Recursive Data Types, **`null`**, and Linked Lists

Lecture 16

UNC DEPARTMENT OF COMPUTER SCIENCE

Data Buddies Survey

CRA Computing Research Association

Undergraduate Survey
http://bit.ly/CSundergraduate

Graduate Survey
http://bit.ly/CSgraduate

What is it?

- Anonymous survey provided by CRA open now through Oct. 31st

Why is it important?

- Your feedback gives department real-time data on curriculum, pedagogy, student support and cultural climate from student POV

What's in it for you?

- Completion of survey means raffle entry and chance to win Amazon gift card *(dept to raffle more than $1K in gift cards)*

Check your email for more details

Announcements

- Women and Minorities in CS – Info Session & Discussion
  - Monday 10/23 at 5pm in SN011

- PS3 Due Tonight

- Wednesday (tomorrow) Office Hours Close at 5pm for Fall Break
  - No review session

Warm-up on References

Compound Data Type Properties

- So far we've focused on classes with value-type properties, such as:
  - string
  - number
  - boolean
- Properties can also be reference types, like:
  - arrays
  - objects

```
class Person {
    name: string;
    pets: Dog[];
}

class Dog {
    name: string;
    breed: string;
}
```

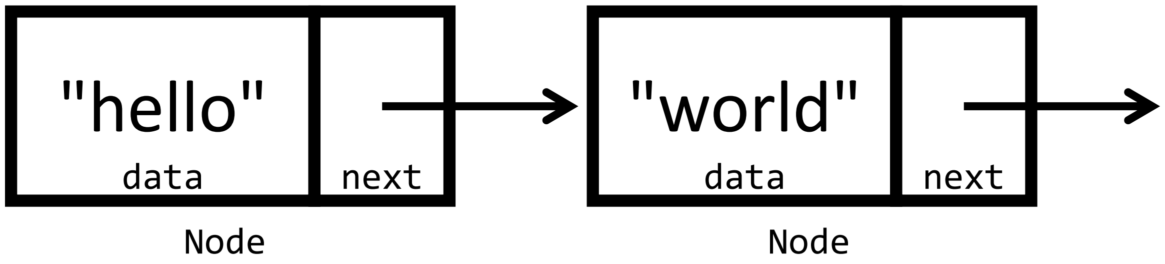
Recursive Data Types

- Properties can refer to other objects of the same type

- Notice the class **Node** left. It has a property named **next** and its value must be... *another Node*.

- This allows us to form a Linked List or a "chain" of Node objects.

```
class Node {
   data: string;
   next: Node;
}
```

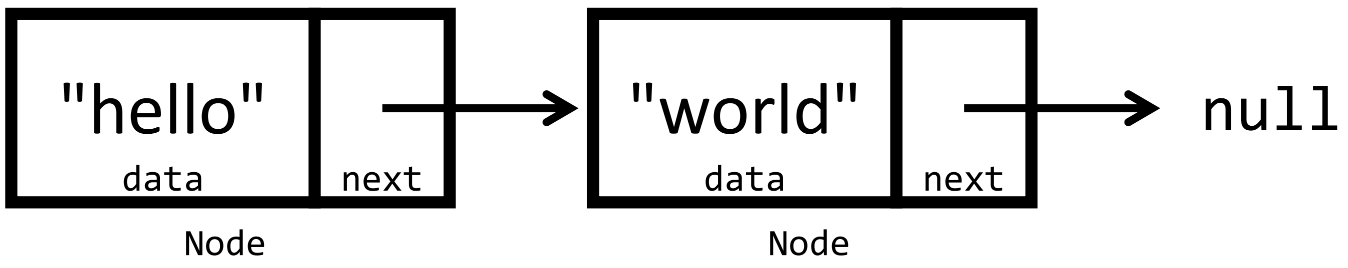


Linked List

- A classic, simple data structure in Computer Science

- Formed by chaining together a sequence of Node objects
  - The first node is referred to as the **head**

- We will work with it today for pedagogical purposes in exploring
  - What is **`null`**?
  - More advanced uses of references
  - Recursive data types



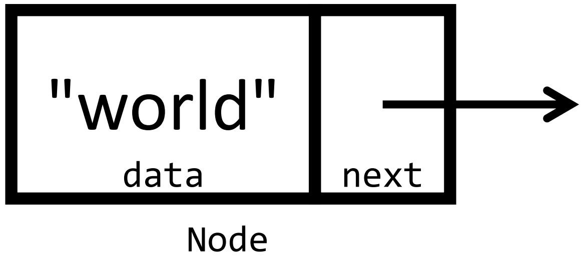
Where does a Linked List end?

- In the class to the right, if a Node refers to a `next` Node, and the `next` `Node` refers to another `next` Node, how does it end?

- In a linked list, the very last Node's `next` value will refer to **`null`** which means nothing or "there is no next Node"

- To permit the next Node to be a null value, TypeScript requires us to give it a special type (next slide)

```
class Node {
   data: string;
   next: Node;
}
```



Possibly **`null`** References (1/3)

- Because the next property may refer to another **Node** *OR* **`null`**, the type of next is specified as:

  `Node | null`

- Notice this looks like the boolean OR operator || but is only a single vertical bar.

- Now our linked list's last **Node** can be assigned a **`null`** reference.

```
class Node {
   data: string;
   next: Node | null;
}
```

Possibly **`null`** References (2/3)

```
let aNode: Node = new Node();
        [ts] Object is possibly 'null'.
        (property) Node.next: Node | null
print(aNode.next.data);
```

- TypeScript helps us avoid errors when we declare the next property's type to be Node `| null`
- Notice in the example left, we are prevented from accessing the data property of the next Node because the next node is possibly null.
- Other languages (like Java) do not provide this protection which gives way to "null pointer" errors.

Handling possibly **`null`** References (3/3)

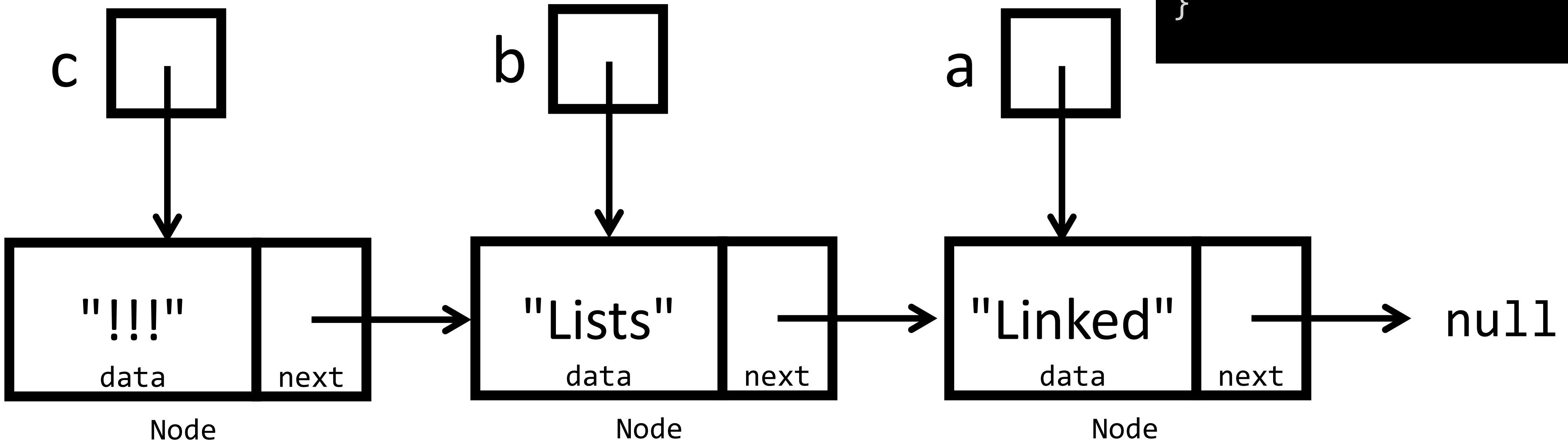
- Whenever you access a reference whose type is "<class> | null", you must test to ensure the value is not null in order to access it.

```
let aNode: Node = new Node();
if (aNode.next !== null) {
   print(aNode.next.data);
}
```

- The above code compiles because TypeScript is smart enough to know that inside of this if-then statement `aNode.next` cannot be `null`.

