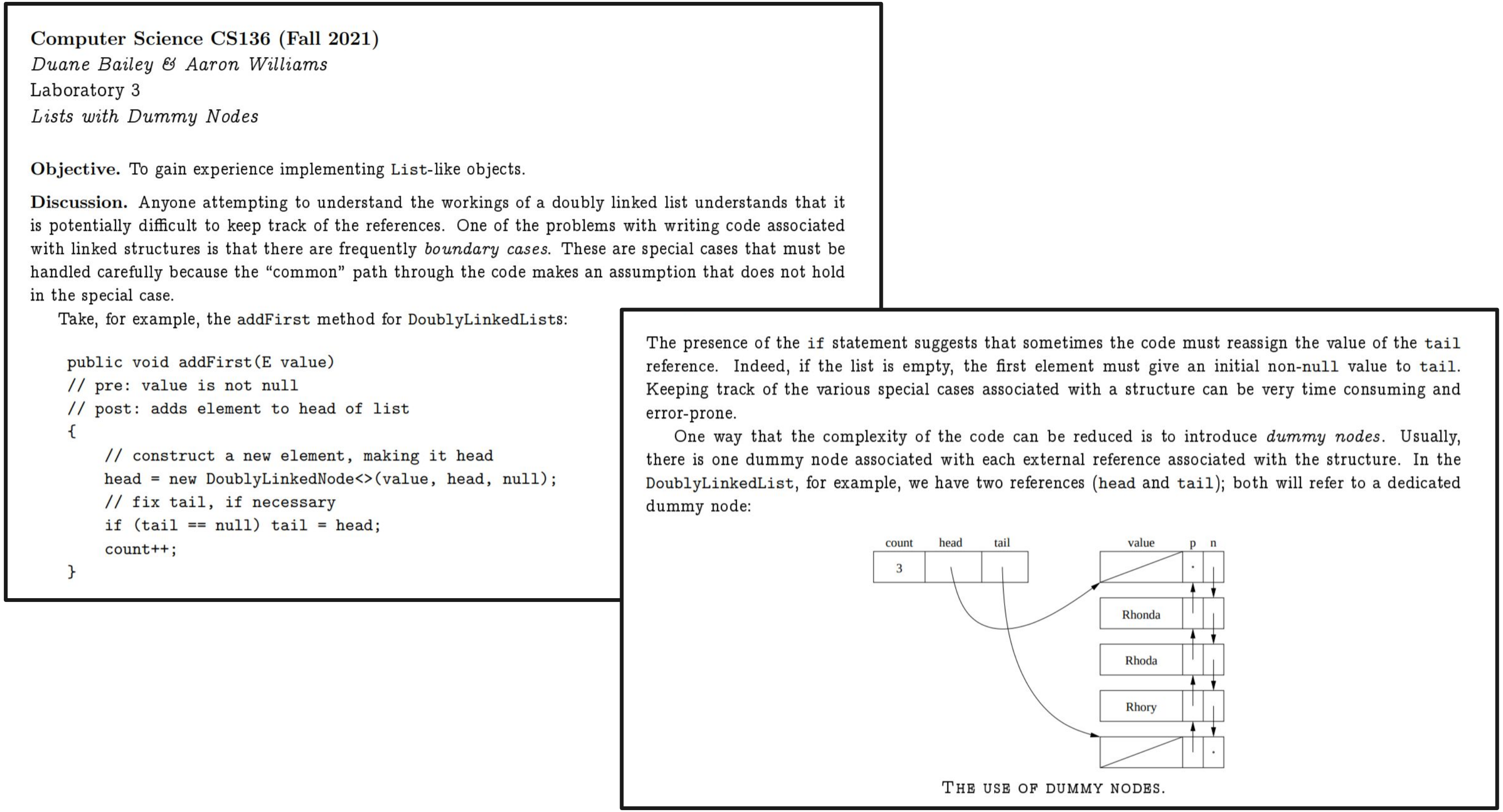
Midterm discussion on Friday

Lecture 10

Lists II

- Lab 3 – Preview
- Linked Lists
  - Nodes
  - `AddFirst`
  - `RemoveFirst`
  - `AddLast`
  - `RemoveLast`

Lab 3 — Preview



Computer Science CS136 (Fall 2021)
Duane Bailey & Aaron Williams
Laboratory 3
Lists with Dummy Nodes

Objective. To gain experience implementing List-like objects.

Discussion. Anyone attempting to understand the workings of a doubly linked list understands that it is potentially difficult to keep track of the references. One of the problems with writing code associated with linked structures is that there are frequently *boundary cases*. These are special cases that must be handled carefully because the "common" path through the code makes an assumption that does not hold in the special case.

Take, for example, the addFirst method for DoublyLinkedLists:

```
public void addFirst(E value)
// pre: value is not null
// post: adds element to head of list
{
    // construct a new element, making it head
    head = new DoublyLinkedNode<>(value, head, null);
    // fix tail, if necessary
    if (tail == null) tail = head;
    count++;
}
```

The presence of the if statement suggests that sometimes the code must reassign the value of the tail reference. Indeed, if the list is empty, the first element must give an initial non-null value to tail. Keeping track of the various special cases associated with a structure can be very time consuming and error-prone.

One way that the complexity of the code can be reduced is to introduce *dummy nodes*. Usually, there is one dummy node associated with each external reference associated with the structure. In the DoublyLinkedList, for example, we have two references (head and tail); both will refer to a dedicated dummy node:

THE USE OF DUMMY NODES.

