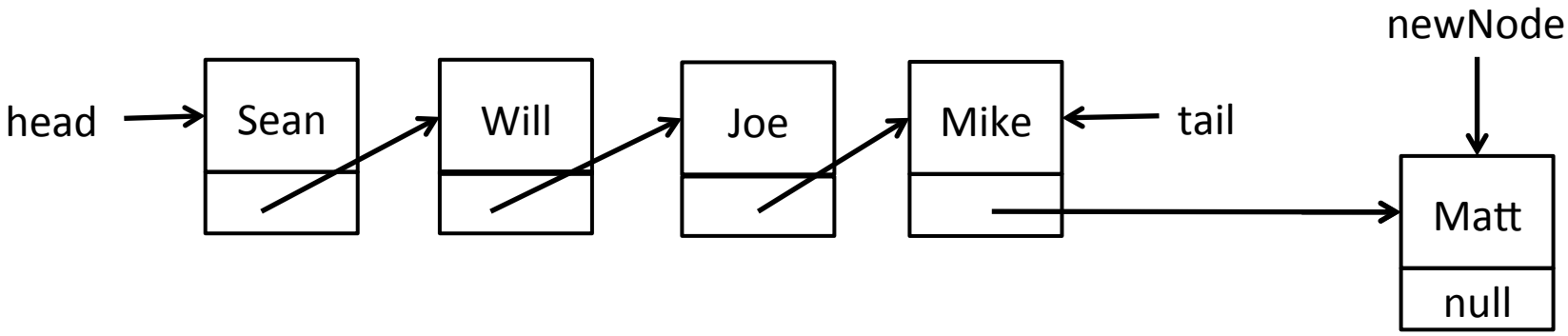


Step 2a: If this is the only node in the list, point the head and tail at it and you're done
Step 2b: Otherwise, set the tail node's pointer to this new node.

```
tail.setNext(newNode);
```

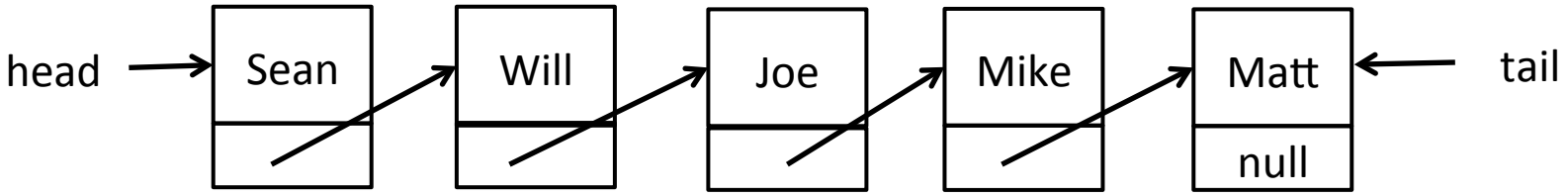


Adding an item to the end of a linked list



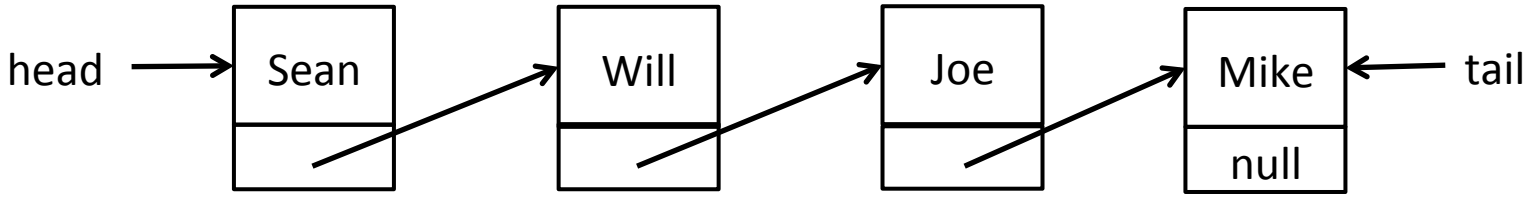
Step 3: Update the tail pointer

```
tail = newNode;
```



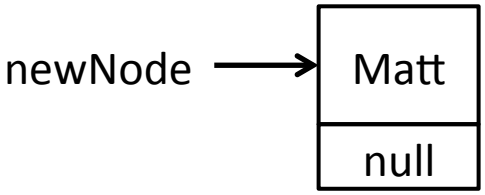
Adding an item at a particular index in a linked list

```
theList.add(3,"Matt");
```



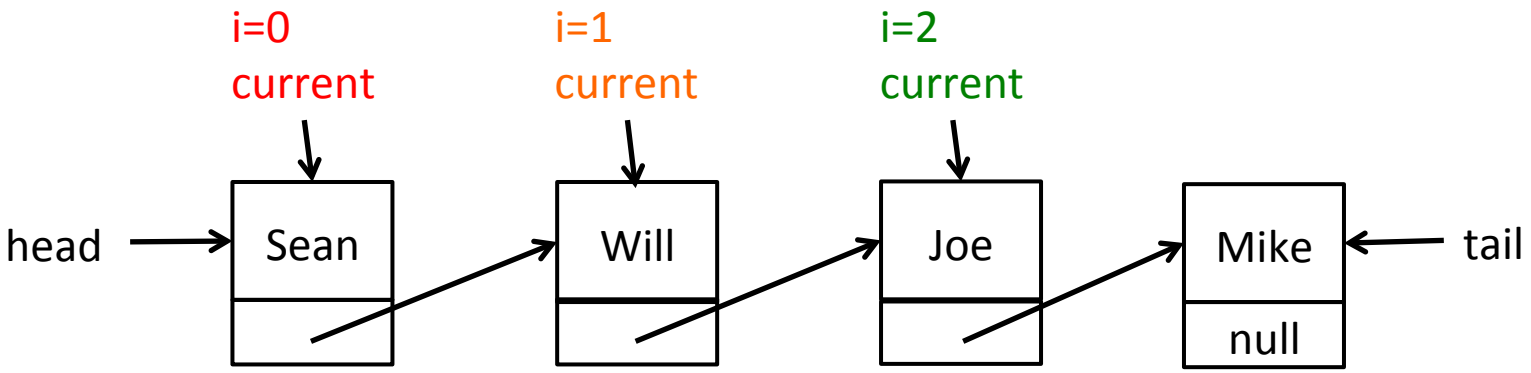
Step 1: Create the new node

```
Node<String> newNode = new Node<>("Matt");
```



