The List Abstract Data Type

Computer Science 112
Boston University

Christine Papadakis-Kanaris

Abstract Data Types

- An *abstract data type* (ADT) is a model of a data structure that specifies:
  - the characteristics of the collection of data
  - the operations that can be performed on the collection

Specifying an ADT Using an Interface

- In Java, we can use an interface to specify an ADT:

```
public interface List {
    Object getItem(int i);
    boolean addItem(Object item, int i);
    Object removeItem(int i);
    int length();
    boolean isFull();
}
```

- An interface specifies a set of methods.
  - includes only their headers
  - does *not* typically include the full method definitions

Implementing an ADT Using a Class

- To implement an ADT, we define a class.
- We specify the corresponding interface in the class header:

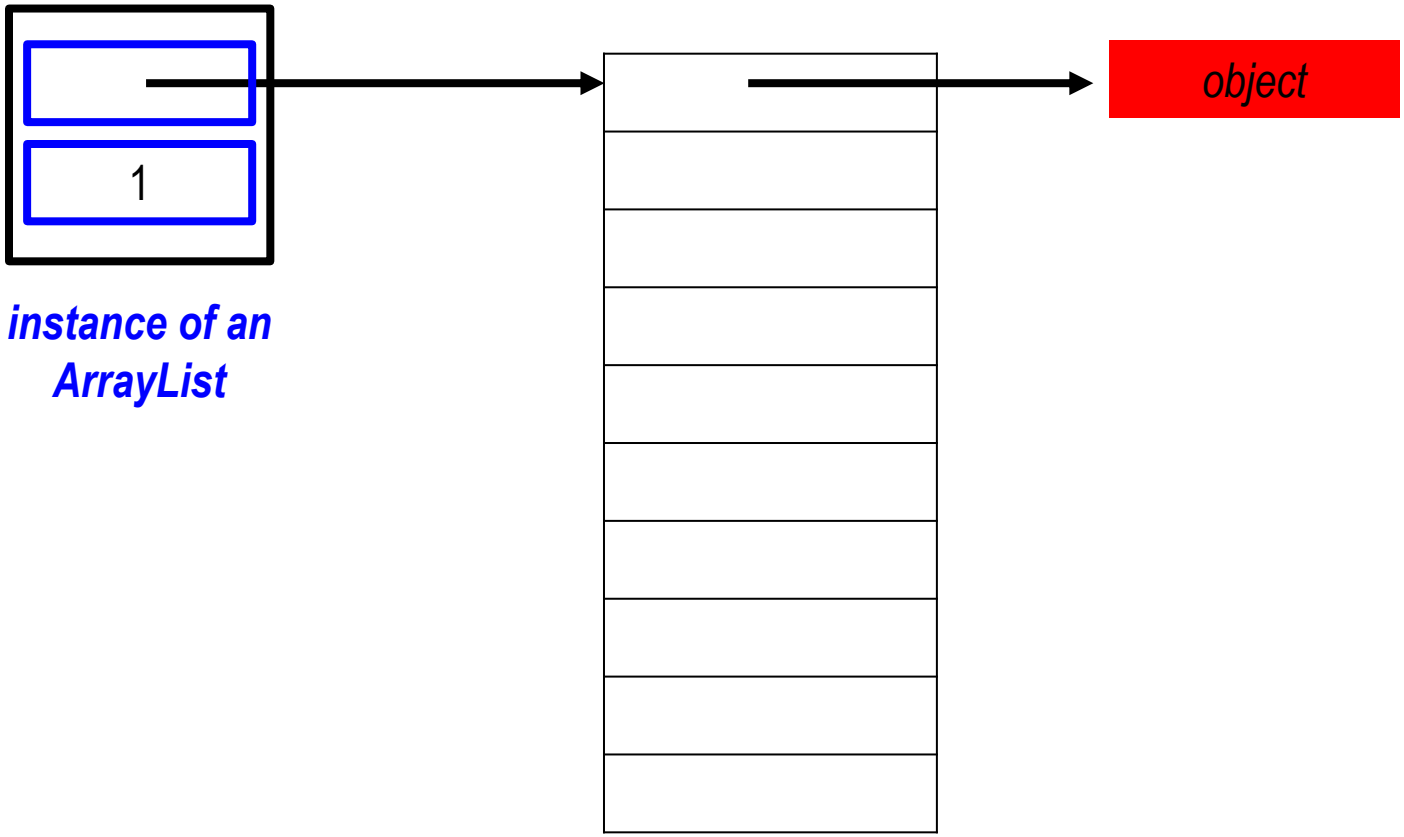
```
public class ArrayList implements List {
    ...
```

  - tells the compiler that the class will define *all* of the methods in the interface
  - if the class doesn't define them, it won't compile

- We'll look at two implementations of the `List` interface:
  - `ArrayList` – uses an array to store the items
  - `LLList` – uses a linked list to store the items

ArrayList Class

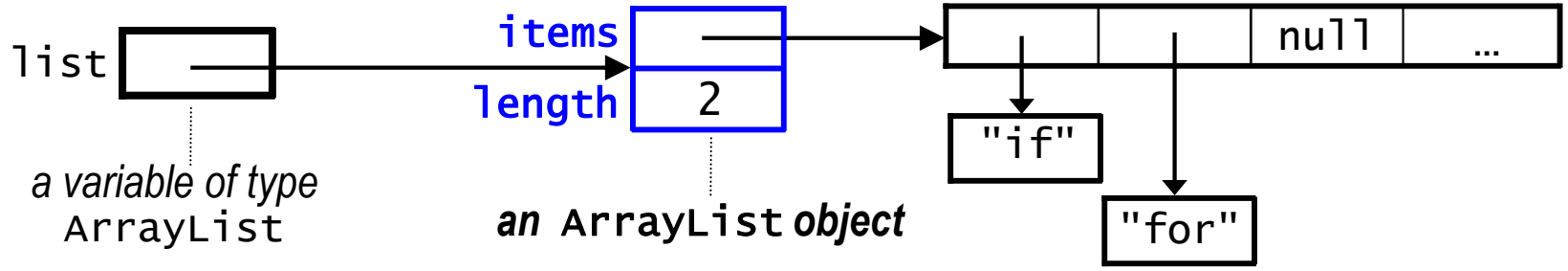
- Implementing the List interface with an **Array**



Implementing a List Using an Array

```
public class ArrayList implements List {
    private Object[] items;
    private int length;

    public ArrayList(int maxSize) {
        this.items = new Object[maxSize];
        this.length = 0;
    }

    public int length() {
        return this.length;
    }

    public boolean isFull() {
        return (this.length == this.items.length);
    }
    ...
}
```

we're showing local variables outside their stack frame!