In Lab 3, you will implement a doubly-linked list with "dummy nodes".
More specifically, you'll extend `DoublyLinkedList<E>` into a new class `LinkedList<E>`.
You'll overwrite some of the trickier methods with simplifications derived from the dummy nodes.

Linked Lists

Nodes

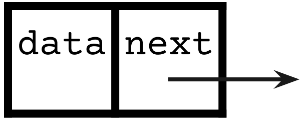
Nodes

Linked lists are comprised of nodes. Each node can be created or deleted one at a time.
This gives linked lists a fundamental advantage over arrays: They can be resized efficiently.

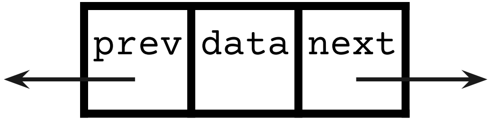
Every node contains at least the following:

- Some type of data. The `structure5` package refers to a generic type or class `<E>`.
- References to one or more nodes, each of which is `null` when there is no corresponding node. Note that *references* are known as *pointers* in some other languages.

In a *singly linked list*, the nodes only have references to the next node.
In a *doubly linked list*, the nodes have references to the next node and the previous node.

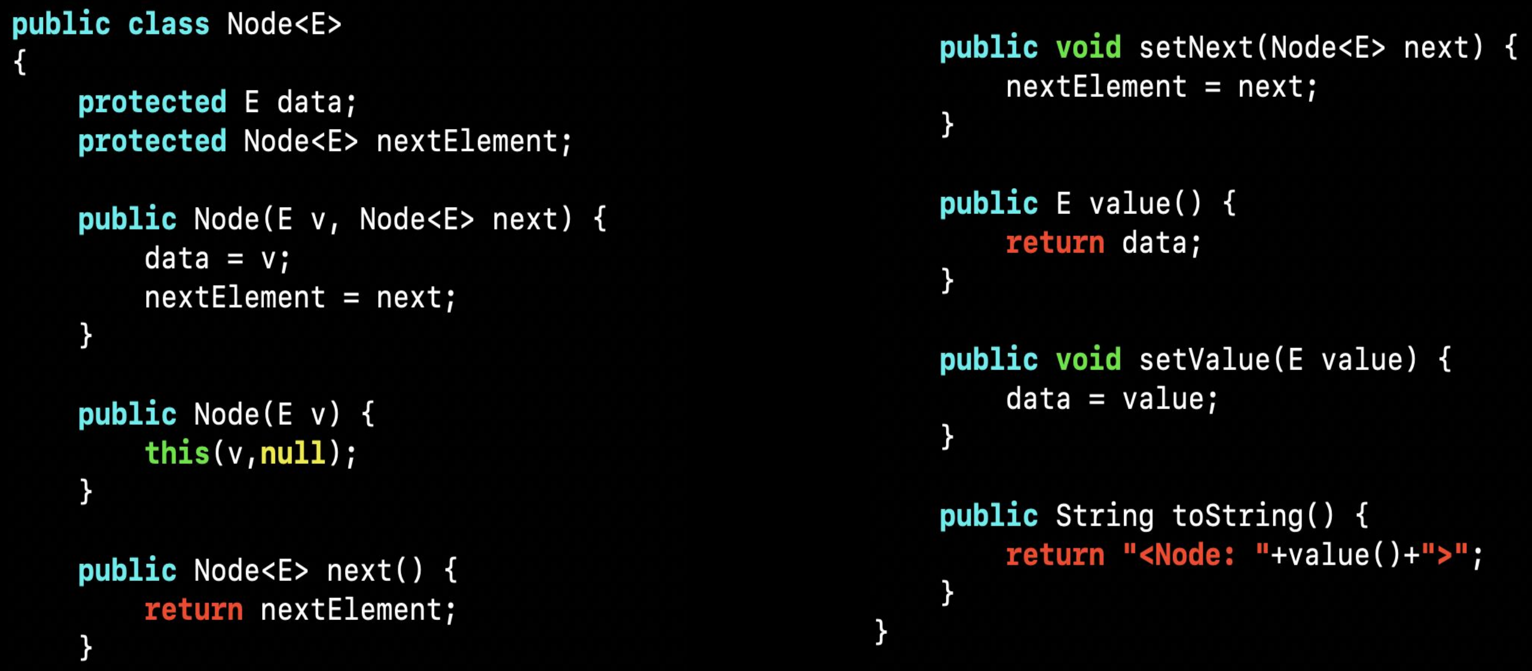


A node in a singly linked list.



A node in a doubly linked list.