Step 2: Find where to insert the new node

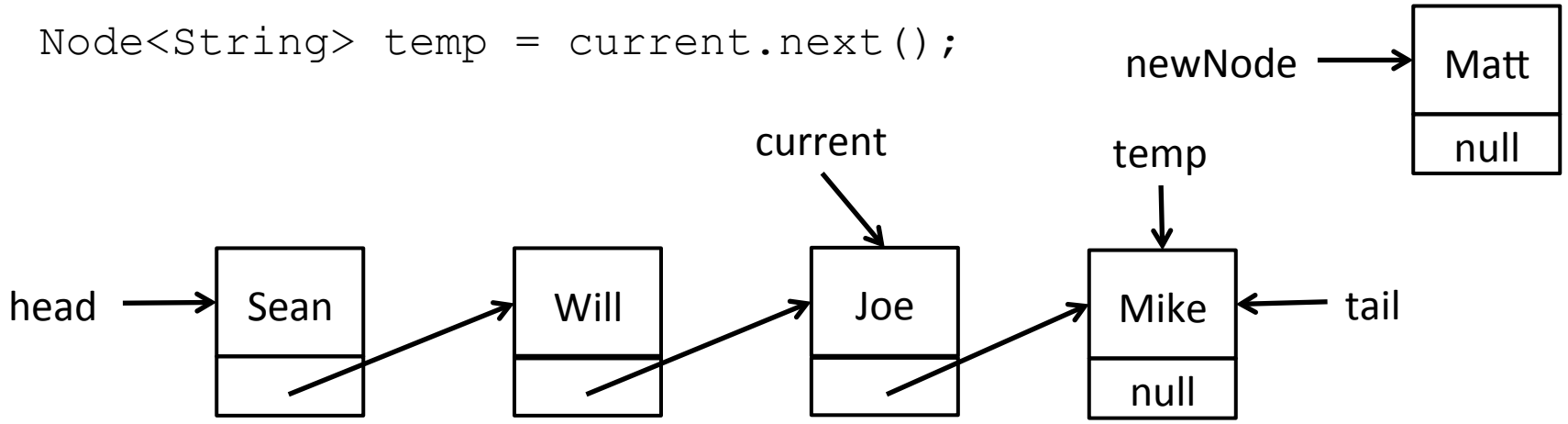
```
Node<String> current = head;
for (int i=0;i<index; i++)
    current = current.next;
```



Adding an item at a particular index in a linked list

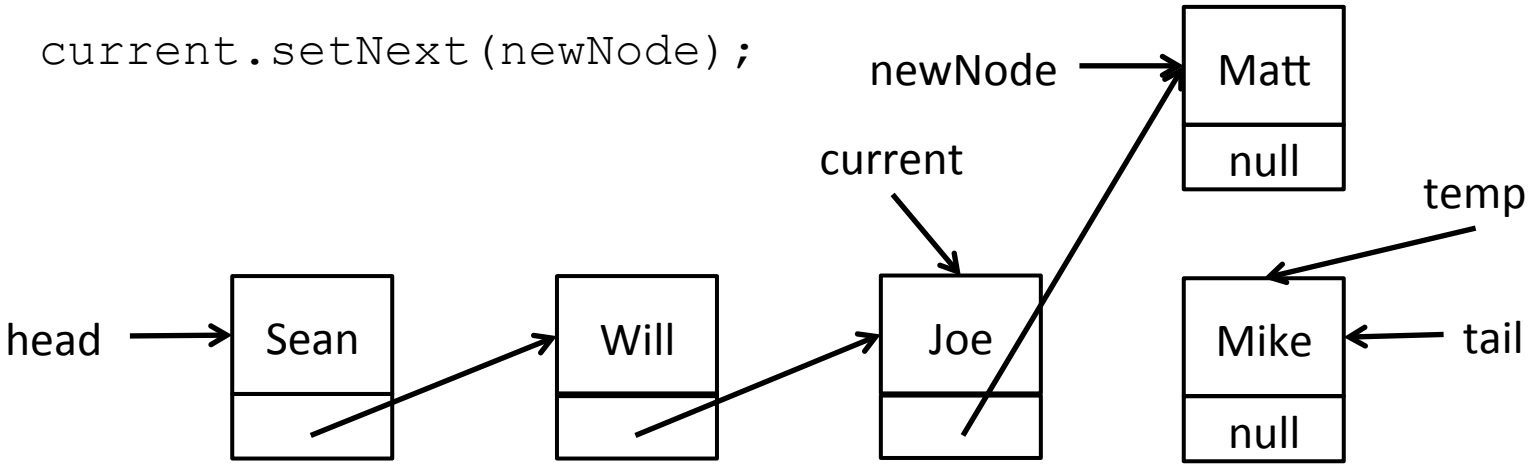
Step 3: Create a temporary point equal to current's next pointer

```
Node<String> temp = current.next();
```