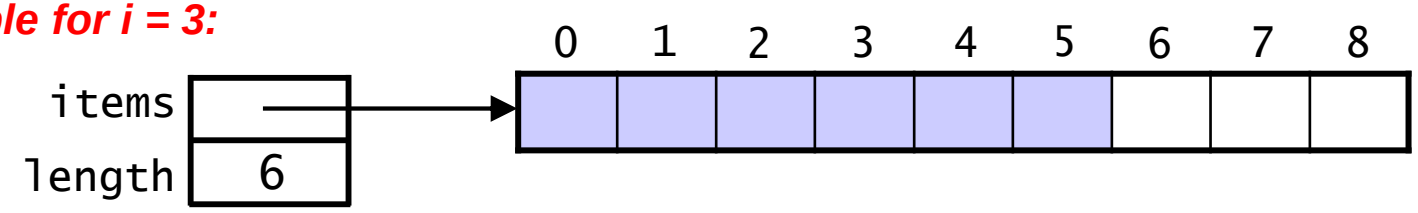


Adding an Item to an `ArrayList`

- Adding at position i (shifting items i, i+1, … to the right by one):

```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length) {
        throw new IndexOutOfBoundsException();
    } else if (isFull()) {
        return false;
    }

    // make room for the new item
    for (int j = length - 1; j >= i; j--) {
        items[j + 1] = items[j];
    }

    items[i] = item;
    length++;
    return true;
}
```

example for i = 3:

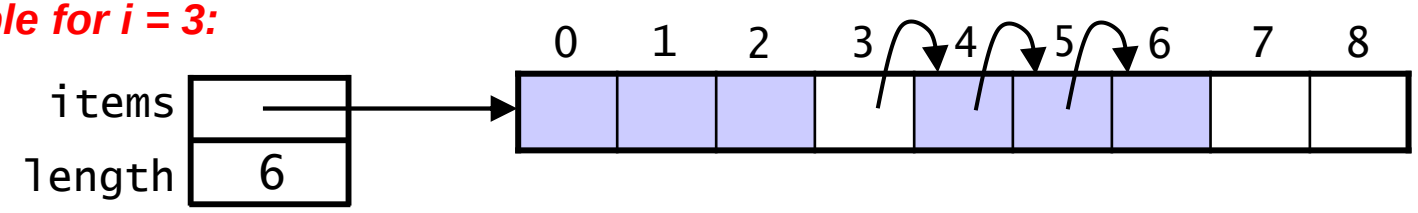


Adding an Item to an `ArrayList`

- Adding at position i (shifting items i, i+1, … to the right by one):

```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length) {
        throw new IndexOutOfBoundsException();
    } else if (isFull()) {
        return false;
    }

    // make room for the new item
    for (int j = length - 1; j >= i; j--) {
        items[j + 1] = items[j];
    }

    items[i] = item;
    length++;
    return true;
}
```

example for i = 3:

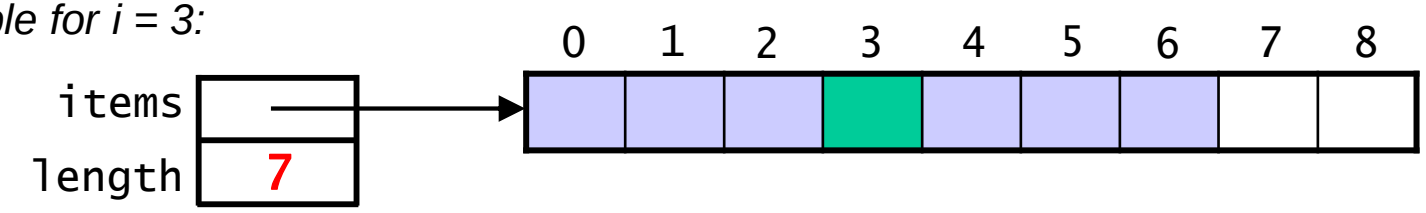


Adding an Item to an `ArrayList`

- Adding at position i (shifting items i, i+1, … to the right by one):

```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length) {
        throw new IndexOutOfBoundsException();
    } else if (isFull()) {
        return false;
    }

    // make room for the new item
    for (int j = length - 1; j >= i; j--) {
        items[j + 1] = items[j];
    }

    items[i] = item;
    length++;
    return true;
}
```

example for i = 3:

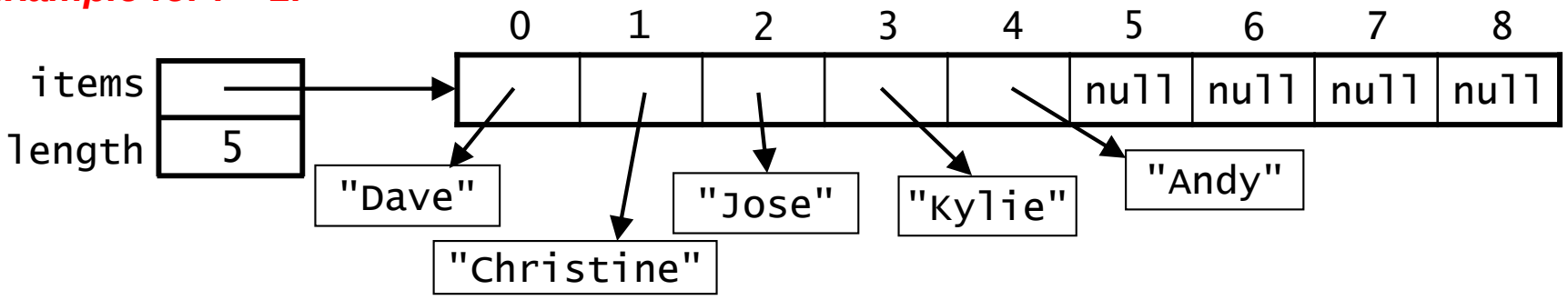


Removing an Item from an `ArrayList`

- Removing item i (shifting items i+1, i+2, … to the left by one):

```
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    Object removed = items[i];

    // shift items after items[i] to the left
    for (int j = i; j < length - 1; j++) {
        ____________________________________;
    }
    items[length - 1] = null;

    length--;
    return removed;
}
```

example for i = 1:

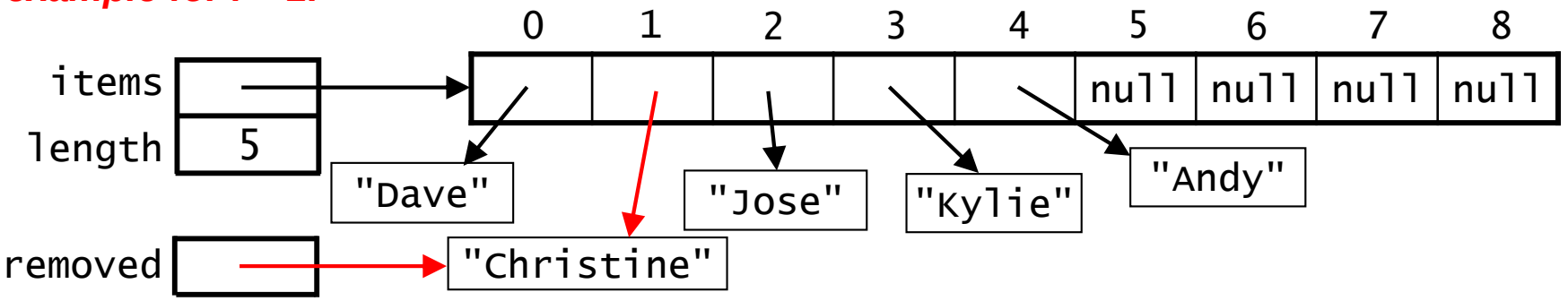


Removing an Item from an `ArrayList`

- Removing item i (shifting items i+1, i+2, … to the left by one):

```
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    Object removed = items[i];

    // shift items after items[i] to the left
    for (int j = i; j < length - 1; j++) {
        ____________________________________;
    }
    items[length - 1] = null;

    length--;
    return removed;
}
```

example for i = 1:

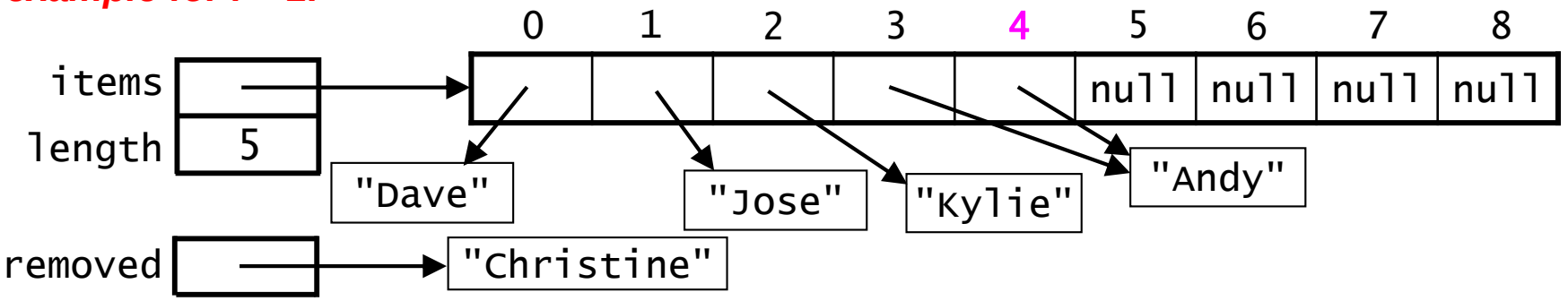


Removing an Item from an `ArrayList`

- Removing item i (shifting items i+1, i+2, … to the left by one):

```
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    Object removed = items[i];

    // shift items after items[i] to the left
    for (int j = i; j < length - 1; j++) {
        items[j] = items[j + 1];
    }
    items[length - 1] = null;

    length--;
    return removed;
}
```

example for i = 1:

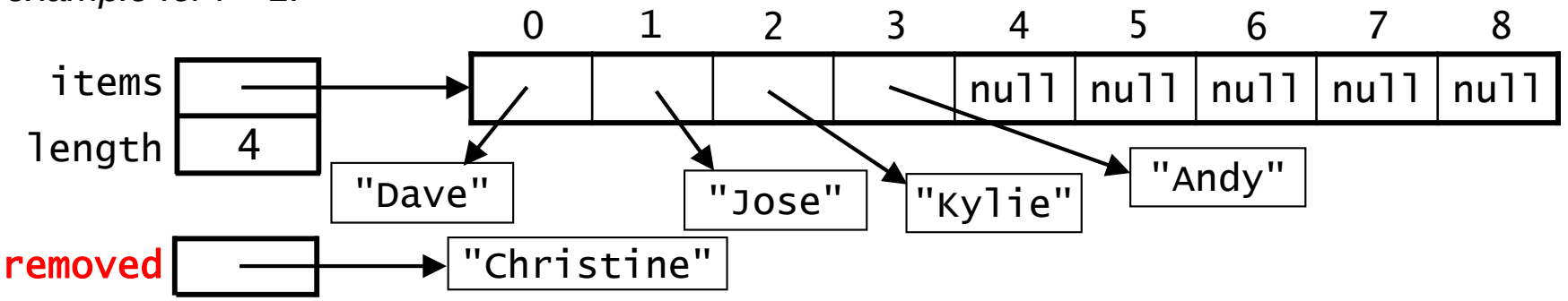


Removing an Item from an `ArrayList`

- Removing item i (shifting items i+1, i+2, … to the left by one):

```
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    Object removed = items[i];

    // shift items after items[i] to the left
    for (int j = i; j < length - 1; j++) {
        items[j] = items[j + 1];
    }
    items[length - 1] = null;

    length--;
    return removed;
}
```

example for i = 1:



Getting an Item from an `ArrayList`

- Getting item i (*without removing it*):

```
public Object getItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    return items[i];
}
```

Implementing an ADT Using a Class

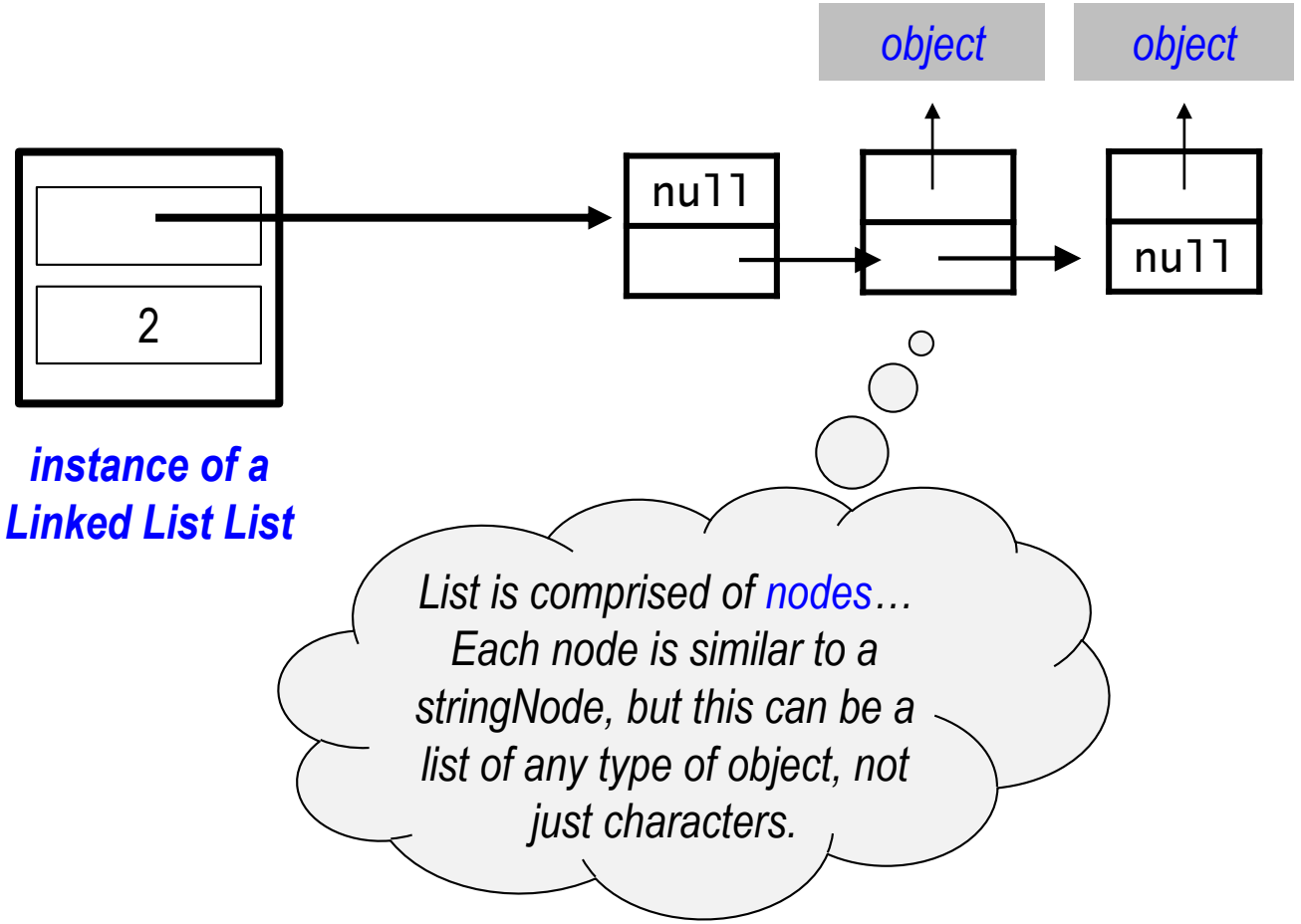
- To implement an ADT, we define a class.
- We specify the corresponding interface in the class header:

```
public class LLList implements List {
    ...
```