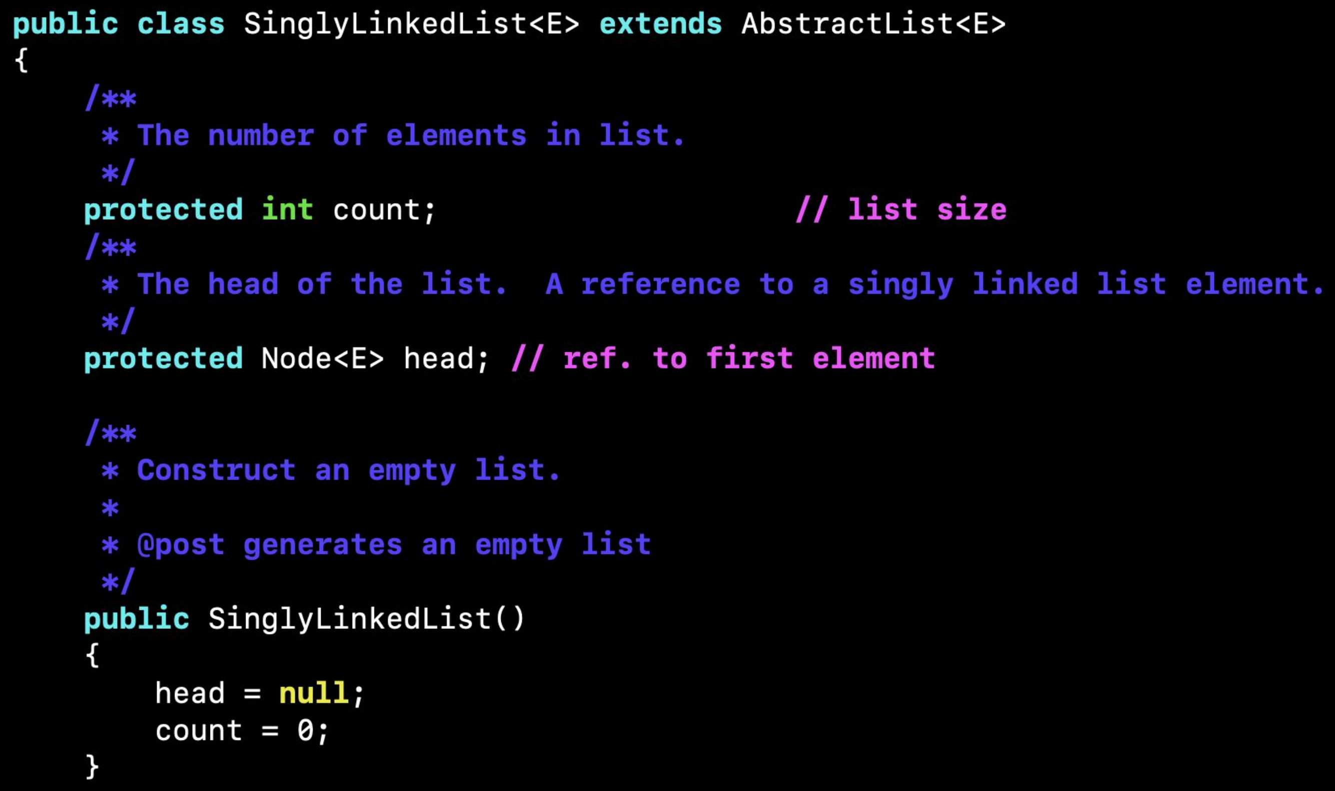
At minimum, a linked list also needs to store a reference to the first node, which is called the *head*.
It may have a reference to the last node called the *tail*. It may keep `count` of its number of nodes.



```java
public class Node<E>
{
    protected E data;
    protected Node<E> nextElement;

    public Node(E v, Node<E> next) {
        data = v;
        nextElement = next;
    }

    public Node(E v) {
        this(v,null);
    }

    public Node<E> next() {
        return nextElement;
    }

    public void setNext(Node<E> next) {
        nextElement = next;
    }

    public E value() {
        return data;
    }

    public void setValue(E value) {
        data = value;
    }

    public String toString() {
        return "<Node: "+value()+">";
    }
}
```

The `Node` class (without comments) in the `structure5` package.