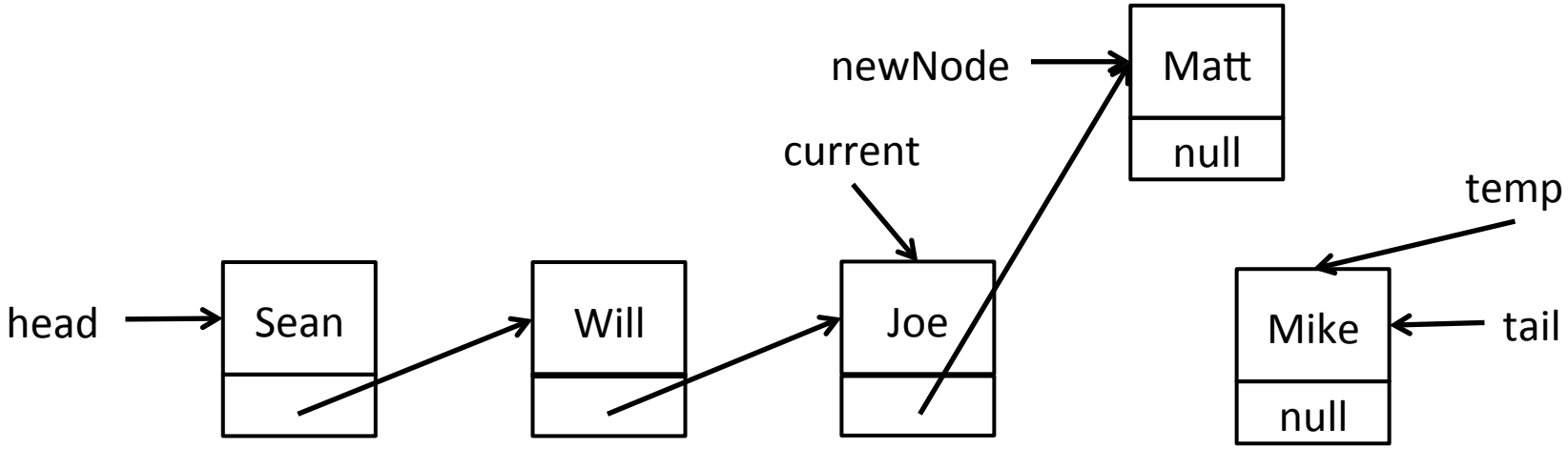


Step 4: Set the current node's next pointer to the new node

```
current.setNext(newNode);
```

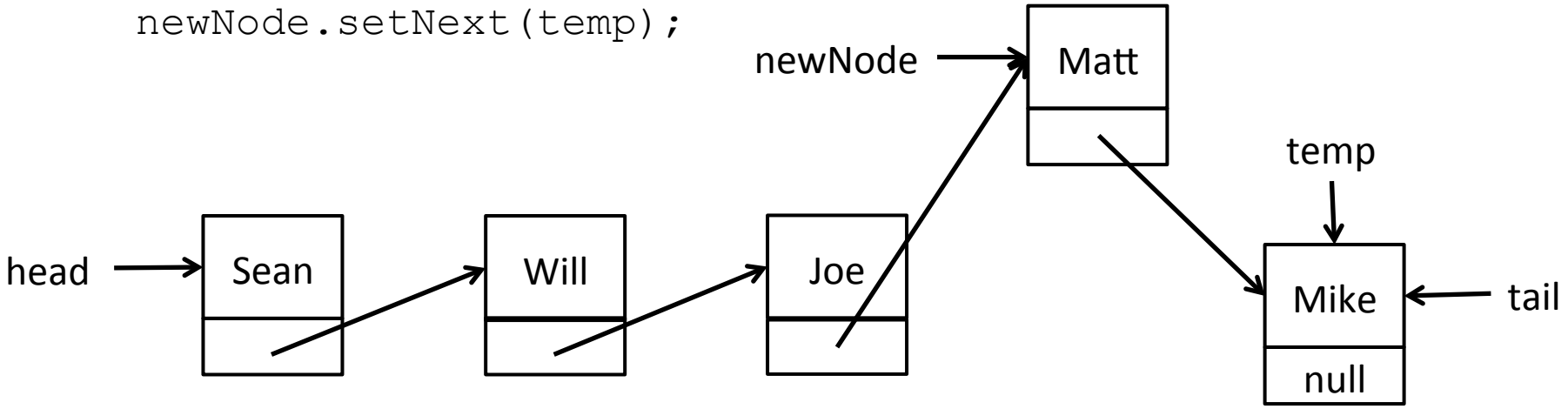


Adding an item at a particular index in a linked list



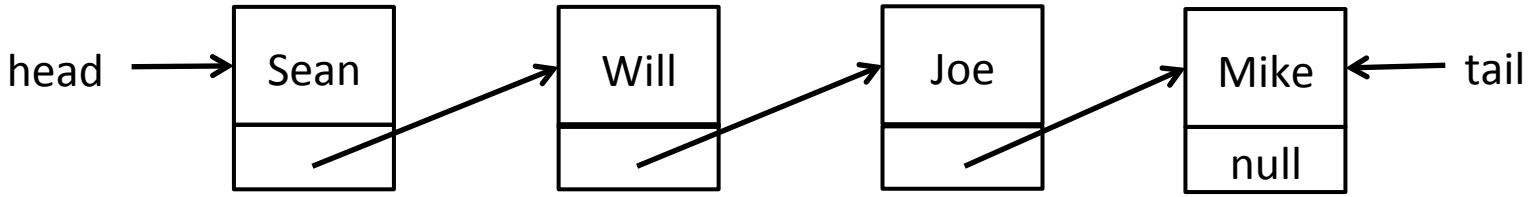
Step 5: Set the new node's next pointer to temp

```
newNode.setNext(temp);
```