  - tells the compiler that the class will define *all* of the methods in the interface

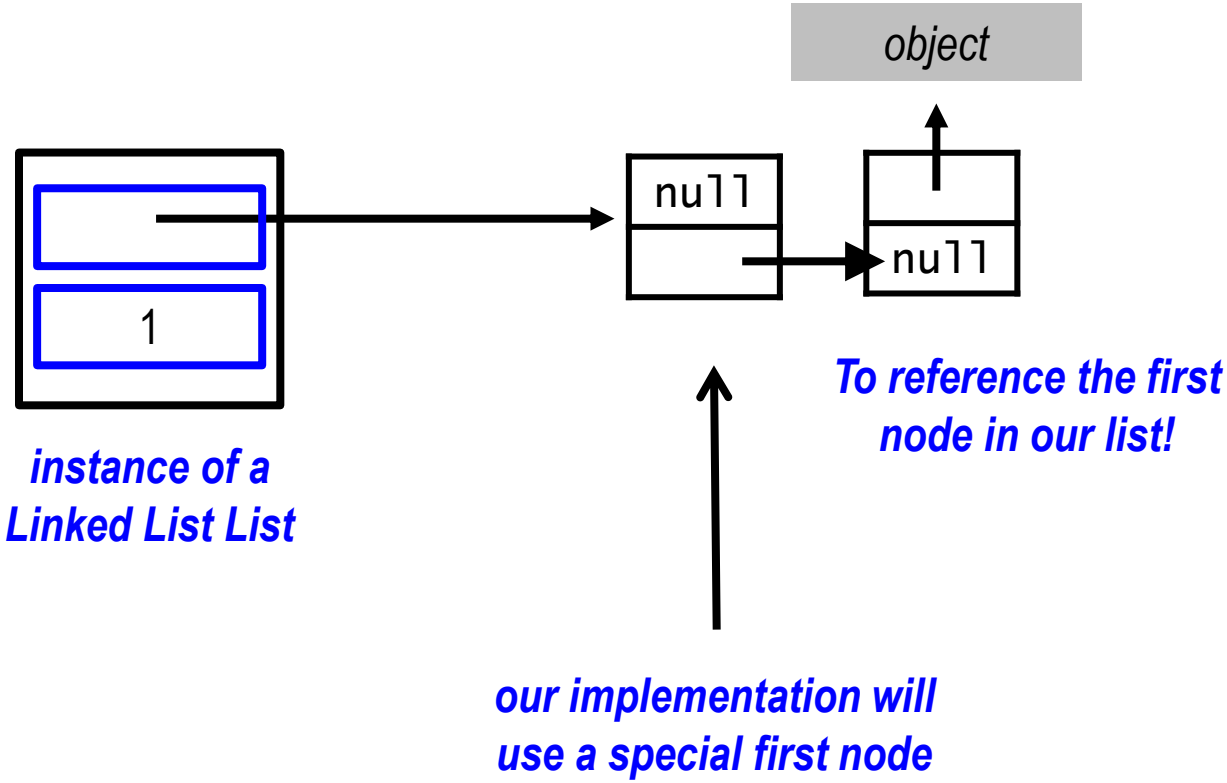
LLList Class

- Implementing the List interface with a **Linked List**



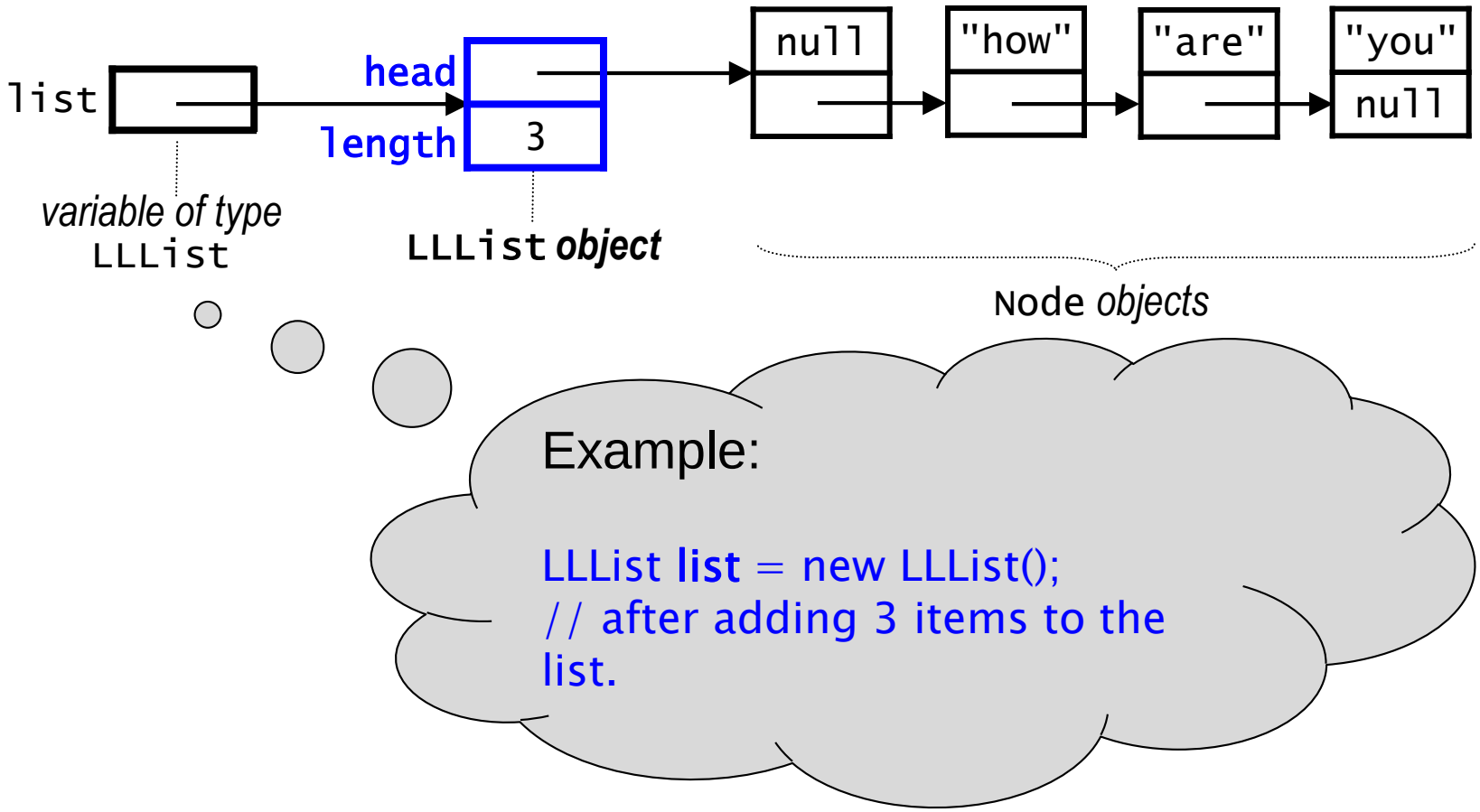
LLList Class

- Implementing the List interface with a **Linked List**



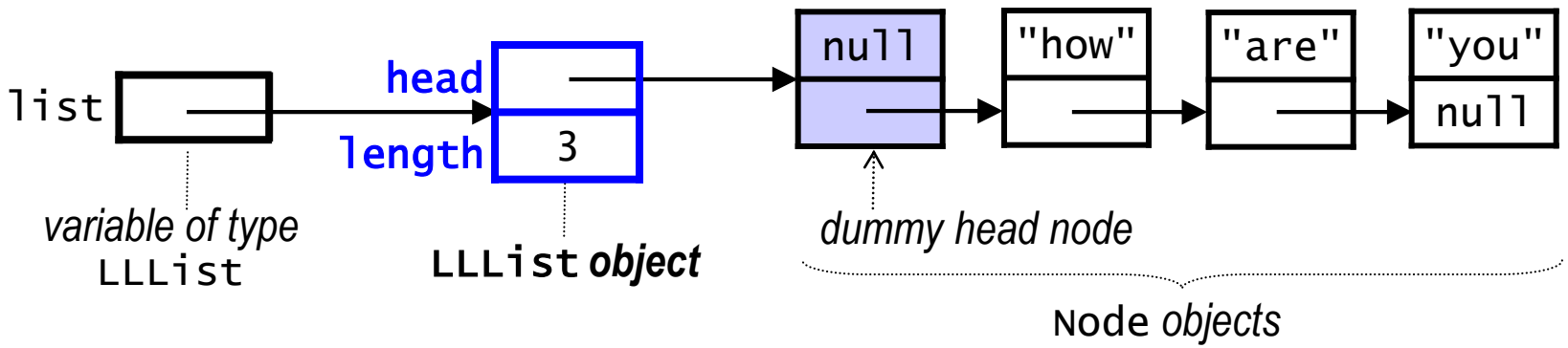
Implementing a List Using a Linked List

```
public class LLList implements List {
    private Node head;
    private int length;
    ...
}
```



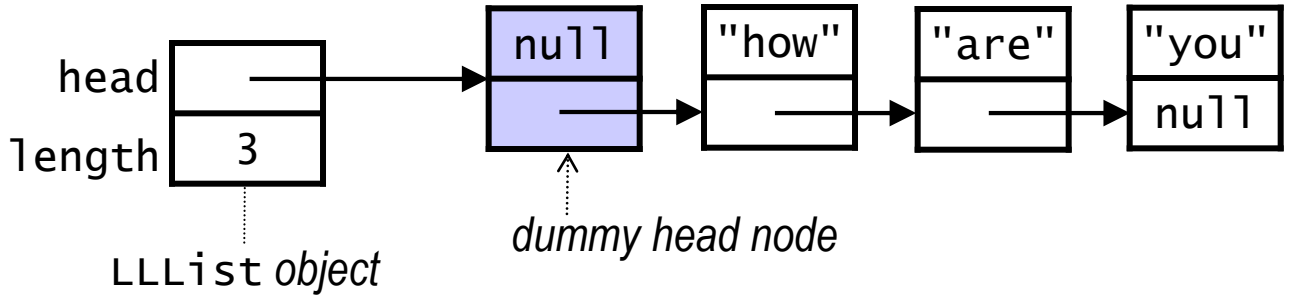
Implementing a List Using a Linked List

```
public class LLList implements List {
    private Node head;
    private int length;
    ...
}
```



- Differences from the linked lists we used for strings:
  - we "embed" the linked list inside another class
    - users of our `LLList` class won't actually touch the nodes
  - we use non-static methods instead of static ones
    `myList.length()` instead of `length(myList)`
  - we use a special *dummy head node* as the first node

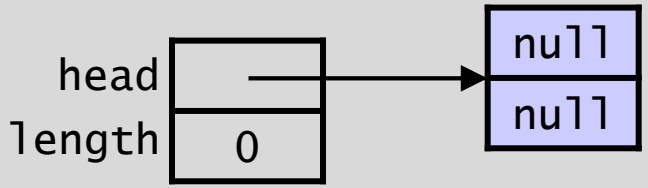
Using a Dummy Head Node



- The dummy head node is always at the front of the linked list.
  - like the other nodes in the linked list, it's of type `Node`
  - it does *not* store an item
  - it does *not* count towards the length of the list

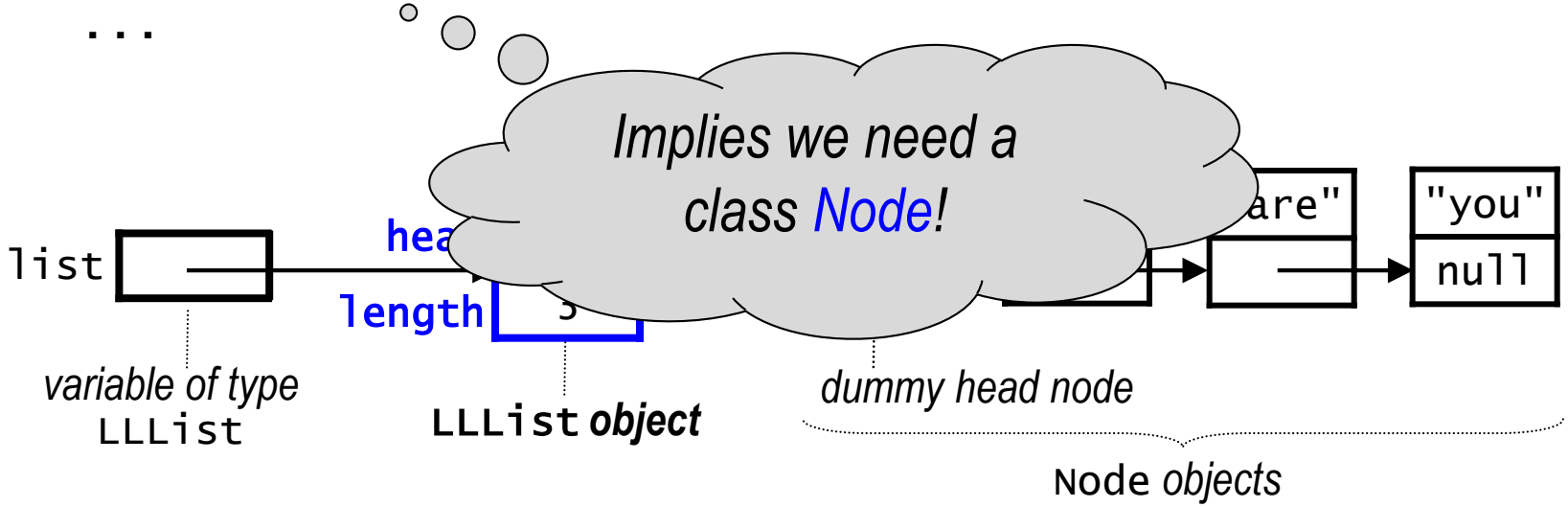
- Using it allows us to avoid special cases when adding and removing nodes from the linked list.

- An empty `LLList` still has a dummy head node:



Implementing a List Using a Linked List

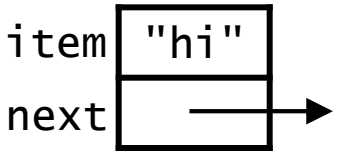
```