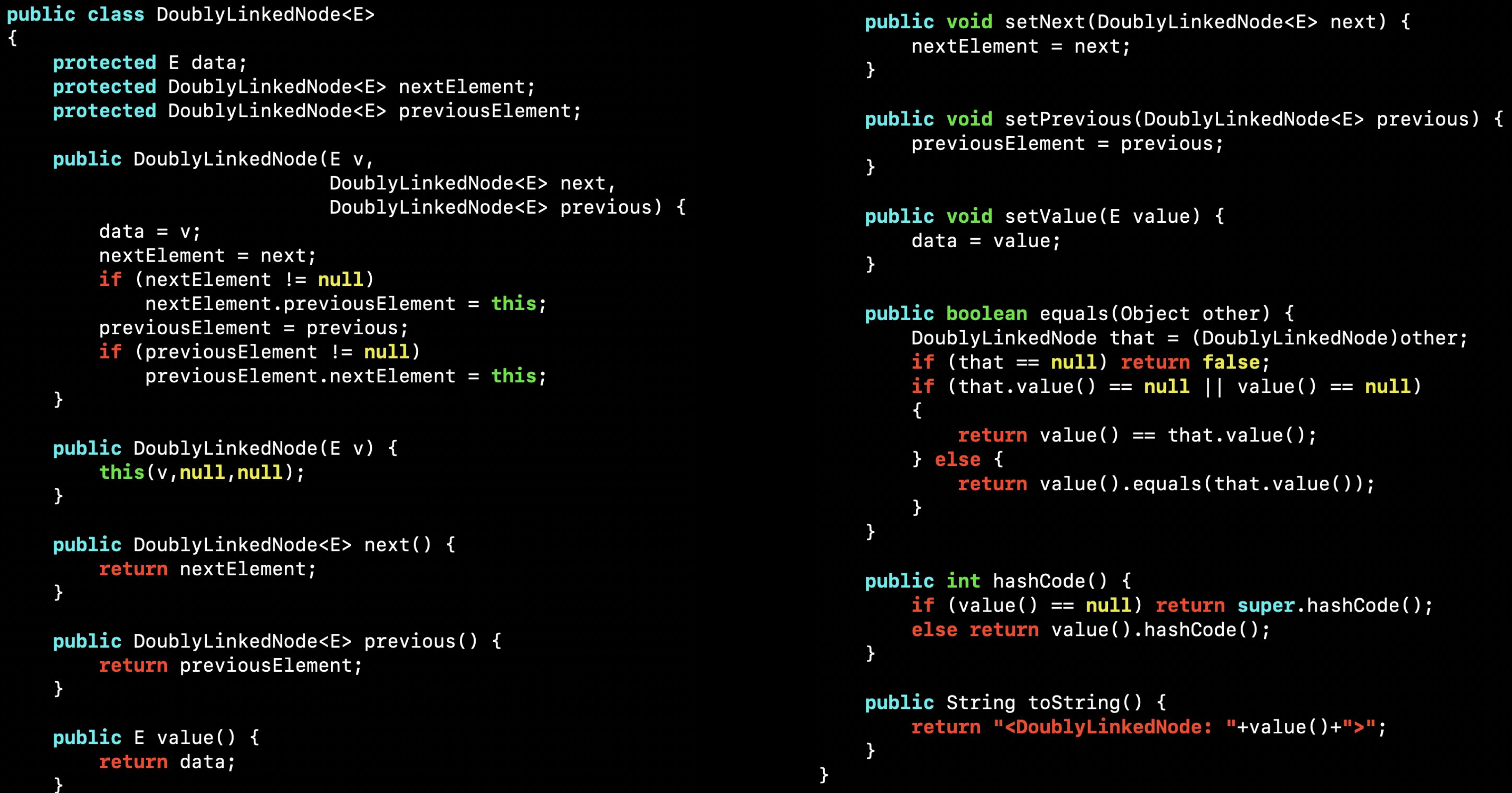
- Why are `data` and `nextElement` set to `protected`?
- What is the purpose of the accessor methods (e.g. `value()`)?
- What is the purpose of the mutator methods (e.g. `setValue(E value)`)?



```java
public class SinglyLinkedList<E> extends AbstractList<E>
{
    /**
     * The number of elements in list.
     */
    protected int count;                    // list size
    /**
     * The head of the list.  A reference to a singly linked list element.
     */
    protected Node<E> head; // ref. to first element

    /**
     * Construct an empty list.
     *
     * @post generates an empty list
     */
    public SinglyLinkedList()
    {
        head = null;
        count = 0;
    }
```

The `SinglyLinkedList` class uses the `Node` class for its nodes.

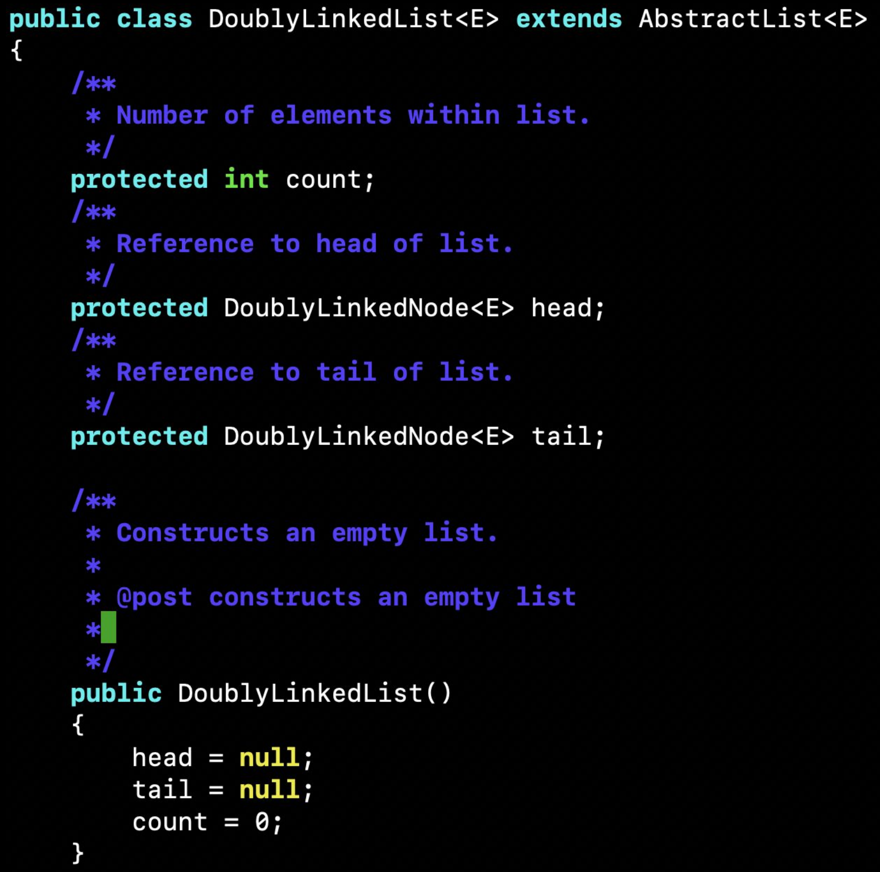
- It also keeps a `count` property.



```java
public class DoublyLinkedNode<E>
{
    protected E data;
    protected DoublyLinkedNode<E> nextElement;
    protected DoublyLinkedNode<E> previousElement;

    public DoublyLinkedNode(E v,
                            DoublyLinkedNode<E> next,
                            DoublyLinkedNode<E> previous) {
        data = v;
        nextElement = next;
        if (nextElement != null)
            nextElement.previousElement = this;
        previousElement = previous;
        if (previousElement != null)
            previousElement.nextElement = this;
    }

    public DoublyLinkedNode(E v) {
        this(v,null,null);
    }

    public DoublyLinkedNode<E> next() {
        return nextElement;
    }

    public DoublyLinkedNode<E> previous() {
        return previousElement;
    }

    public E value() {
        return data;
    }

    public void setNext(DoublyLinkedNode<E> next) {
        nextElement = next;
    }

    public void setPrevious(DoublyLinkedNode<E> previous) {
        previousElement = previous;
    }

    public void setValue(E value) {
        data = value;
    }

    public boolean equals(Object other) {
        DoublyLinkedNode that = (DoublyLinkedNode)other;
        if (that == null) return false;
        if (that.value() == null || value() == null)
        {
            return value() == that.value();
        } else {
            return value().equals(that.value());
        }
    }

    public int hashCode() {
        if (value() == null) return super.hashCode();
        else return value().hashCode();
    }

    public String toString() {
        return "<DoublyLinkedNode: "+value()+">";
    }
}
```