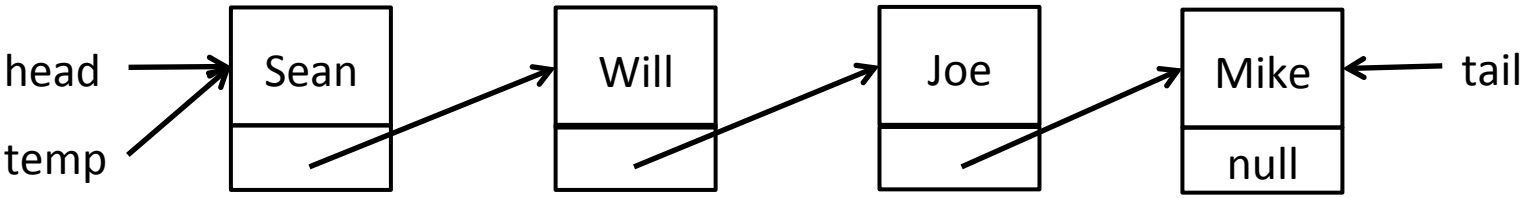
Removing an item from the beginning of a linked list

```
String firstString = theList.removeFirst();
```

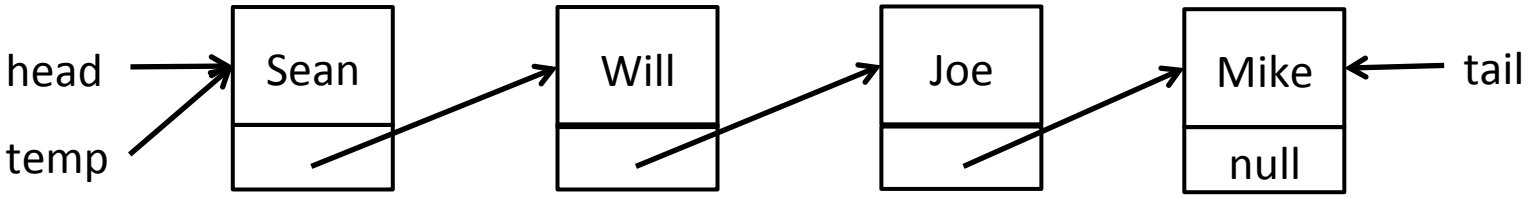


Step 0: If the list is empty, return null
Step 1: Create a temporary Node pointer to store the current head of the list

```
Node<String> temp = head;
```

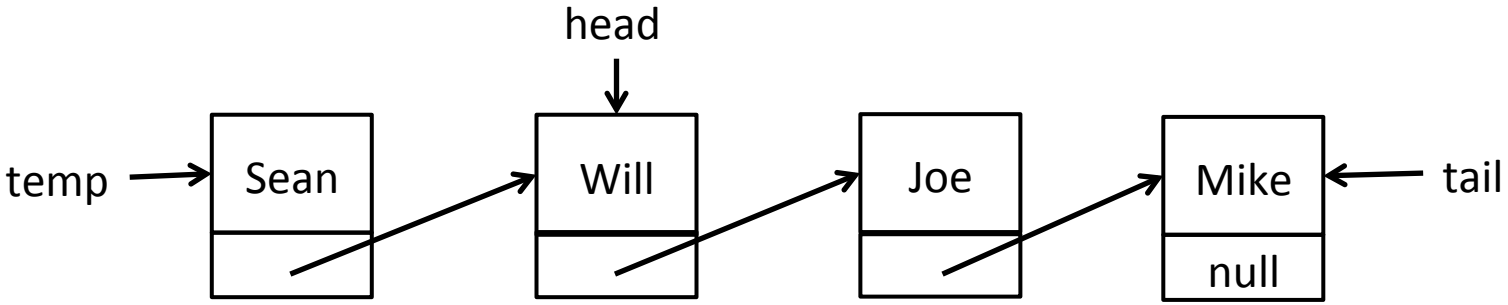


Removing an item from the beginning of a linked list



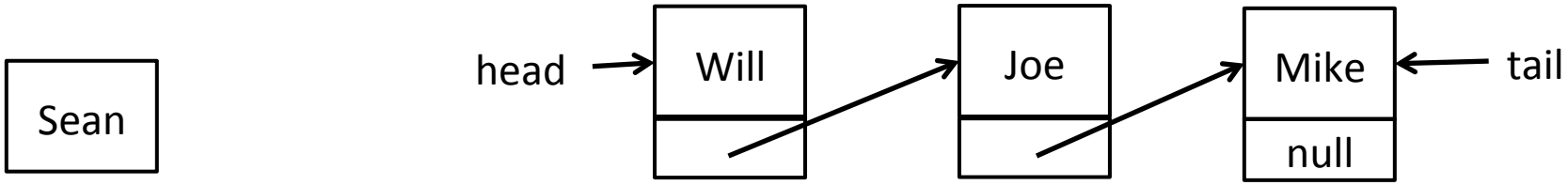
Step 2: Move the head to the next node

```
head = head.next;
```



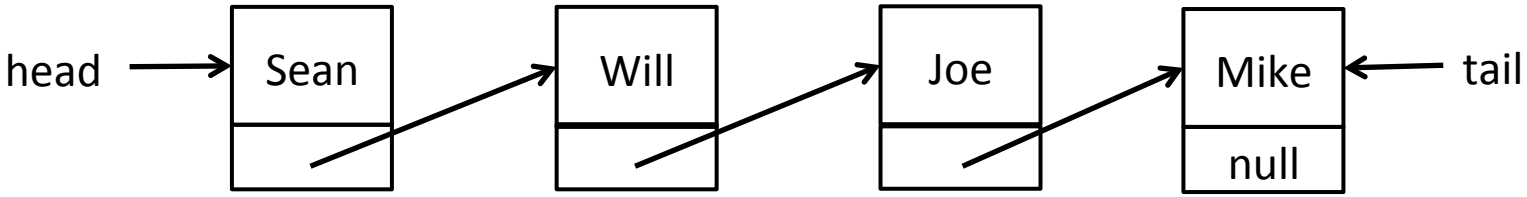
Step 3: If the head is now null, set the tail to null also (the list is now empty)
Step 4: Return the element of the temporary node pointer

```
return temp.element;
```