public class LLList implements List {
    private Node head;
    private int length;
    ...
}
```



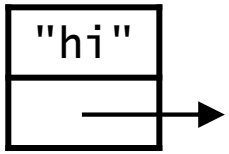
- Differences from the linked lists we used for strings:
  - we "embed" the linked list inside another class
    - users of our `LLList` class won't actually touch the nodes
  - we use non-static methods instead of static ones
    `myList.length()` instead of `length(myList)`
  - we use a special *dummy head node* as the first node

An Inner Class for the Nodes

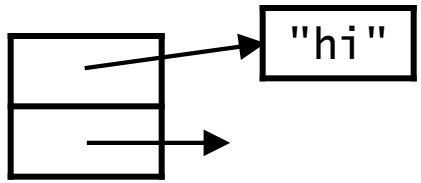
```
public class LLList implements List {
    private class Node {
        private Object item;
        private Node next;

        private Node(Object i, Node n) {
            item = i;
            next = n;
        }
    }
    ...
}
```



- We make `Node` an *inner class*, defining it within `LLList`.
  - allows the `LLList` methods to directly access `Node`'s private fields, while restricting access from outside `LLList`
  - the compiler creates this class file: `LLList$Node.class`
- For simplicity, our diagrams may show the items inside the nodes.



instead of



Other Details of Our `LLList` Class

```
public class LLList implements List {
    private class Node {
        // see previous slide
    }

    private Node head;
    private int length;

    public LLList() {
        head = new Node(null, null);
        length = 0;
    }

    public boolean isFull() {
        return false;
    }
    ...
}
```

- Unlike `ArrayList`, there's no need to preallocate space for the items. The constructor simply creates the dummy head node.