The `DoublyLinkedNode` class (without comments) in the `structure5` package.

- Why is the first constructor more complicated? What is it doing?
- What is `this`? How is it used in `= this;` and `this(v,null,null)`?
- Why does `equals` check `that == null` and `that.value() == null` (in that order)?



```java
public class DoublyLinkedList<E> extends AbstractList<E>
{
    /**
     * Number of elements within list.
     */
    protected int count;
    /**
     * Reference to head of list.
     */
    protected DoublyLinkedNode<E> head;
    /**
     * Reference to tail of list.
     */
    protected DoublyLinkedNode<E> tail;

    /**
     * Constructs an empty list.
     *
     * @post constructs an empty list
     *
     */
    public DoublyLinkedList()
    {
        head = null;
        tail = null;
        count = 0;
    }
```

The `DoublyLinkedList` class uses the `DoublyLinkedNode` class for its nodes.

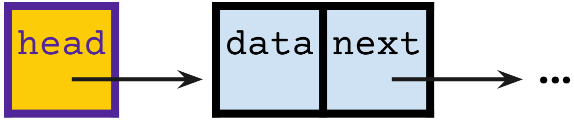
- It also keeps a `count` property.

`AddFirst`

Adding a value to the front of a singly linked list

Let's conceptualize how to add a value to the front of a singly linked list.

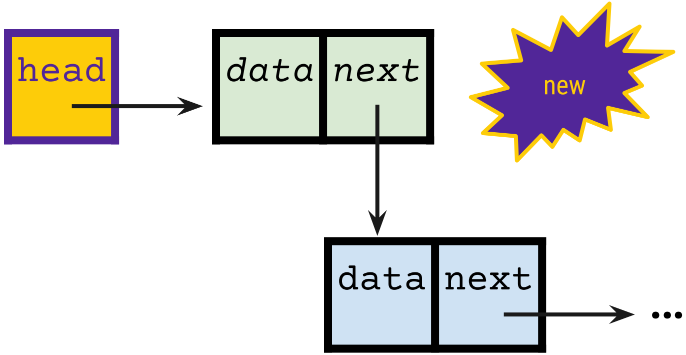
After we identify everything that needs to be done, we'll take a look at the implementation in the `structure5` package.



Before the addition.