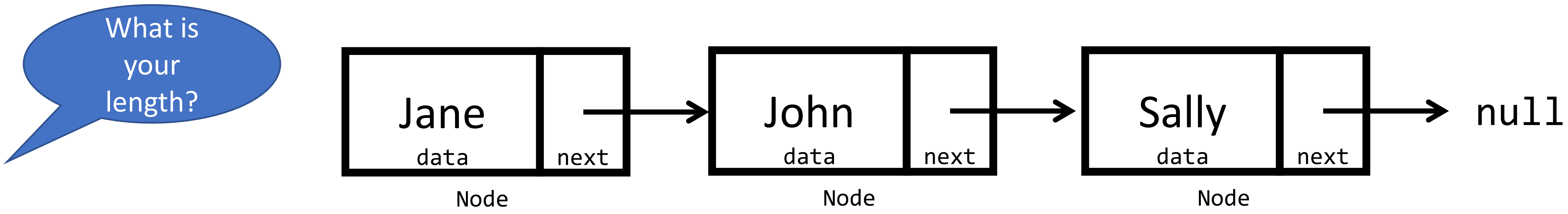
Follow-Along: Constructing a Linked List

```
let a: Node = new Node();
a.data = "Linked";
a.next = null;

let b: Node = new Node();
b.data = "Lists";
b.next = a;

let c: Node = new Node();
c.data = "!!!";
c.next = b;

if (c.next !== null && c.next.next !== null) {
    print(c.data);
    print(c.next.data);
    print(c.next.next.data);
}
```

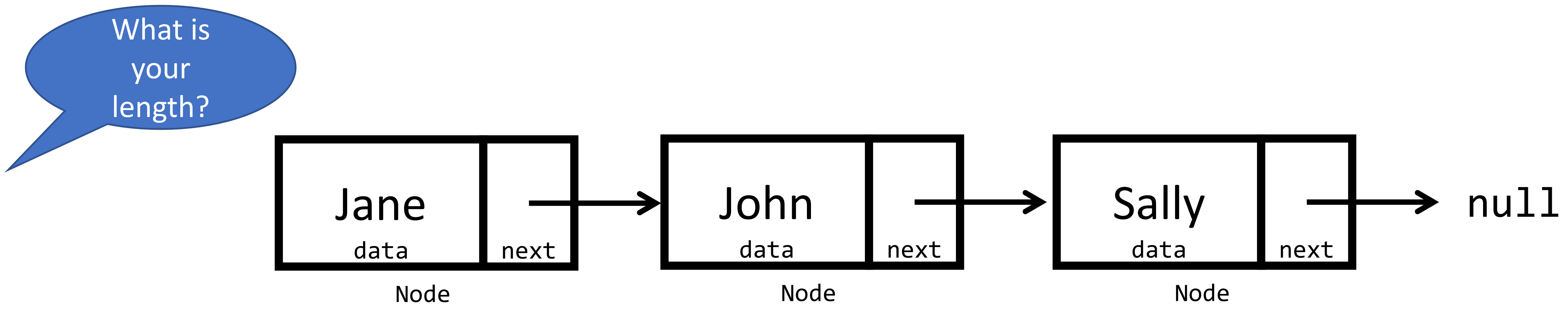


Forming a Human Linked List

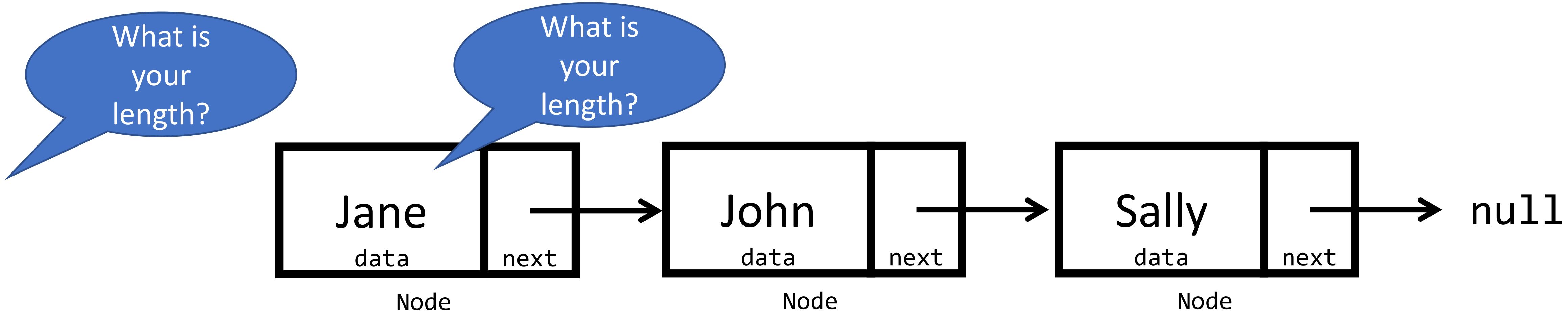
- Live demo with UTAs as Nodes
- The first Node will be the "tail" and its next is null
- Subsequent Nodes will place one hand on the next Node's shoulder



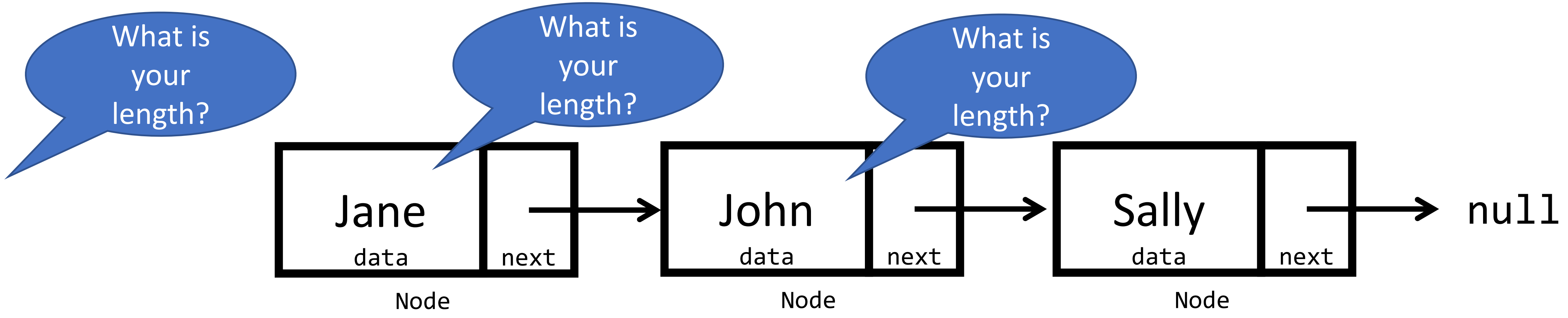
Finding the length of a Human Linked List