Removing an item from the end of a linked list

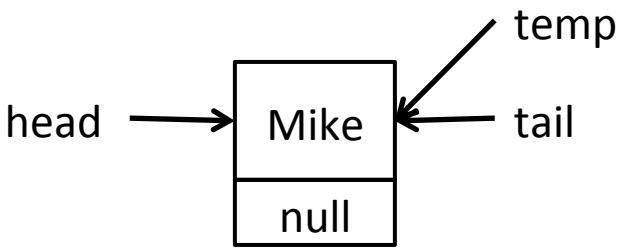
```
String lastString = theList.removeLast();
```



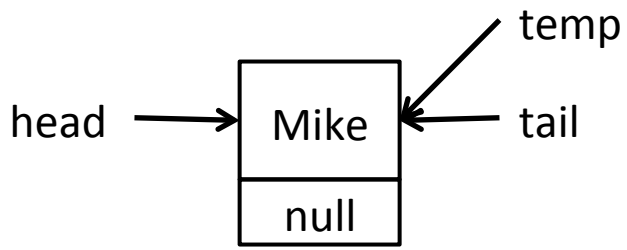
There are three cases.
In the first case, the list is empty. Just return null.
In the second case, the list has only one item:

Step 1: Create a temporary node pointer to store the current tail

```
Node<String> temp = tail;
```

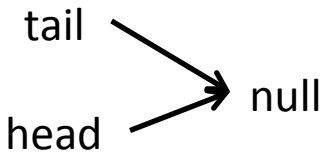


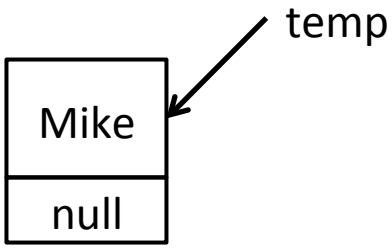
Removing an item from the end of a linked list (single item case)



Step 2: Set the head and tail pointer to null (the list is now empty)

```
head = null;
tail = null;
```

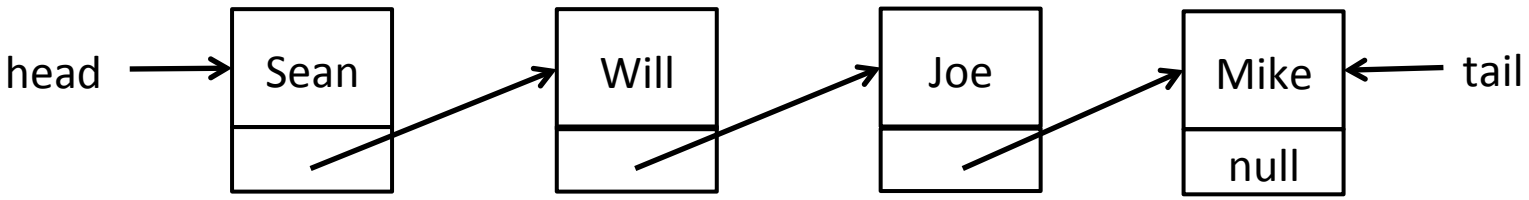




Step 3: return the element of the temporary node pointer

```
return temp.element;
```

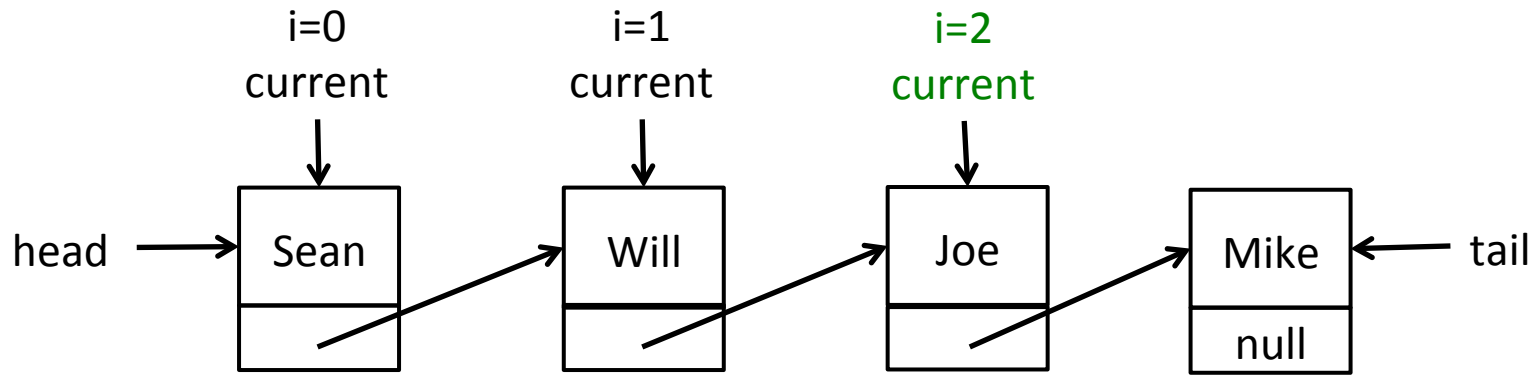
Removing an item from the end of a linked list (multiple item case)



