Outline

- Java's List Interface
- ArrayList
  - Aside: Generics
- Linked List
- Iterator
- Conclusion
Motivation for Linked Lists

- The *problem* with ArrayLists:
  - Insertion and deletion can be slow
    - Lots of data right-shifting required in the average or worst cases
  - Possible waste of space

- Time is spent shuffling data around
  - Due to the usage of arrays as internal storage. *Chairs don't float.*
Insight:

- **What if we don't use arrays as the backing store?**
  - Allow individual items to "float" instead
    - Instead of being stuck in a row
- **Each item knows about the next item**
  - How could this help with performance?
  - The last item (tail of list) points to **null**.

**Important:** Each seat is called a **Node** in the Linked List. Nodes can "float" in space
The Node<E> Class

- A Node<E> is what we'll call an individual item in a Linked List
  - Stores a generic data element
  - Stores a reference to the next Node

- This is important!!
  - Nodes in a Linked List don't have an index!
  - They only know about the data stored in it, and the next Node.

- (Let's see the Node<E> class)
public class Node<E> {
    private E data;  /** what's being stored? */
    private Node<E> next; /** what's after me? */

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

    public Node(E data, Node<E> next) {
        this.data = data;
        this.next = next;
    }

    public E getData() {
        return this.data;
    }

    public Node<E> getNext() {
        return this.next;
    }

    public void setData(E data) {
        this.data = data;
    }

    public void getNext(Node<E> next) {
        this.next = next;
    }
}

But how do you use it?

// create nodes that float in space
Node<String> d = new Node<*>("Doc");
Node<String> g = new Node<*>("Goofy");
Node<String> s = new Node<*>("Sneezy");

// link them up!
g.setNext(d);
d.setNext(s);

// print them all out!
// TODO

Problem! That's not convenient at all!
We Want a Nice User Interface Again!

- Remember how easy it was to deal with ArrayLists?

```java
MyList<String> dwarfs = new MyArrayList<>();
dwarfs.add("Doc");
dwarfs.add("Goofy");
dwarfs.add("Sneezy");

for (int i = 0; i < dwarfs.size(); i++) {
    System.out.println(dwarfs.get(i));
}
```

- Wouldn't it be nice if we could do the same with Linked Lists?
  - Implement the MyList interface!

```java
MyList<String> dwarfs = new SinglyLinkedList<>();
dwarfs.add("Doc");
dwarfs.add("Goofy");
dwarfs.add("Sneezy");

for (int i = 0; i < dwarfs.size(); i++) {
    System.out.println(dwarfs.get(i));
}
```
Writing the SinglyLinkedList\(<E>\) Class

- **Problem**: Nodes are too hard to manage by hand

- Write a *SinglyLinkedList\(<E>\)* class to manage the collection of Nodes
  - It only needs:
    - (0) to implement the *MyList\(<E>\)* interface
    - (1) to know the head Node
    - (2) to know size of the linked list (that is, how many nodes in the list?)
Implement `MyList<E>` Interface Again

- `SinglyLinkedList<E>` must implement the `MyList<E>` interface
  - According to interface, users shouldn't know the `Node<E>` class even exists!
    - Hide it... but how?

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public boolean add(E item)</code></td>
<td>Adds item at the end of the list. Always returns true</td>
</tr>
<tr>
<td><code>public void add(int index, E item)</code></td>
<td>Adds a reference to item, inserting it before the item at position index</td>
</tr>
<tr>
<td><code>public E get(int index)</code></td>
<td>Returns the data in the element at position index</td>
</tr>
<tr>
<td><code>public int indexOf(E item)</code></td>
<td>Searches for target and returns the position of the first occurrence, or –1 if it is not in the List</td>
</tr>
<tr>
<td><code>public E remove(int index)</code></td>
<td>Removes the entry formerly at position index and returns it</td>
</tr>
<tr>
<td><code>public boolean remove(E item)</code></td>
<td>Stores a reference to item in the element at position index. Returns the data formerly at position index</td>
</tr>
<tr>
<td><code>public E set(int index, E item)</code></td>
<td>Stores a reference to item in the element at position index. Returns the data formerly at position index</td>
</tr>
<tr>
<td><code>public int size()</code></td>
<td>Return the current size</td>
</tr>
</tbody>
</table>
public class SinglyLinkedList<E> implements MyList<E> {
    private Node<E> head;  /** head of the list */
    private int size;

    public SinglyLinkedList() {
        this.head = null;
        this.size = 0;
    }
}
public class SinglyLinkedList<E> implements MyList<E> {
    private Node<E> head; //** head of the list */
    private int size;

    public SinglyLinkedList() {
        this.head = null;
        this.size = 0;
    }

