References vs. objects

variable = value;

- a variable (left side of =) is an arrow (the base of an arrow)
- a value (right side of =) is an object (a box; what an arrow points at)

• For the list at right:
  - a.next = value;
    means to adjust where 1 points

  - variable = a.next;
    means to make variable point at 2
Reassigning references

• when you say:
  - `a.next = b.next;`

• you are saying:
  - "Make the variable `a.next` refer to the same value as `b.next`.
  - Or, "Make `a.next` point to the same place that `b.next` points."
Linked node question

• Suppose we have a long chain of list nodes:

```
list 
<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>990</td>
<td></td>
</tr>
</tbody>
</table>
```

− We don't know exactly how long the chain is.

• How would we print the data values in all the nodes?
Algorithm pseudocode

- Start at the **front** of the list.
- While (there are more nodes to print):
  - Print the current node's **data**.
  - Go to the **next** node.

**How do we walk through the nodes of the list?**

```
list = list.next;  // is this a good idea?
```
Traversing a list?

• One (bad) way to print every value in the list:

```java
while (list != null) {
    System.out.println(list.data);
    list = list.next;  // move to next node
}
```

– What's wrong with this approach?
  • (It loses the linked list as it prints it!)
A current reference

- Don't change `list`. Make another variable, and change that.
  - A `ListNode` variable is NOT a `ListNode` object

```java
ListNode current = list;
```

- What happens to the picture above when we write:

```java
current = current.next;
```
Traversing a list correctly

• The correct way to print every value in the list:

```java
ListNode current = list;
while (current != null) {
    System.out.println(current.data);
    current = current.next;  // move to next node
}
```

– Changing `current` does not damage the list.
Linked list vs. array

- Algorithm to print list values:

```java
ListNode front = ...;

ListNode current = front;
while (current != null) {
    System.out.println(current.data);
    current = current.next;
}
```

- Similar to array code:

```java
int[] a = ...;

int i = 0;
while (i < a.length) {
    System.out.println(a[i]);
    i++;
}
```
• Let's write a collection class named `LinkedIntList`.
  – Has the same methods as `ArrayIntList`:
    • `add`, `add`, `get`, `indexOf`, `remove`, `size`, `toString`

  – The list is internally implemented as a chain of linked nodes
    • The `LinkedIntList` keeps a reference to its `front` as a field
    • `null` is the end of the list; a `null` front signifies an empty list

```
LinkedIntList

front

add(value)
add(index, value)
indexOf(value)
remove(index)
size()
toString()

ListNode

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

element 0  element 1  element 2
```
public class LinkedIntList {
    private ListNode front;

    public LinkedIntList() {
        front = null;
    }

    methods go here
}
// Adds the given value to the end of the list.
public void add(int value) {
    ...
}

- How do we add a new node to the end of a list?
- Does it matter what the list's contents are before the add?
Adding to an empty list

- Before adding 20:

  front = □

- After:

  front = □

  data  next
  20     □

  element 0

  - We must create a new node and attach it to the list.
The `add` method, 1st try

// Adds the given value to the end of the list.
public void add(int value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list

        ...
    }
}
Adding to non-empty list

- Before adding value 20 to end of list:

  - Front = element 0
  - Element 0: data = 42, next = element 1
  - Element 1: data = -3

- After:

  - Front = element 0
  - Element 0: data = 42, next = element 1
  - Element 1: data = -3
  - Element 2: data = 20
Don't fall off the edge!

• To add/remove from a list, you must modify the _next_ reference of the node _before_ the place you want to change.

- Where should _current_ be pointing, to add 20 at the end?
- What loop test will stop us at this place in the list?
The `add` method

```java
// Adds the given value to the end of the list.
public void add(int value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ListNode current = front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new ListNode(value);
    }
}
```
Implementing `get`

// Returns value in list at given index.
public int get(int index) {
    ...
}

- Exercise: Implement the `get` method.
The `get` method

// Returns value in list at given index.
// Precondition: 0 <= index < size()  
public int get(int index) {
    ListNode current = front;  
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
// Inserts the given value at the given index.
public void add(int index, int value) {
    ...
}

- Exercise: Implement the two-parameter `add` method.
The `add` method (2)

// Inserts the given value at the given index.
// Precondition: 0 <= index <= size()
public void add(int index, int value) {
    if (index == 0) {
        // adding to an empty list
        front = new ListNode(value, front);
    } else {
        // inserting into an existing list
        ListNode current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current.next;
        }
        current.next = new ListNode(value, current.next);
    }
}