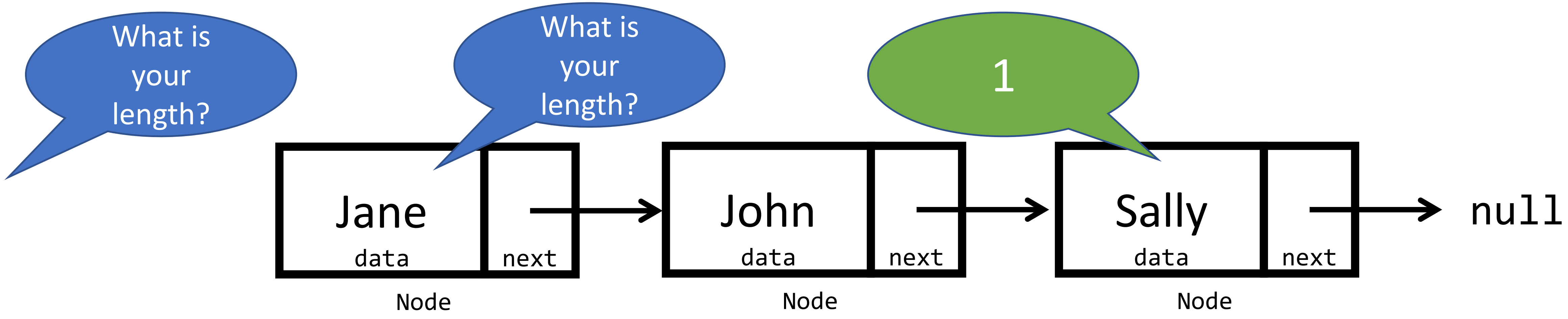
Finding the length of a Human Linked List



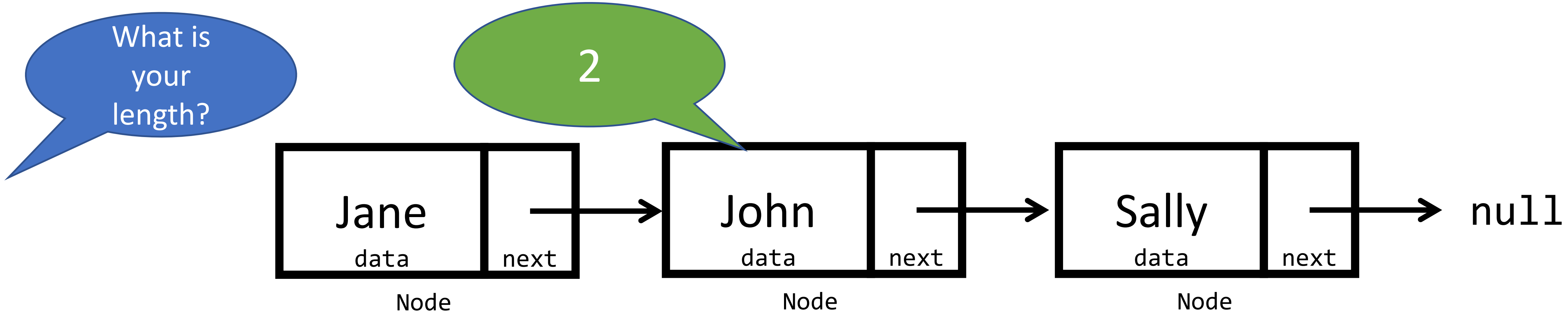
Finding the length of a Human Linked List



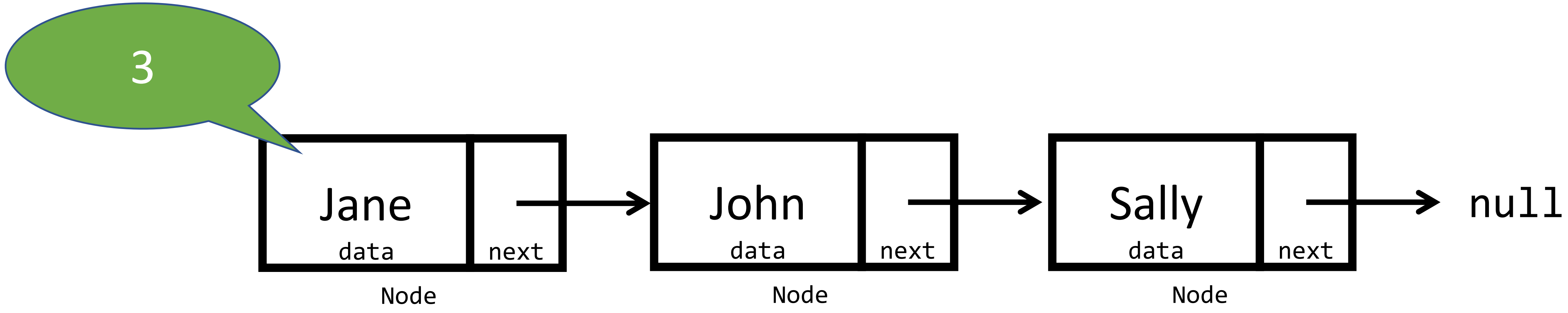
Finding the length of a Human Linked List



Finding the length of a Human Linked List



Finding the length of a Human Linked List



Hands-on: Implement the Length Function

1. Open lec16's list-functions.ts and 01-list-playground-app.ts
2. If the node's **next** property is a **`null`** reference, return 1
3. Otherwise, return 1 + the result of calling length with the **next** node
4. Check-in when your length function is properly implemented
5. Done? Try adding an additional Node to the list in the playground's makeList function

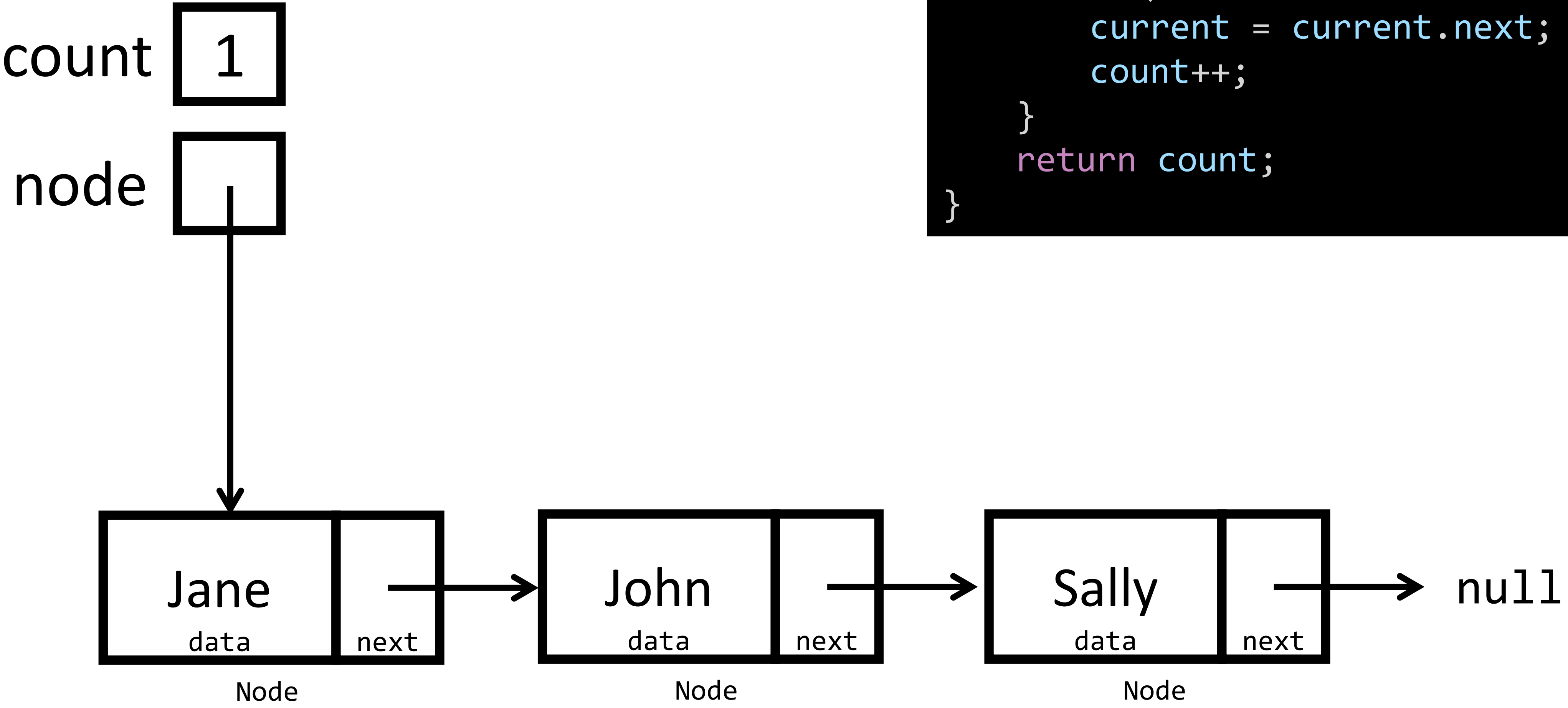
```
export function length(head: Node): number {
    if (head.next === null) {
        return 1;
    } else {
        return 1 + length(head.next);
    }
}
```