Other Details of Our `LLList` Class

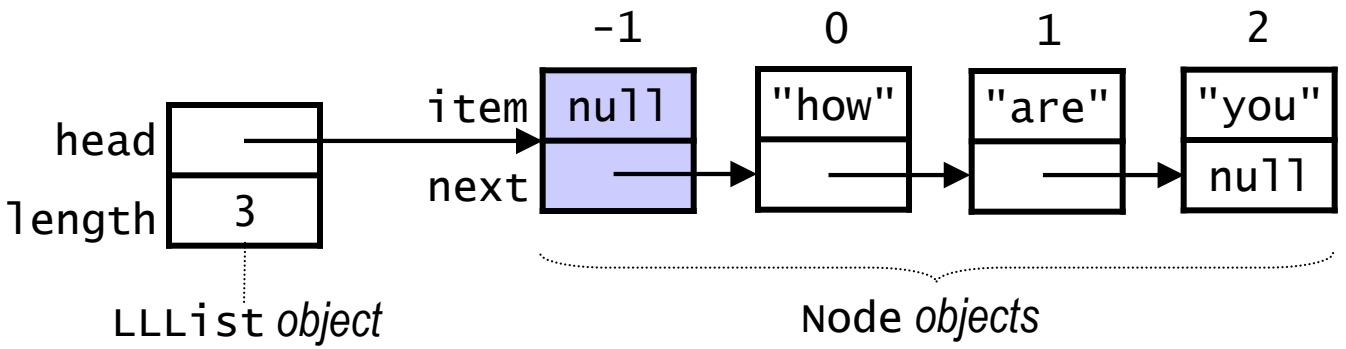
```
public class LLList implements List {
    private class Node {
        // see previous slide
    }

    private Node head;
    private int length;

    public LLList() {
        head = new Node(null, null);
        length = 0;
    }

    public boolean isFull() {
        return false;
    }
    ...
}
```

- Unlike `ArrayList`, there's no need to preallocate space for the items. The constructor simply creates the dummy head node.
- The linked list can grow indefinitely, so the list is never full!

Getting a Node

- Private helper method for getting node `i`
  - to get the dummy head node, use `i = -1`

```
private Node getNode(int i) {
    // private method, so we assume i is valid!

    Node trav = head;
    int travIndex = -1;
    while (travIndex < i) {
        travIndex++;
        trav = trav.next;
    }

    return trav;
}
```



Getting a Node

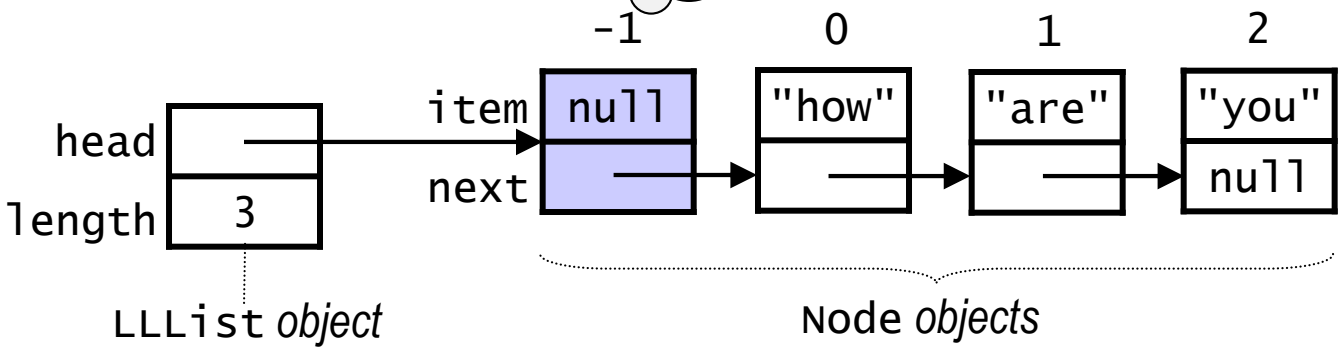
- Private helper method for getting node `i`
  - to get the dummy head node, use `i = -1`

```
private Node getNode(int i) {
    // private method, so w

    Node trav = head;
    int travIndex = -1;
    while (travIndex < i)
        travIndex++;
        trav = trav.next;
    }
    return trav;
}
```

Note `travIndex` is initialized to -1 because:

a. `head` is referencing the dummy node and that is its position in the list.

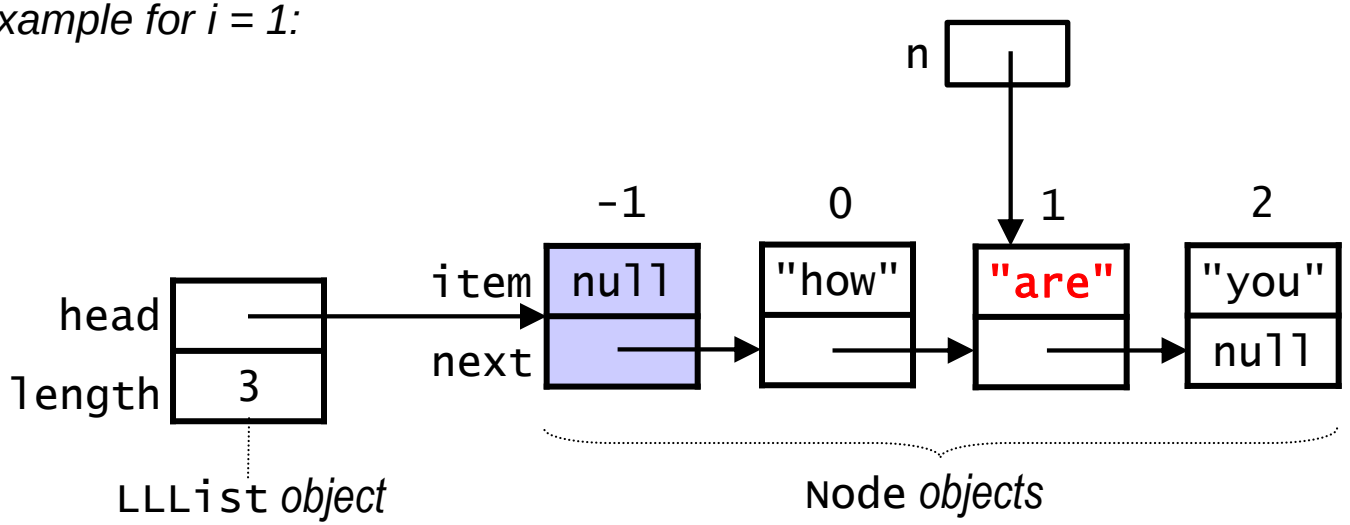
b. you may need to return a reference to the dummy node when adding/removing node at position 0.