Checklist

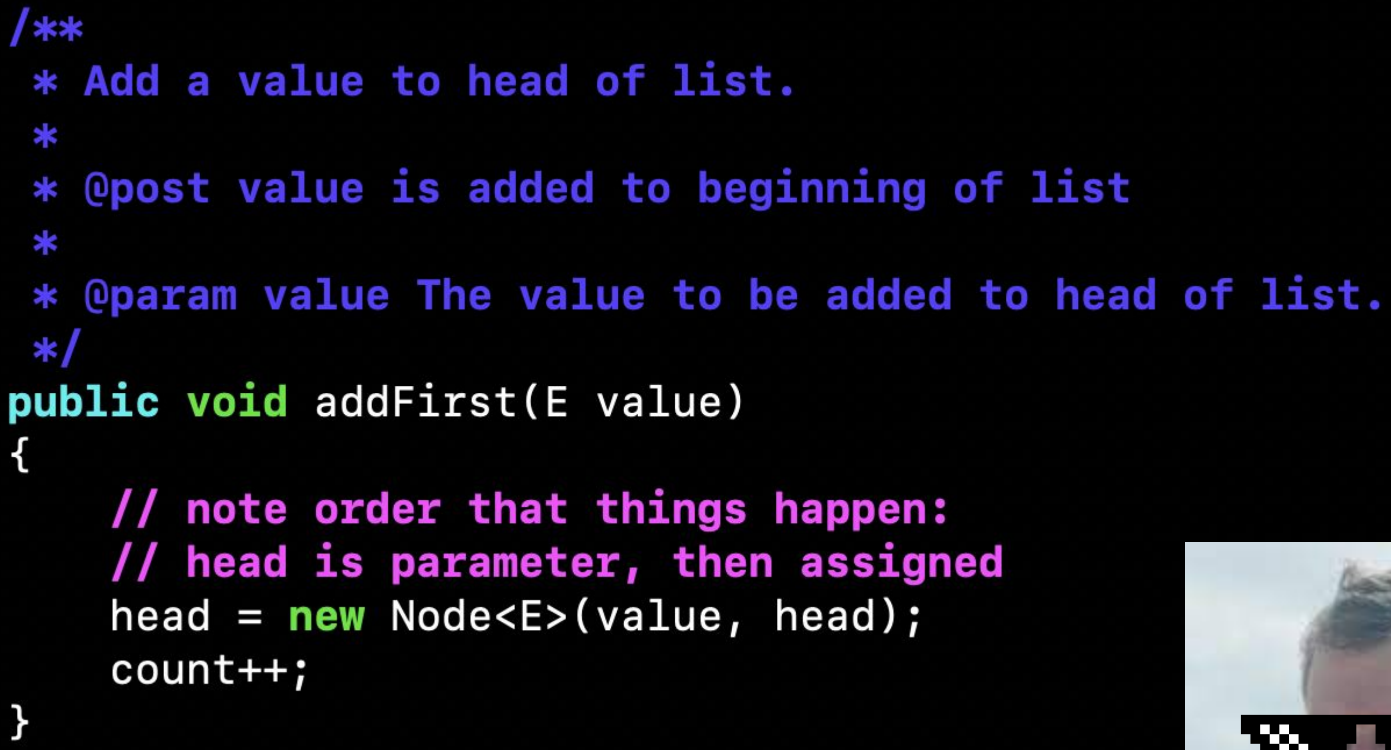
- Make a new node.
    - Set `data` to the new value.
    - Set `next` to reference the current first node.
- Update the `head` reference to the new node.
- Increment `count`.



After the addition.

Edge Cases

- What if the list is currently empty?
  Do the same steps handle this case?



```
/**
 * Add a value to head of list.
 *
 * @post value is added to beginning of list
 *
 * @param value The value to be added to head of list.
 */
public void addFirst(E value)
{
    // note order that things happen:
    // head is parameter, then assigned
    head = new Node<E>(value, head);
    count++;
}
```

The implementation is pretty nice!

- Let’s step through the Checklist again.
- Let’s also check that the Edge Case is handled property.

Adding a value to the front of a doubly linked list

Let's conceptualize how to add a value to the front of a doubly linked list.

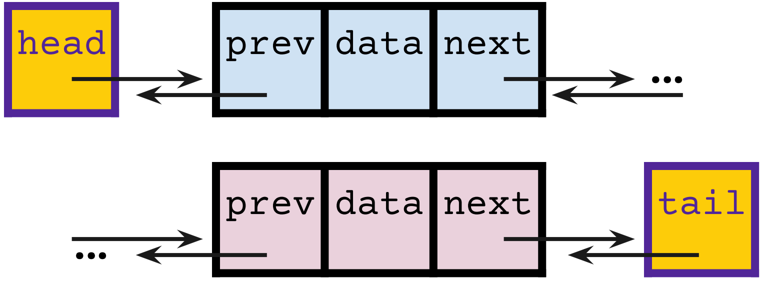
After we identify everything that needs to be done, we'll take a look at the implementation in the `structure5` package.

Checklist

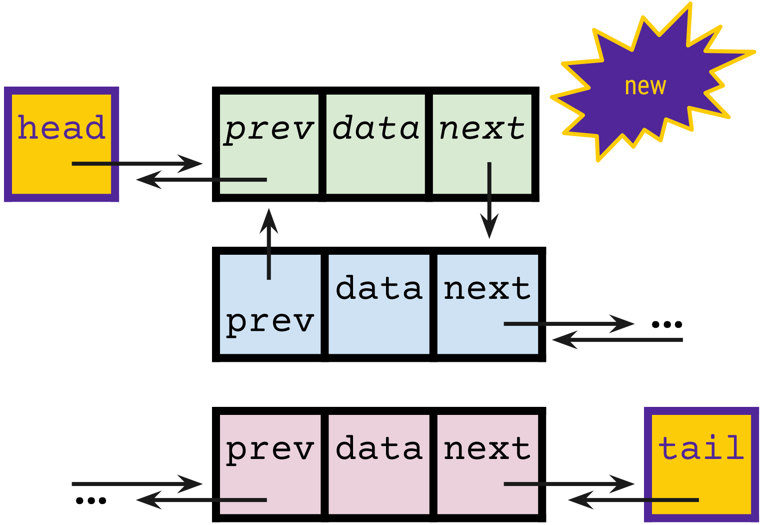
- You got this!

Edge Cases

- You got this!



Before the addition.



After the addition.

Activity: Completing the Conceptualization

Complete the steps needed for adding a value to the front of a doubly linked list.