Finding the length imperatively with a loop

- How might we address this without recursion?
- We can walk the list Node-by-Node until the next node is null and count up by 1 each time.
- This is an example of when a single reference-type variable is reassigned to many different objects

```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```

Finding the length imperatively

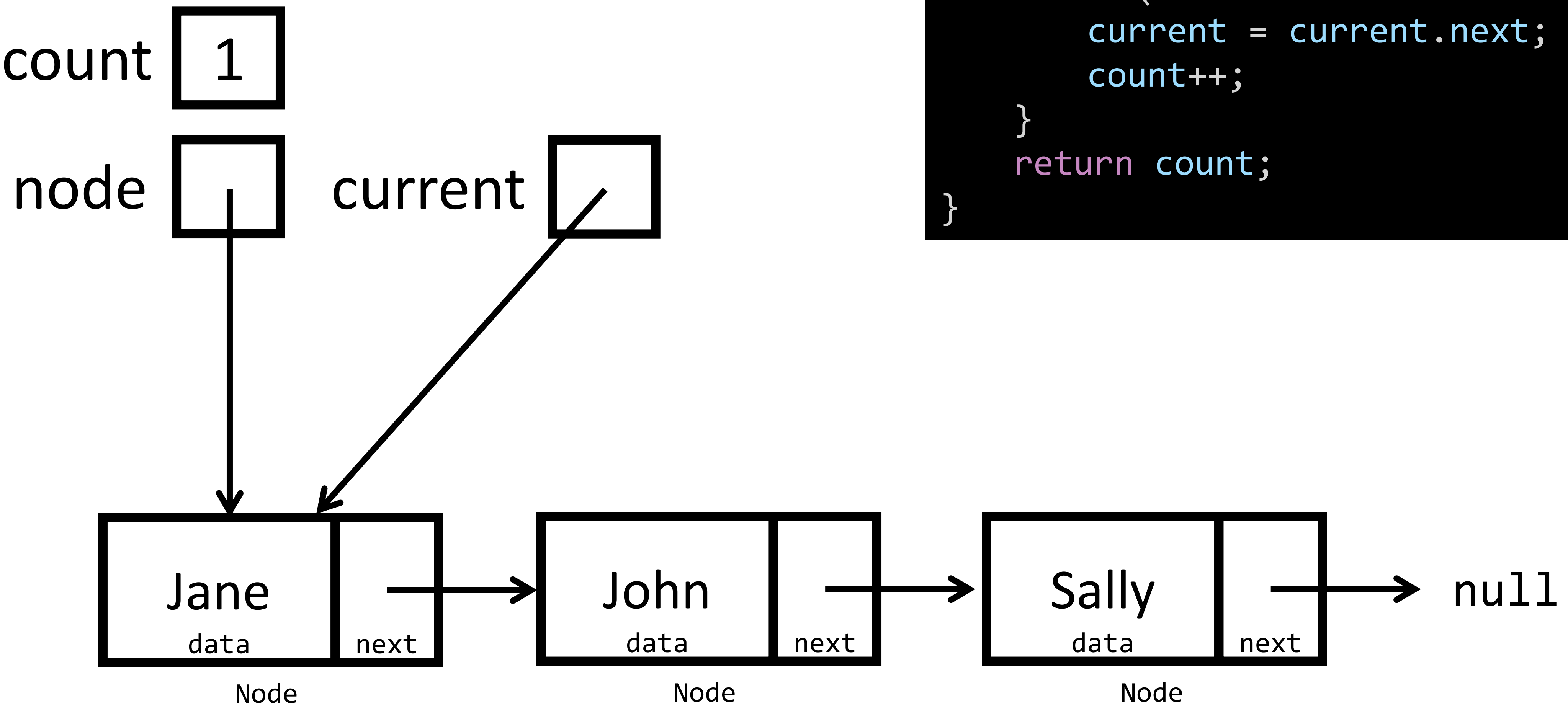



```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```

We enter the function with node referring to the first node in our list. The count variable is initialized to 1.