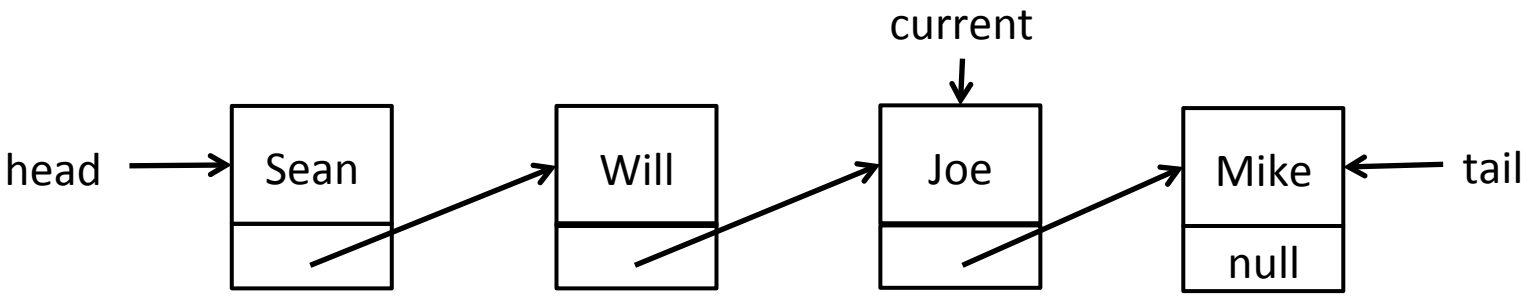
In the last case, the list has more than one item.

Step 1: Move through the list until you find the node right before the tail

```
Node<String> current = head;
for (int i=0; i<size-2; i++) {
   current = current.next;
}
```

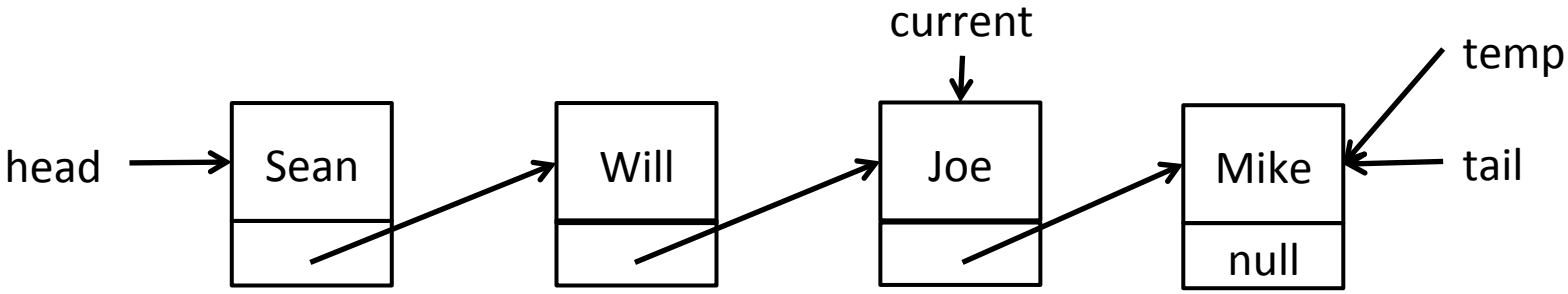


Removing an item from the end of a linked list (multiple item case)

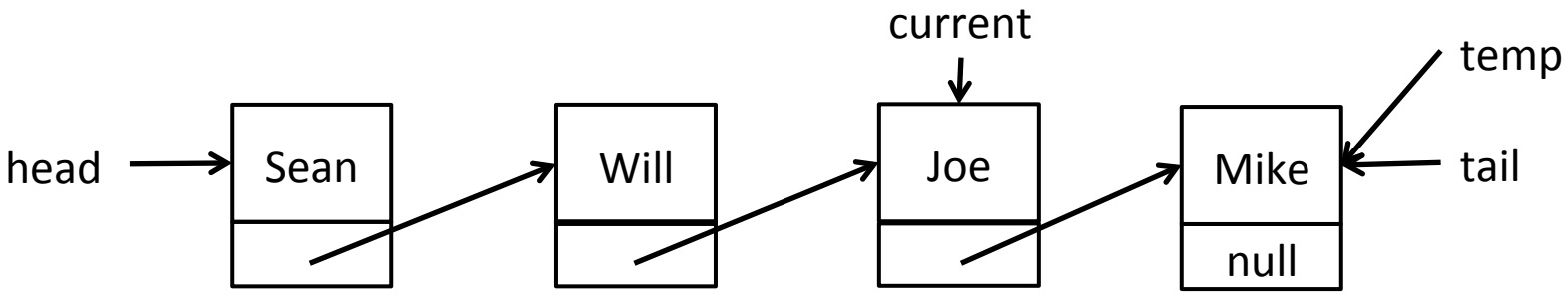


Step 2: Create a temporary node pointer to store the current tail

```
Node<String> temp = tail;
```

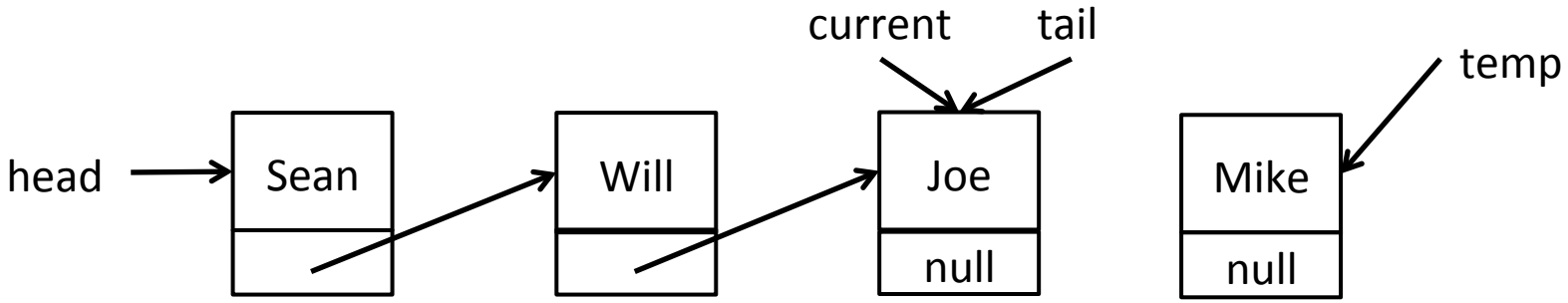


Removing an item from the end of a linked list (multiple item case)