Getting an Item

```
public Object getItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }

    Node n = getNode(i);
    return n.item;
}
```

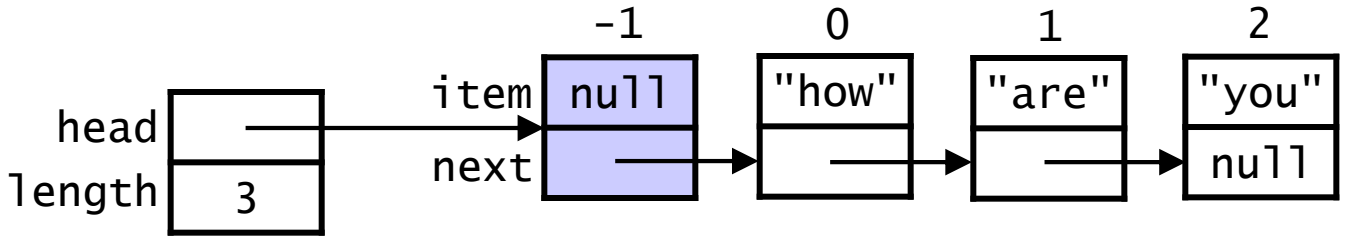
example for i = 1:



Adding an Item to an `LLList`

```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length) {
        throw new IndexOutOfBoundsException();
    }
    Node newNode = new Node(item, null);
    Node prevNode = getNode(i - 1);
    newNode.next = prevNode.next;
    prevNode.next = newNode;

    length++;
    return true;
}
```

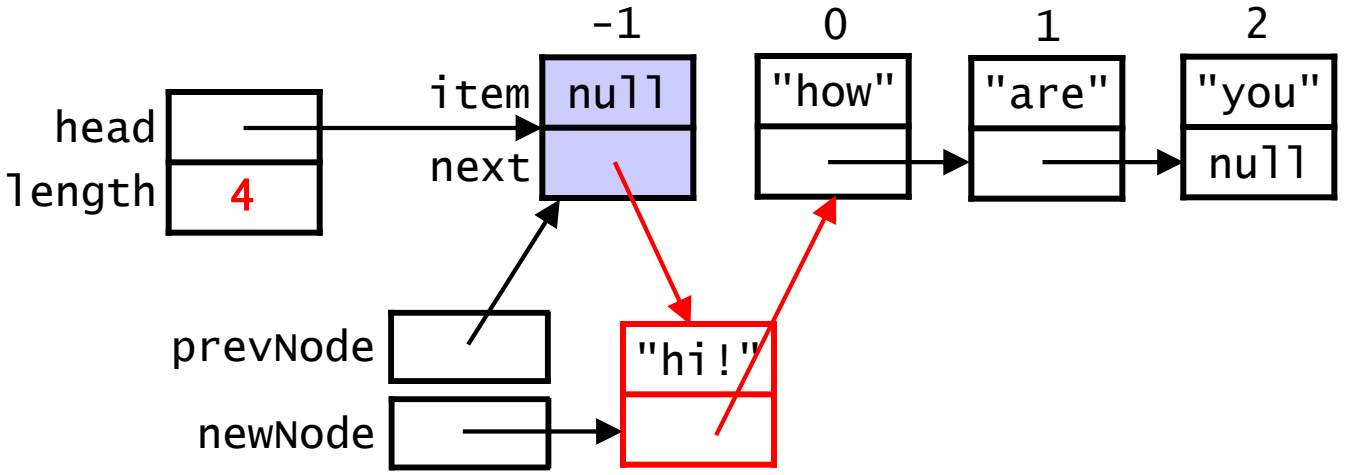
- This works even when adding at the front of the list (`i = 0`):



Adding an Item to an `LLList`

```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length) {
        throw new IndexOutOfBoundsException();
    }
    Node newNode = new Node(item, null);
    Node prevNode = getNode(i - 1);
    newNode.next = prevNode.next;
    prevNode.next = newNode;

    length++;
    return true;
}
```

- This works even when adding at the front of the list (`i = 0`):



addItem() Without a Dummy Head Node

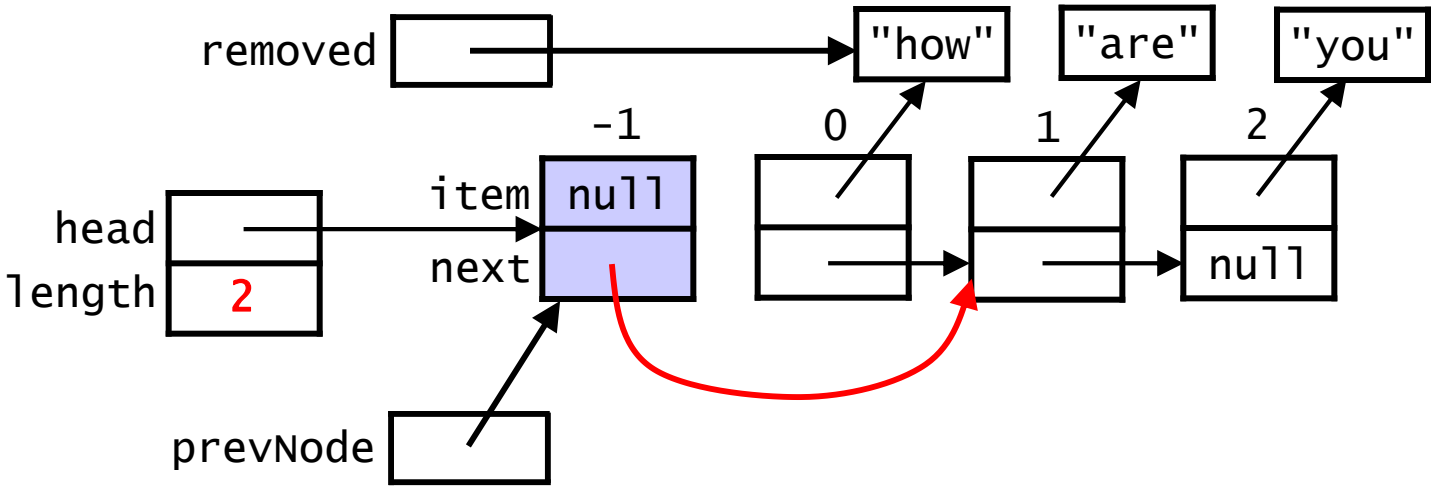
```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length) {
        throw new IndexOutOfBoundsException();
    }
    Node newNode = new Node(item, null);

    if (i == 0) {                    // case 1: add to front
        newNode.next = head;
        head = newNode;
    } else {                         // case 2: i > 0
        Node prevNode = getNode(i - 1);
        newNode.next = prevNode.next;
        prevNode.next = newNode;
    }

    length++;
    return true;
}
```

(the gray code shows what we would need to add if we didn't have a dummy head node)

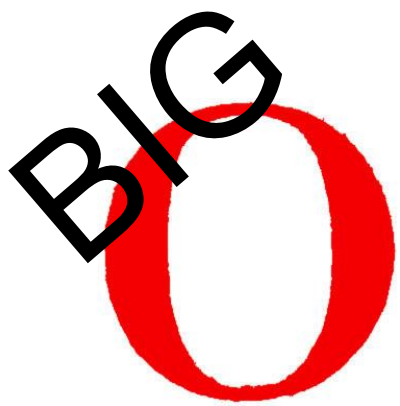
Removing an Item from an `LLList`

```
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    Node prevNode = getNode(i - 1);
    Object removed = prevNode.next.item;
    prevNode.next = prevNode.next.next;

    length--;
    return removed;
}
```