    /** A Node is a building block of linked lists! */
    private static class Node<E> {
        private E data; //** what's being stored? */
        private Node<E> next; //** what's after me? */

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

        public Node(E data, Node<E> next) {
            this.data = data;
            this.next = next;
        }
    }

    Called the outer class

    Called a nested class

    (Where are the getters and setters? Don't need them)
Aside: Nested Classes

- Outer class vs. nested (inner) class
  - Outer class must be declared `public`
  - Nested classes can be public, protected, or private

- **Know this:** Visibility of inner class items
  - Outer class can access *any* field and method in inner class
  - *Even if they're private to the nested class!*
  - *This is why we don't need getters and setters in Node<E> anymore.*
Aside: Nested Classes

Visibility (cont.)

• If inner class is private:
  - Its instances can only be created from the outer class and from within itself!
  - Inner class can access all instance variables (even private ones!) in the outer class

• If inner class is static:
  - Inner class cannot access instance variables in the outer class
  - External classes can instantiate it without having created an object of inner class

Rule of thumb:

• When you need an inner class: make them private static.
public class SinglyLinkedList<E> implements MyList<E> {
    private Node<E> head; /** head of the list */
    private int size;

    public SinglyLinkedList() {
        this.head = null;
        this.size = 0;
    }

    /** A Node is a building block of linked lists! */
    private static class Node<E> {
        private E data; // what's being stored?
        private Node<E> next; // what's after me?

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

        public Node(E data, Node<E> next) {
            this.data = data;
            this.next = next;
        }
    }

    // Called the outer class
    // Called a nested class
    // (Where are the getters and setters? Don't need them)
}
## Implement MyList\<E\> Interface Again

- Want SinglyLinkedList\<E\> to implement our MyList\<E\> interface

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public ( E ) get(int index)</td>
<td>Returns the data in the element at position ( \text{index} )</td>
</tr>
<tr>
<td>public ( E ) set(int index, ( E ) item)</td>
<td>Stores a reference to ( \text{item} ) in the element at position ( \text{index} ). Returns the data formerly at position ( \text{index} )</td>
</tr>
<tr>
<td>public void add(int index, ( E ) item)</td>
<td>Adds a reference to ( \text{item} ), inserting it before the item at position ( \text{index} )</td>
</tr>
<tr>
<td>public boolean add(( E ) item)</td>
<td>Adds ( \text{item} ) at the end of the list. Always returns true</td>
</tr>
<tr>
<td>public int indexOf(( E ) search)</td>
<td>Searches for ( \text{target} ) and returns the position of the first occurrence, or (-1) if it is not in the List</td>
</tr>
<tr>
<td>public ( E ) remove(int index)</td>
<td>Removes the entry formerly at position ( \text{index} ) and returns it</td>
</tr>
<tr>
<td>public boolean remove(( E ) item)</td>
<td>Stores a reference to ( \text{item} ) in the element at position ( \text{index} ). Returns the data formerly at position ( \text{index} )</td>
</tr>
<tr>
<td>public int size()</td>
<td>Return the current size</td>
</tr>
</tbody>
</table>
Let's Write the Getter and Setter

- **Signatures:**
  - `public E get(int index) O(?)`
  - `public E set(int index, E item) O(?)`

- **Slight problem:**
  - Nodes in a linked list no longer correlate with an index!
    - (That was one of the nice things about ArrayLists)
  - Need to traverse the linked list to find a Node at a given index
    - We'll write a helper method to do this
      - `private Node<E> getNodeAt(int index)`
Next up: We have a *getter* now...

- **public int indexOf(E item)**
  - Returns index of the given item if found, or -1 if not found.
  - What's the worst-case running time of Version 1?

```java
// Version 1 uses the get() we just wrote
public int indexOf(E search) {
    for (int i = 0; i < this.size; i++) {
        E item = this.get(i); // use get()!
        if (search.equals(item)) {
            return i;
        }
    }
    return -1;
}
```

```java
// Version 2 doesn't use get(). Traverses the list itself.
public int indexOf(E search) {
    Node<E> current = this.head;
    for (int i = 0; i < this.size; i++) {
        if (search.equals(current.data)) {
            return i;
        }
        current = current.next;
    }
    return -1; //not found
}
```