Step 3: Set the tail pointer to current and set the current node's next pointer to null (it's the new end of the list)

```
tail = current;
current.setNext(null);
```

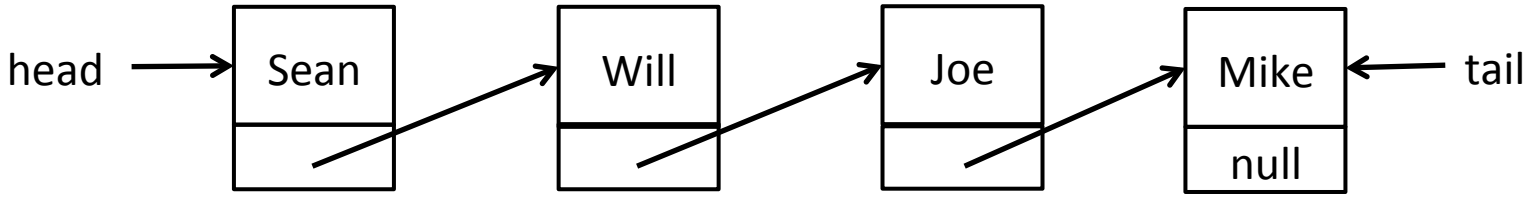


Step 4: Return the element of the temporary node pointer

```
return temp.element;
```

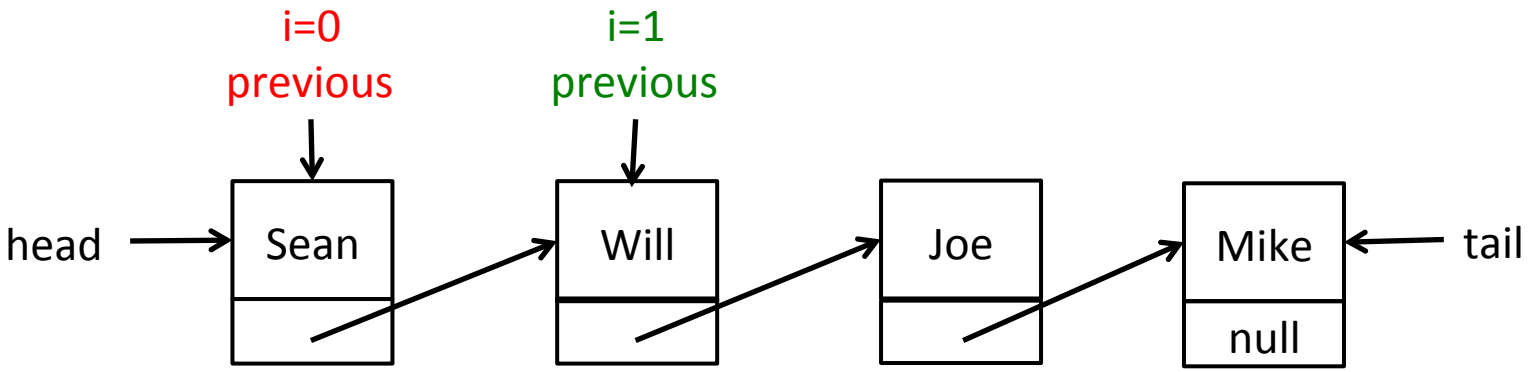
Removing an item from a particular index in a linked list

```
String thirdString = theList.remove(2);
```



Step 1: Find the node right the specified index

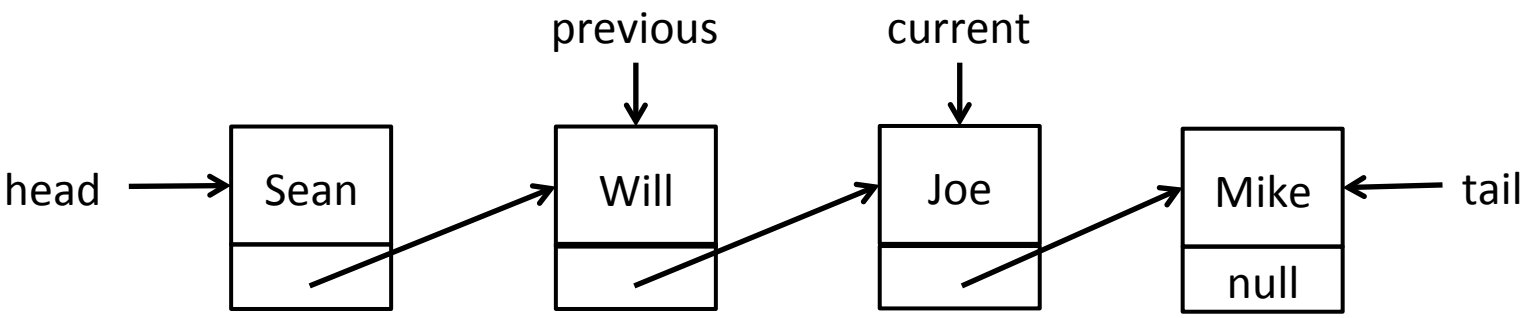
```
Node<String> previous = head;
for (int i=0;i<index; i++)
   previous = previous.next;
```