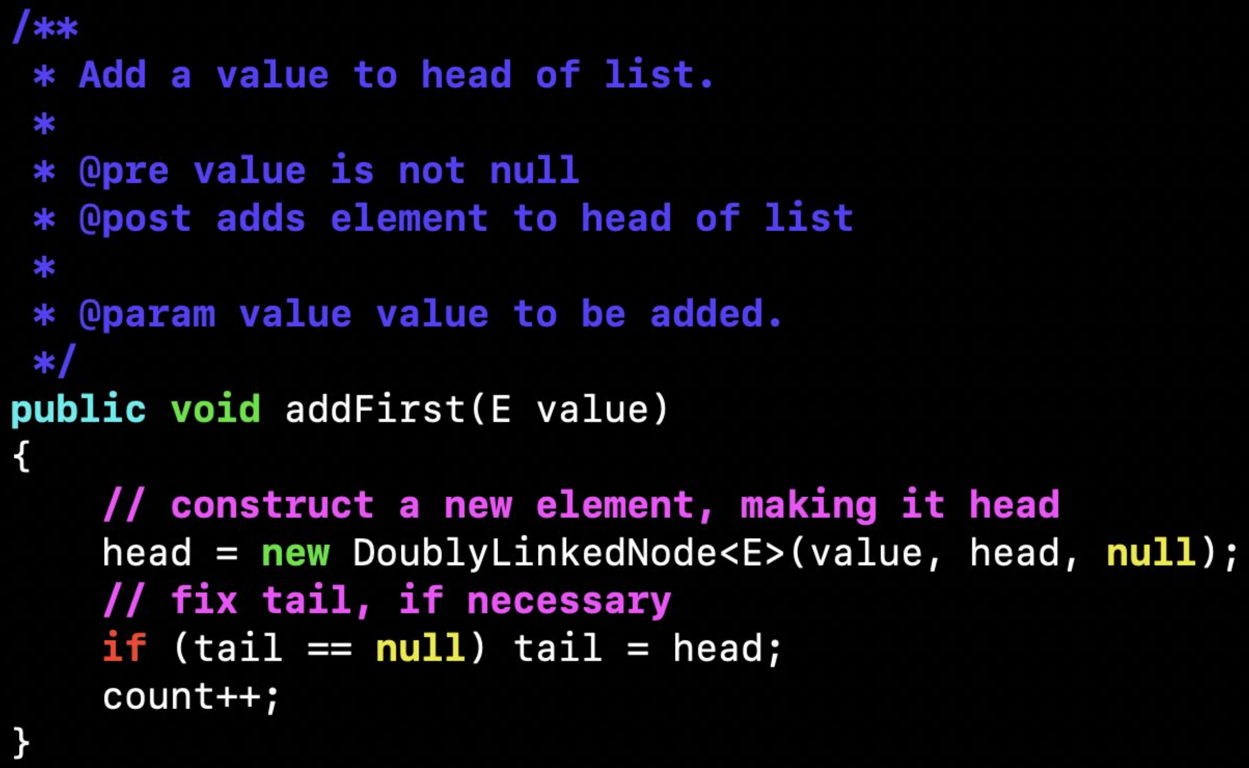
- Checklist
- Edge Cases

Hint: There is at least one new edge case to consider.

Think about this for 2 minutes.
Then discuss it with your neighbor for 3 minutes.