Finding the length imperatively

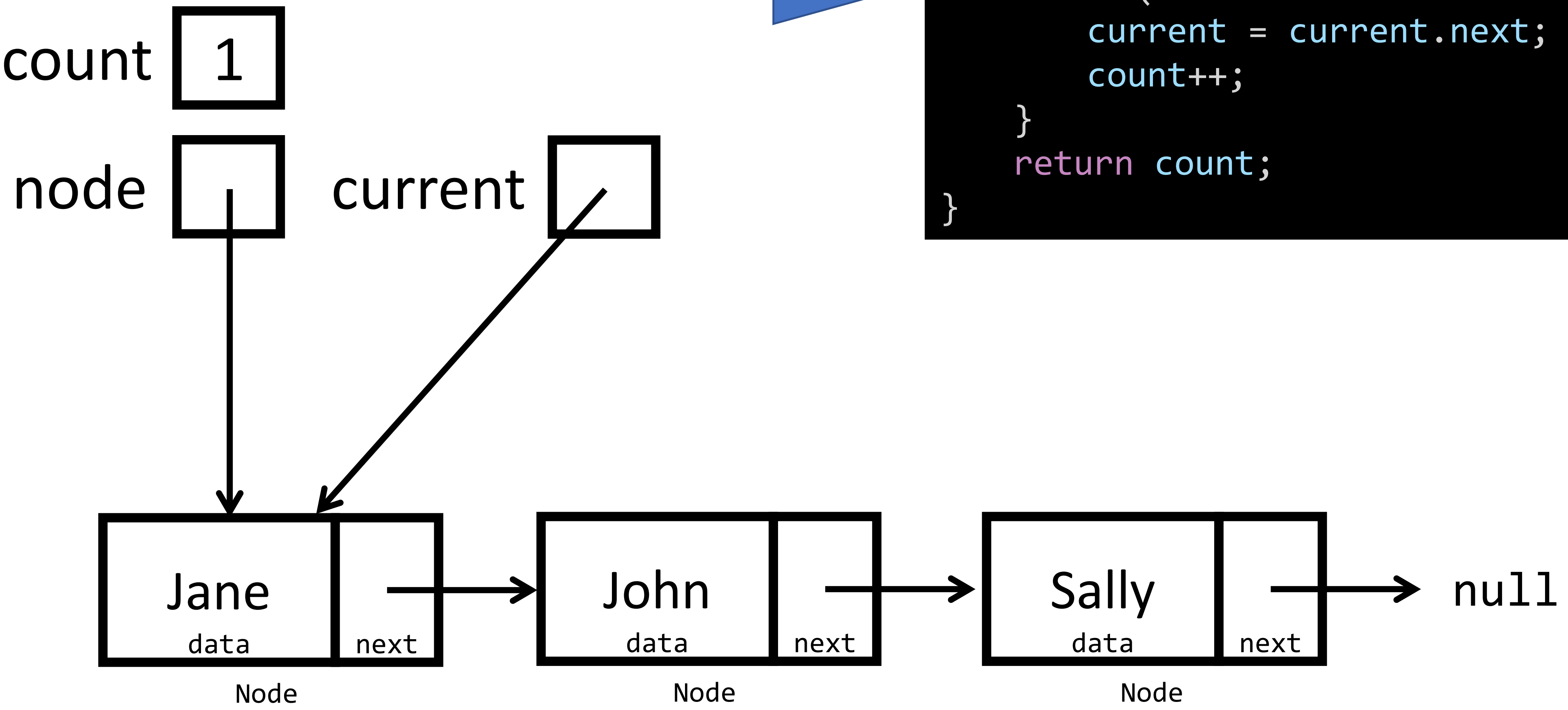
```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```



The current variable is setup and assigned a reference to the same object as the node parameter refers to.

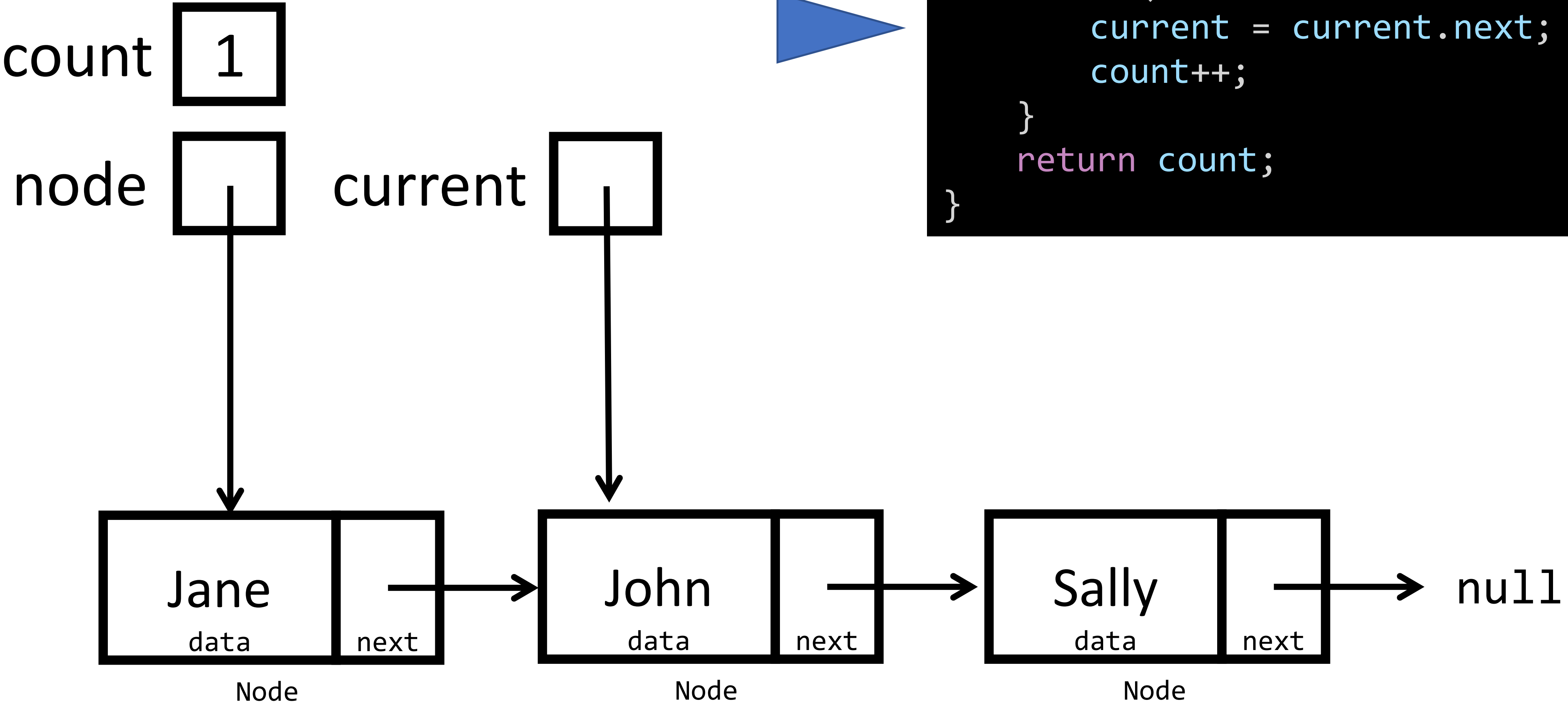
Finding the length imperatively

```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```



The current node's next reference is not null, so we'll enter the while loop next.