- This works even when removing the first item (`i = 0`):



Efficiency of the List Implementations



Computer Science 112
Boston University

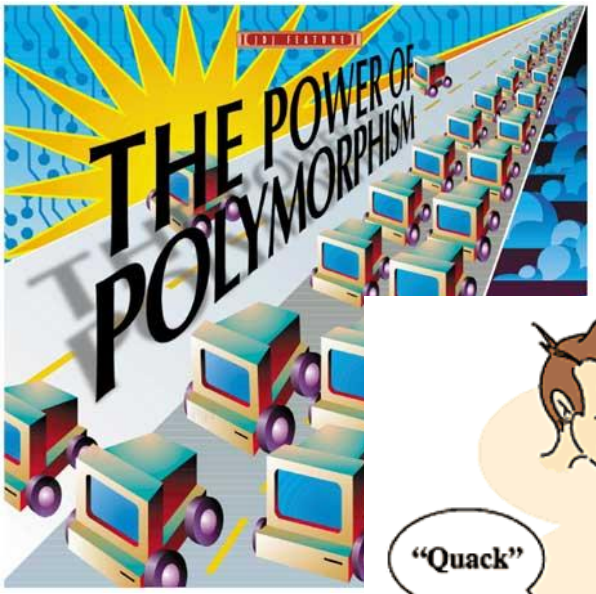
Christine Papadakis-Kanaris

Efficiency of the List Implementations

summary

	`ArrayList`	`LLList`
`getItem()`	*O*(1) because arrays provide random access	*O*(n) in the average and worst cases because you need to traverse the linked list
`addItem()`	• *O*(n) in the average and worst cases because you need to shift items right • *O*(1) if adding to the back of the list	• *O*(n) in the average and worst cases because you need to traverse the linked list • *O*(1) if adding to the front • could make adding to the back *O*(1) by maintaining a reference to the last node
`removeItem()`	• *O*(n) in the average and worst cases because you need to shift items left • *O*(1) if removing the last item	• *O*(n) in the average and worst cases • *O*(1) if removing the first item • could we improve the efficiency of removing the last item? not unless we use a doubly-linked list
space efficiency	*O*(m) where m is the *anticipated* maximum length	*O*(n)

Polymorphism and Interfaces



Recall: Polymorphism

- A reference variable of type T can hold a reference to an object from a *subclass* of T:

```
Rectangle r1 = new Square(50, "cm");
```

  - this works because `Square` is a subclass of `Rectangle`
  - a square *is* a rectangle!

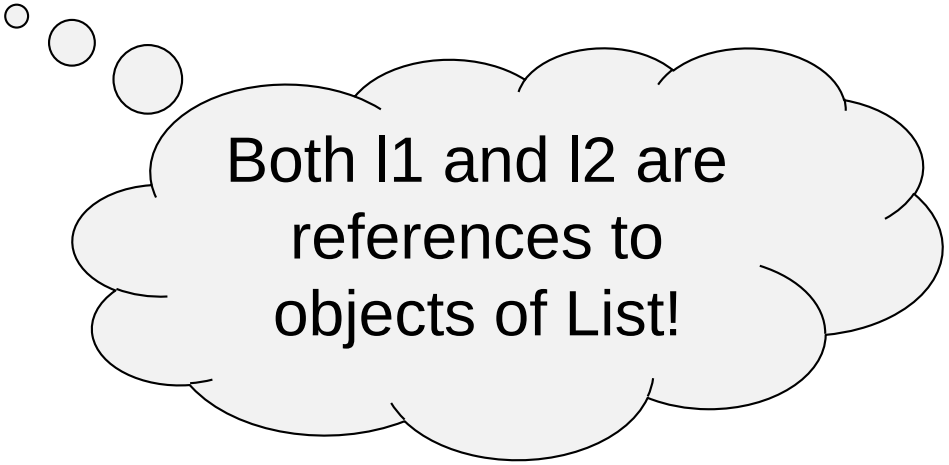
Another Example of Polymorphism

- An interface can be used as the type of a variable:

```
List myList;
```

- We can then assign an object of any class that implements the interface:

```
List l1 = new ArrayList(20);
List l2 = new LLList();
```



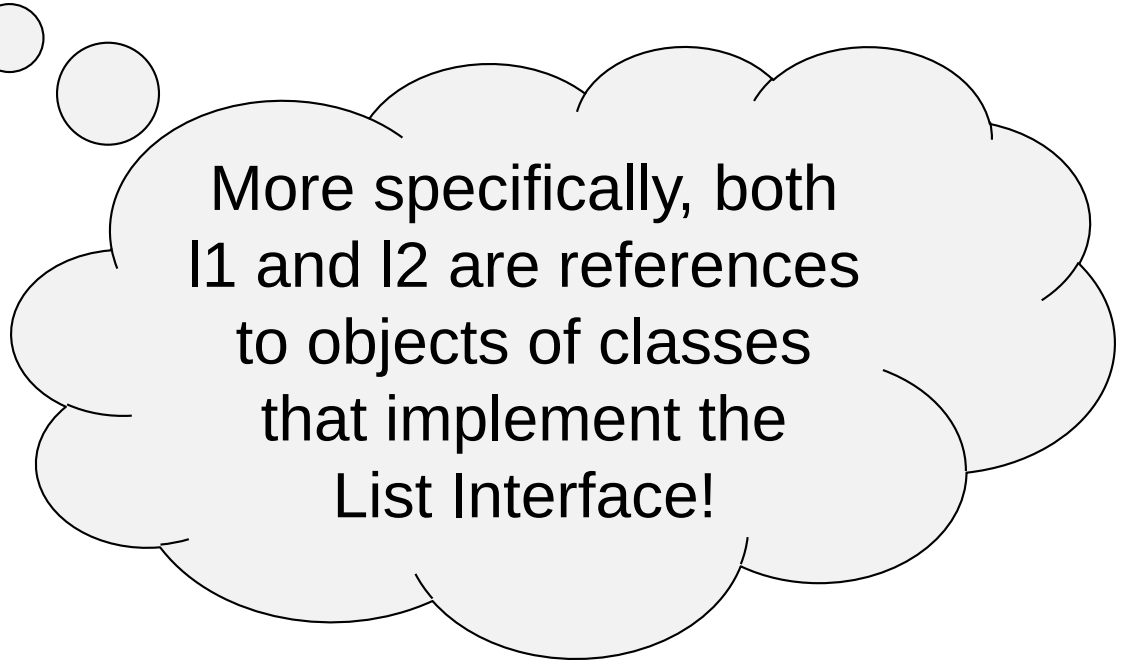
Another Example of Polymorphism

- An interface can be used as the type of a variable:

```
List myList;
```

- We can then assign an object of any class that implements the interface:

```
List l1 = new ArrayList(20);
List l2 = new LLList();
```



Another Example of Polymorphism

- An interface can be used as the type of a variable:

```
List myList;
```

- We can then assign an object of any class that implements the interface:

```
List l1 = new ArrayList(20);
List l2 = new LLList();
```

- This allows us to write code that works with *any* implementation of an ADT:

```
public static void processList(List vals) {
    for (int i = 0; i < vals.length(); i++) {
      …
}
```

  - `vals` can be an object of *any* class that implements `List`
  - regardless of which class `vals` is from, we know it has all of the methods in the `List` interface