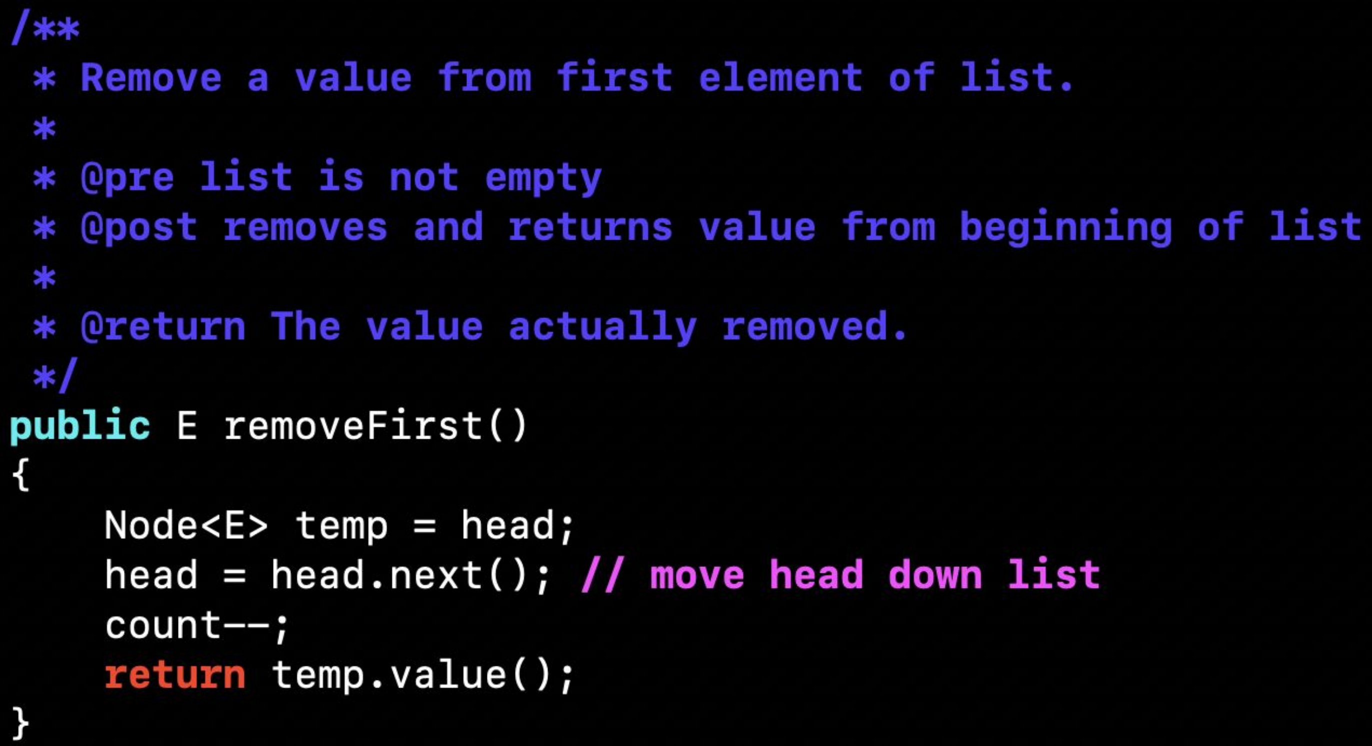
<u>Time permitting</u>

- Would this be any easier with dummy nodes?



```
/**
 * Add a value to head of list.
 *
 * @pre value is not null
 * @post adds element to head of list
 *
 * @param value value to be added.
 */
public void addFirst(E value)
{
    // construct a new element, making it head
    head = new DoublyLinkedNode<E>(value, head, null);
    // fix tail, if necessary
    if (tail == null) tail = head;
    count++;
}
```

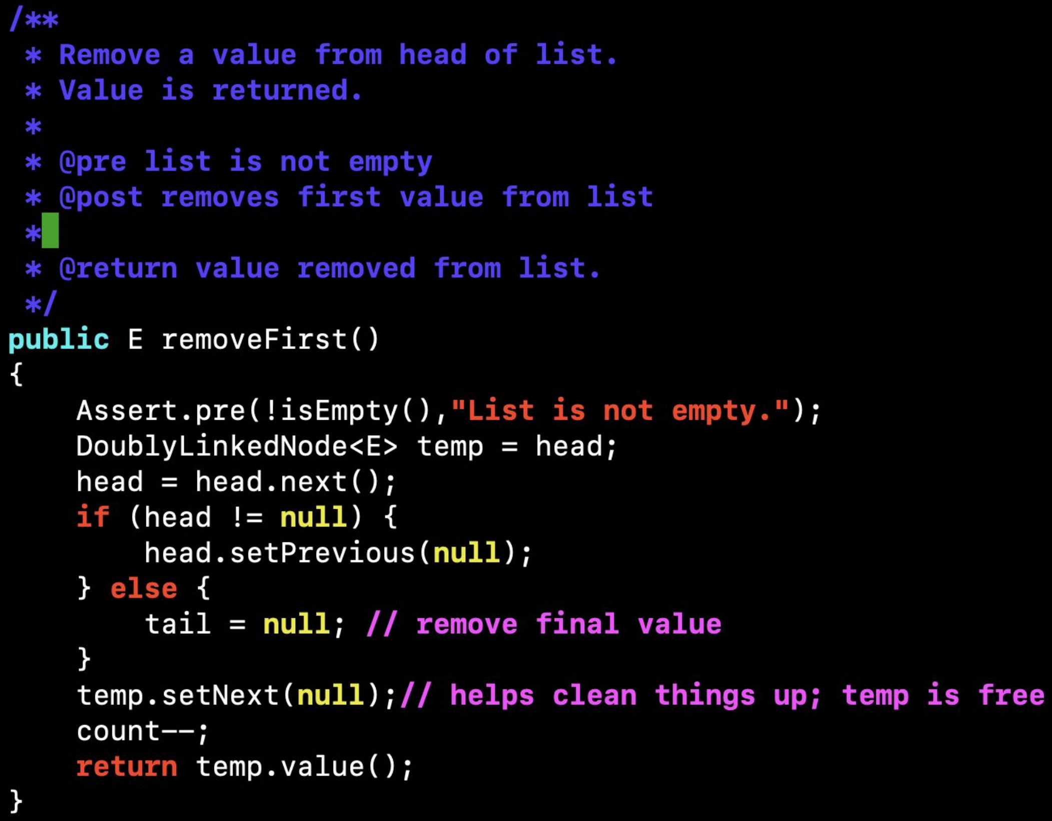
`RemoveFirst`



```
/**
 * Remove a value from first element of list.
 *
 * @pre list is not empty
 * @post removes and returns value from beginning of list
 *
 * @return The value actually removed.
 */
public E removeFirst()
{
    Node<E> temp = head;
    head = head.next(); // move head down list
    count--;
    return temp.value();
}
```

`removeFirst` in `SinglyLinkedList`

- Any surprises?
- What happened to the node that was removed?



```
/**
 * Remove a value from head of list.
 * Value is returned.
 *
 * @pre list is not empty
 * @post removes first value from list
 *
 * @return value removed from list.
 */
public E removeFirst()
{
    Assert.pre(!isEmpty(),"List is not empty.");
    DoublyLinkedNode<E> temp = head;
    head = head.next();
    if (head != null) {
        head.setPrevious(null);
    } else {
        tail = null; // remove final value
    }
    temp.setNext(null);// helps clean things up; temp is free
    count--;
    return temp.value();
}
```

`removeFirst` in `DoublyLinkedList`

- Any surprises?

`addLast` and `removeLast`

Activity: Conceptualizing `addLast`

Try drawing a diagram for the `addLast` method in a singly linked list.

- If you finish, then do the same for a doubly linked list.
- If you finish, then do the same for `removeLast`.

Think about this for 3 minutes.
Then we'll discuss it as a group.

Time permitting

- Look at the code together.