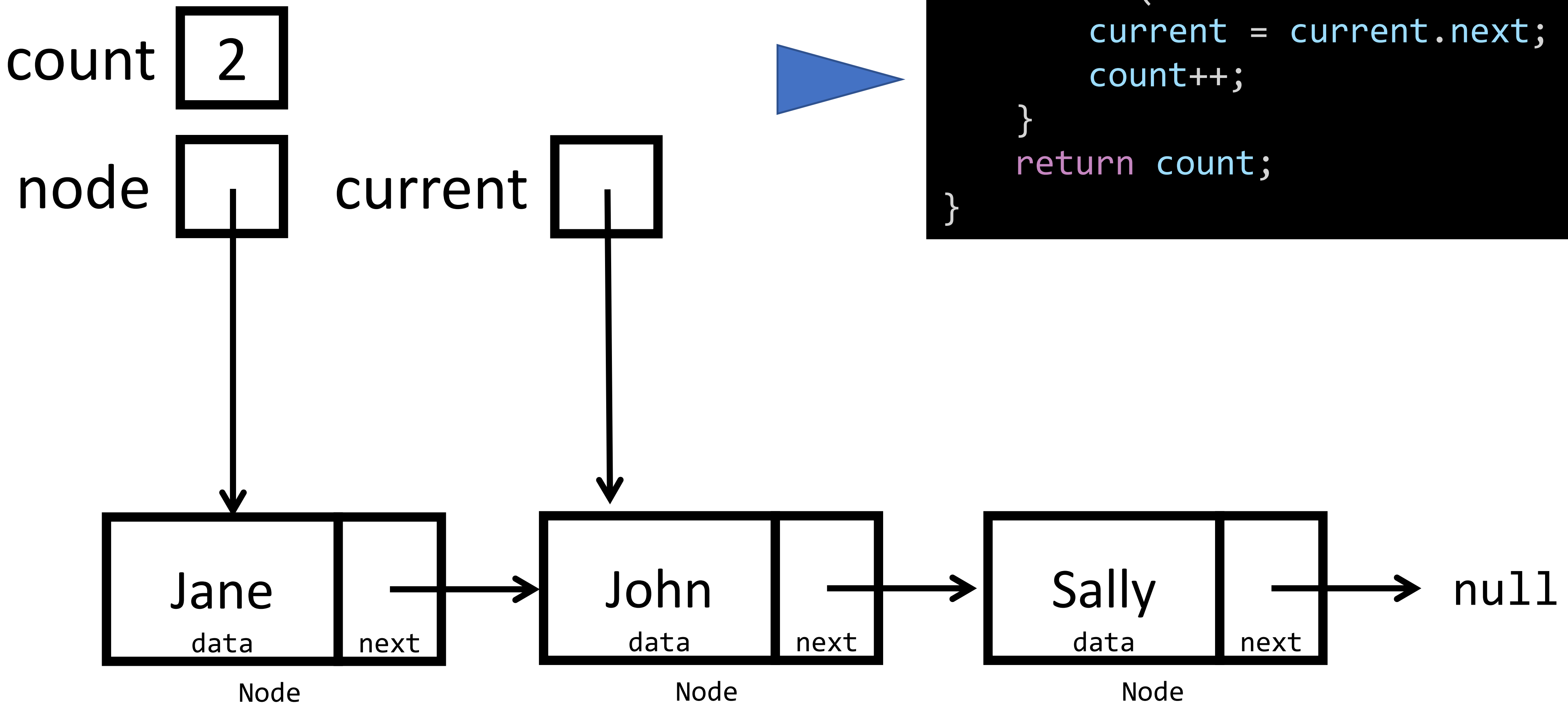
Finding the length imperatively




```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```

The current Node variable is reassigned to be its next Node (the 2nd node in this case).

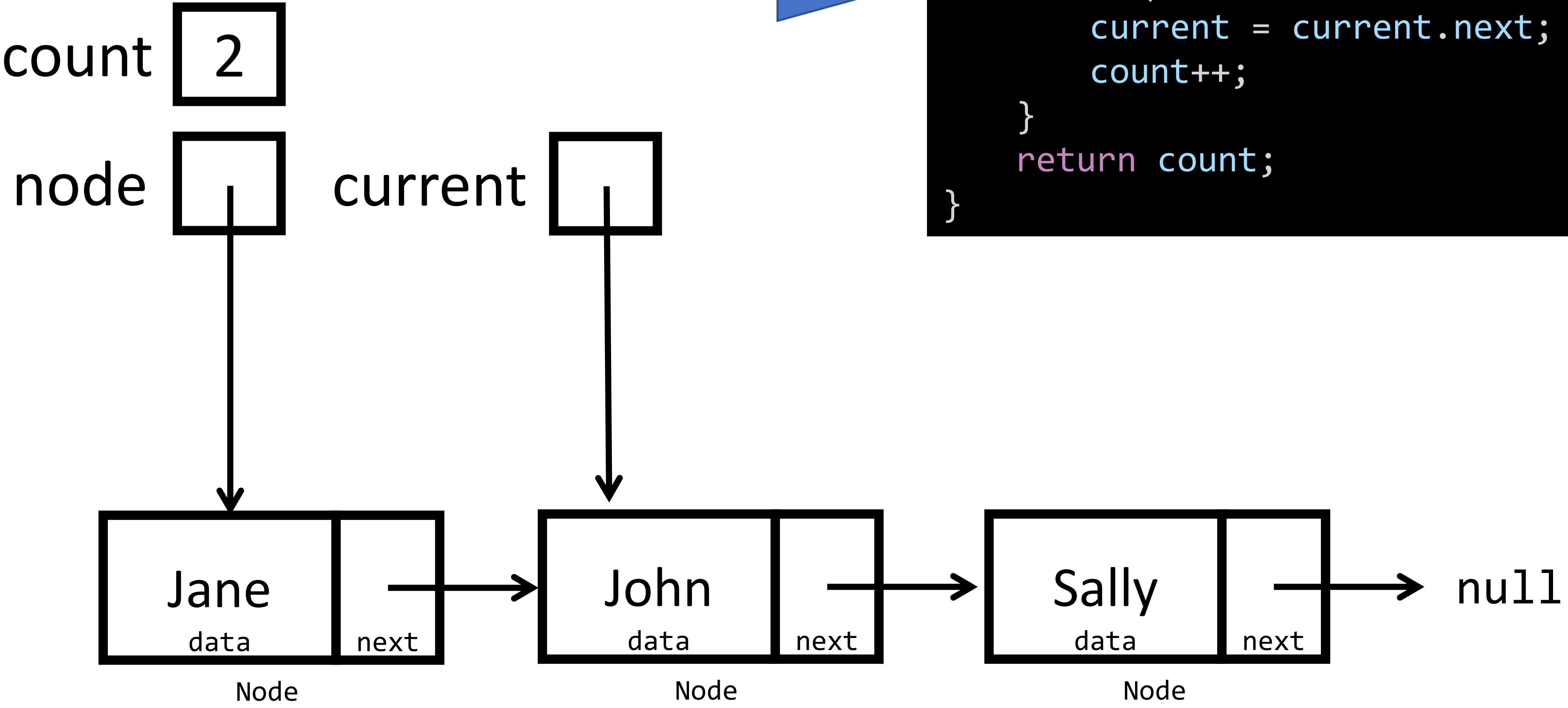
Finding the length imperatively




```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```

The count variable is incremented because we have walked to another node.