Removing an item from a particular index in a linked list

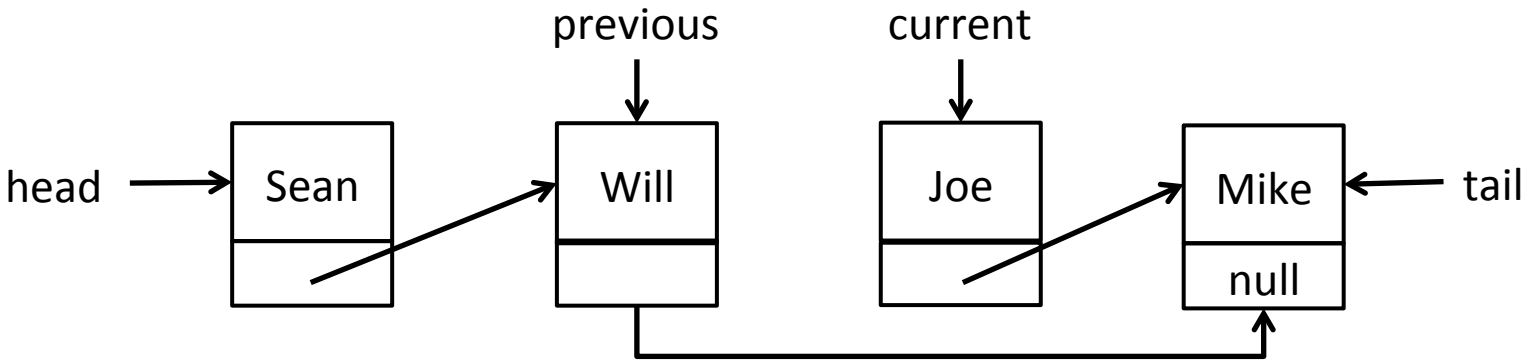
Step 2: Call the node right after the previous node 'current'

```
Node<String> current = previous.next;
```

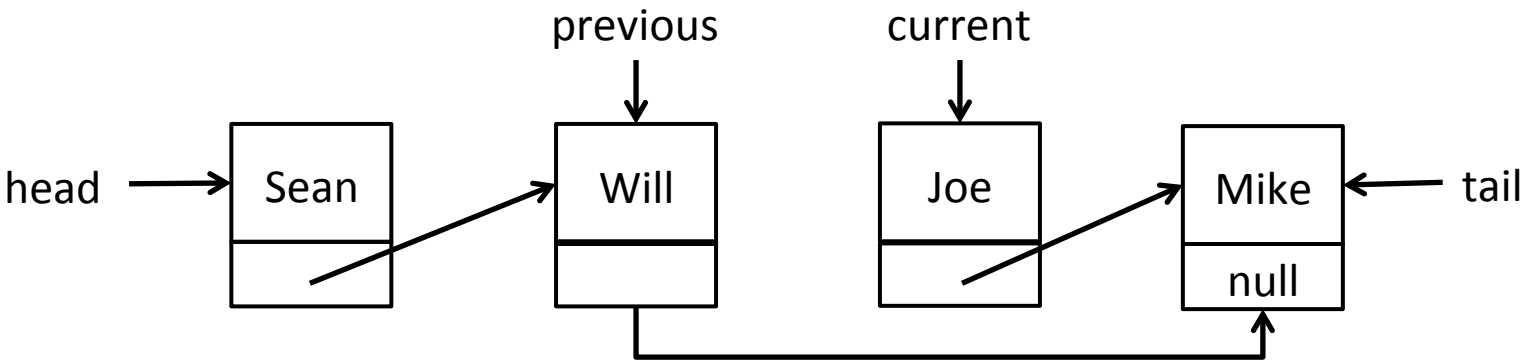


Step 3: Assign the 'previous' node's next pointer to the node after 'current'

```
previous.next = current.next;
```

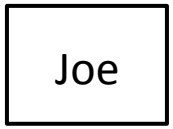


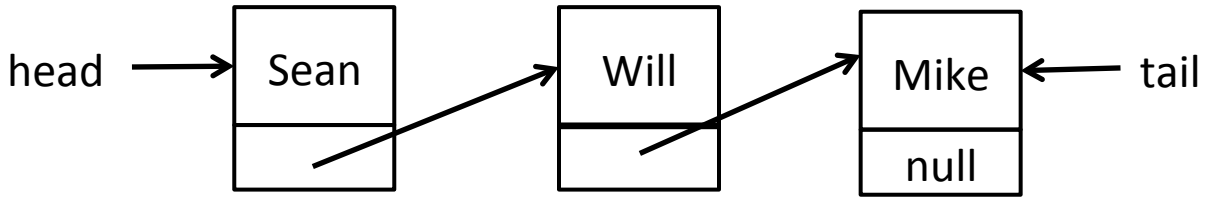
Removing an item from a particular index in a linked list



Step 4: Return the element of the current node

```
return current.element;
```





Doubly Linked and Circular Linked Lists

The linked list we just talked about is called a singly linked list. There is also a doubly linked list, where each Node object contains two pointers: previous and next (instead of just next). Removing items from the end of a doubly linked list is fast (it is slow for a singly linked list because you have to find the node right before the tail Node by starting at the head pointer and going all the way through the list). Your textbook also talks about circular linked lists, which have a ring shape an only one pointer.

Computational Complexity of Array versus Linked Node List Implementation

For the array implementation, accessing an item at a specific index if fast, but adding or removing items at the beginning of the list is slow (because you have to scoot all of the list items over one space).

For a linked implementation the reverse is true – accessing an item at a specific index is slow (because you have to start at the head pointer and move through the items until you get to the right one) but adding and removing items at the beginning of the list is fast (due to the head pointer).

You should be able to consider what types of accesses a program you are writing will need to make most frequently and choose the appropriate data structure based on that.