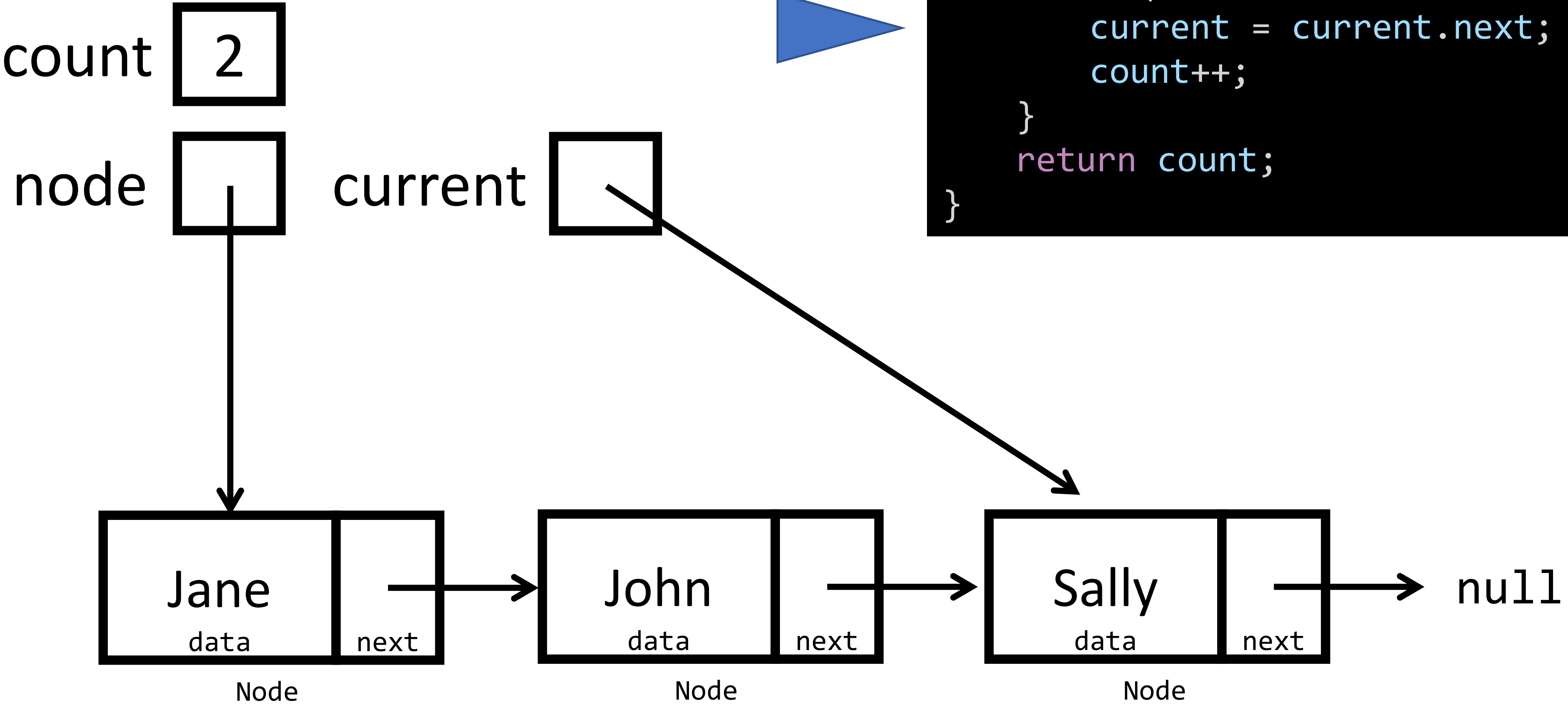
Finding the length imperatively




```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```

The while condition is tested again. Is the current Node's next property null? No, so we enter the loop.

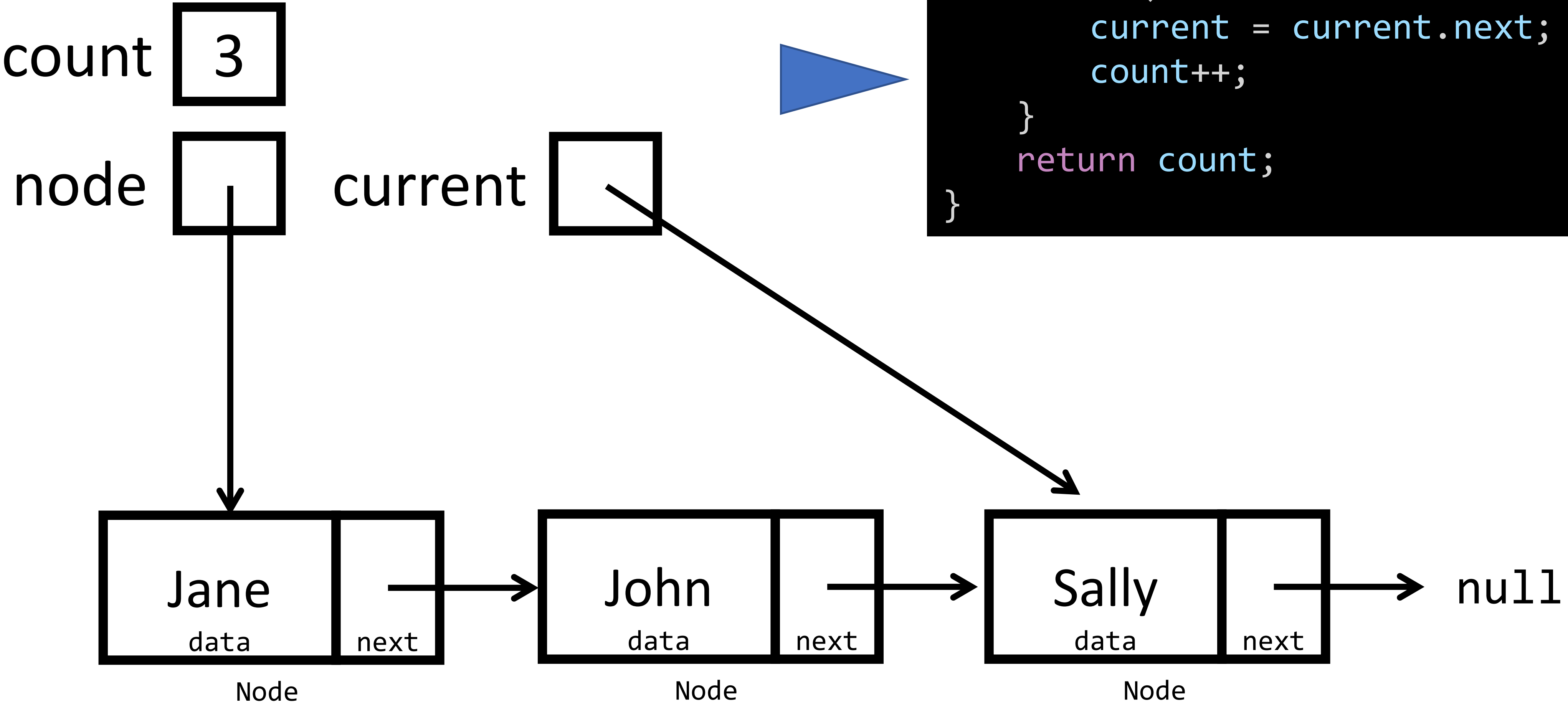
Finding the length imperatively




```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```

Current is reassigned to be a reference to its next Node.

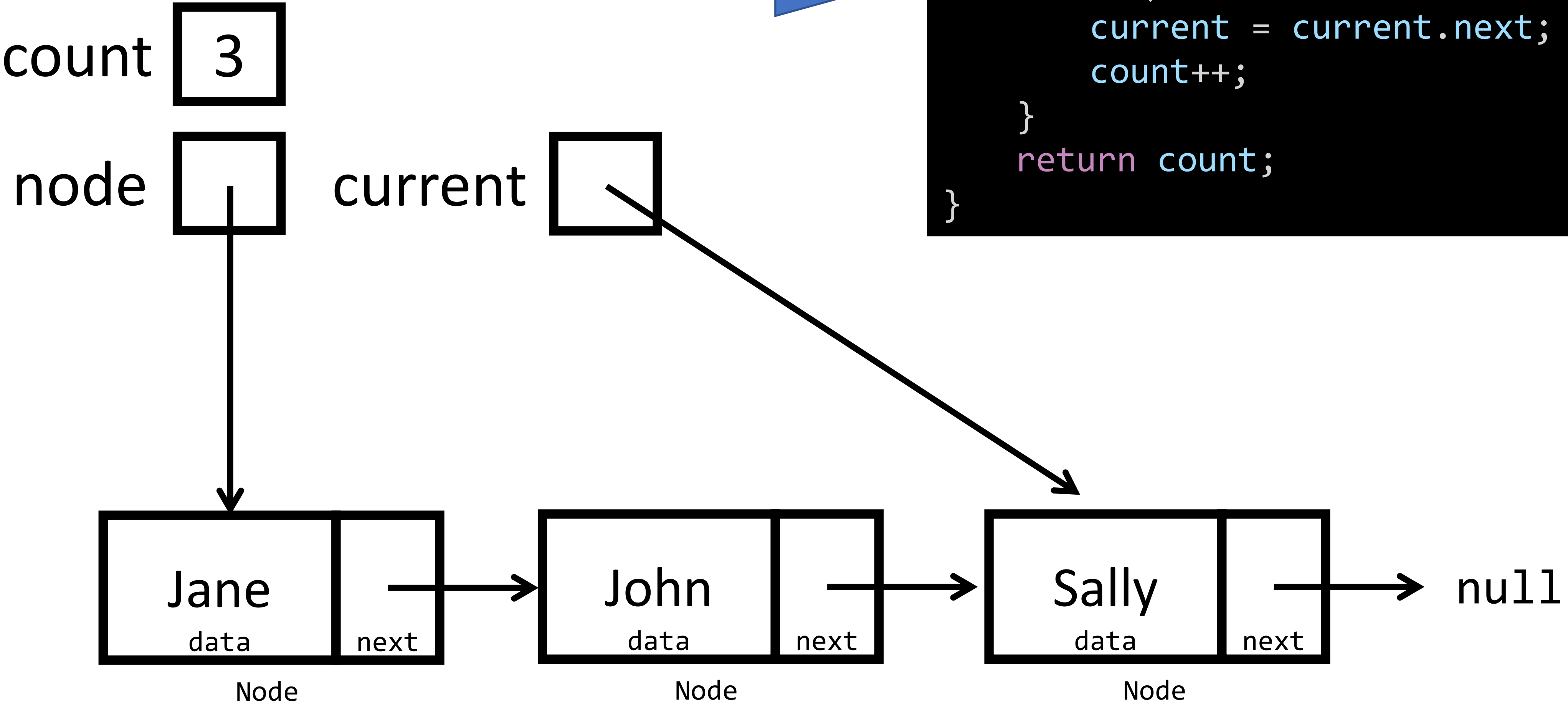
Finding the length imperatively




```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```

The count variable is incremented by 1.

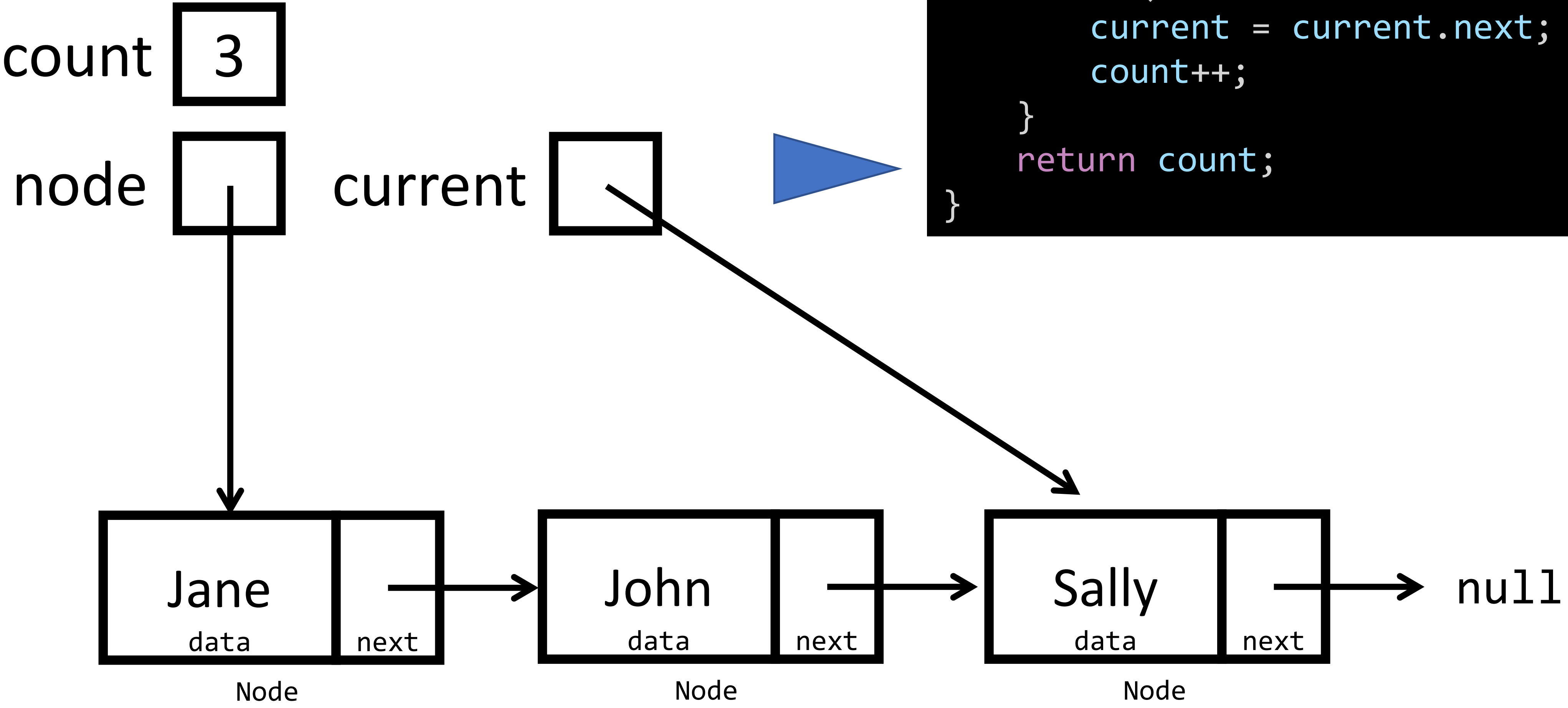
Finding the length imperatively




```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```

We test again. Is the current Node's next property null? Yes, so we do not loop again.

Finding the length imperatively



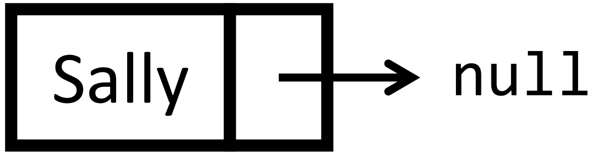

```
function lengthLoop(node: Node): number {
    let count: number = 1;
    let current: Node = node;
    while (current.next !== null) {
        current = current.next;
        count++;
    }
    return count;
}
```

Finally, count is returned to the caller of lengthLoop.

Generating a string Representation of the List

- Let's write a function with the following requirements.

- When we call with a **tail** node (*its next property is null*), it should return the node's data concatenated with "-> null". For example:

`toString(` 

`)` returns **"Sally -> null"**

- When we call with a non-tail node, it should ultimately return a string with every node's data separated by -> arrows and end with null.

`toString(` 

`)` returns **"John -> Sally -> null"**

Hands-on: Generating a string Representation

1. In 01-list-playground's main, try:

   ```
   print(toString(list));
   ```

2. In the toString function of list-functions.ts…

3. If the node's next property is null, use concatenation to return:
   ```
   "<node's value> -> null"
   ```

4. Otherwise, use concatenation and recursion to return:
   ```
   "<node's value> -> <toString of the next node>"
   ```

5. Check-in on PollEverywhere when complete

```
export function toString(node: Node): string {
    if (node.next === null) {
        return node.data + " -> null";
    } else {
        return node.data + " -> " +
        toString(node.next);
    }
}
```

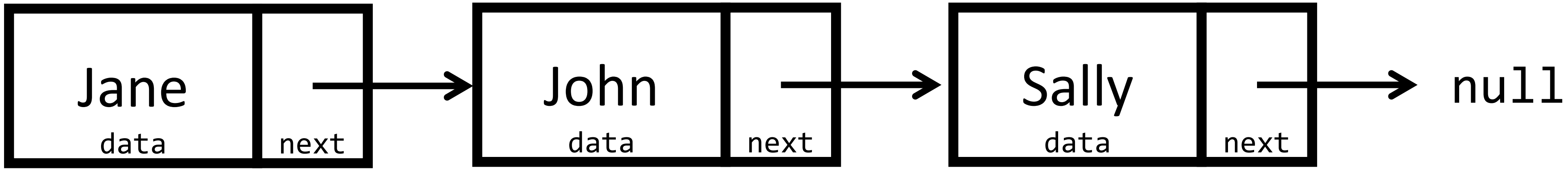
Accessing the **n**th Index

- With an array we can return the 0-indexed nth element using:
  - a[0], a[1], and so on…

- Let's implement a **`get`** function with the same semantics and the following signature:

```
get(node: Node, i: number): string | null
```

- For example:
  - `get(list, 0)` returns "Jane"
  - `get(list, 2)` returns "Sally"
  - `get(list, 40)` returns null



Accessing the **n**th Index: There are *2* **base cases**

1. Is i === 0?
    1. Yes: we've found the i^{th} Node!

2. No, i > 0? Is the next node null?
    1. Yes: there is no i^{th} Node!

- No, the next node is not null. So let's try calling get again on the *next* node and subtract 1 from i.

Follow-along: Implementing the **get** function

```
export function get(node: Node, i: number): string | null {
    if (i === 0) {
        return node.data;
    } else if (node.next === null) {
        return null;
    } else {
        return get(node.next, i - 1);
    }
}
```

Challenge Functions

- Copying a List
- Appending to a List
- Reversing a List
- Want a challenge? Try implementing the remaining functions in list-functions.ts

Challenge function solutions

copy

```
export function copy(node: Node): Node {
    if (node.next === null) {
        return node;
    } else {
        return link(node.data, copy(node.next));
    }
}
```

append

```typescript
export function append(data: string, node: Node): void {
    if (node.next === null) {
        let tail: Node = new Node();
        tail.data = data;
        node.next = tail;
    } else {
        append(data, node.next);
    }
}
```

reverse

```
export function reverse(node: Node): Node {
    if (node.next === null) {
        return node;
    } else {
        let reversed: Node = reverse(node.next);
        node.next.next = node;
        node.next = null;
        return reversed;
    }
}
```