Values vs. References

- Does the following swap method work? Why or why not?

```java
public static void main(String[] args) {
    int a = 7;
    int b = 35;
    // swap a with b
    swap(a, b);
    System.out.println(a + " " + b);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Value semantics

- **value semantics**: Behavior where values are copied when assigned to each other or passed as parameters.
  - When one primitive is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```java
int x = 5;
int y = x; // x = 5, y = 5
y = 17;   // x = 5, y = 17
x = 8;    // x = 8, y = 17
```

Reference semantics

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one reference variable is assigned to another, the object is **not** copied; both variables refer to the same object.

```java
int[] a1 = {4, 5, 2, 12, 14, 14, 9};
int[] a2 = a1; // refers to same array as a1
a2[0] = 7;
System.out.println(a1[0]); // 7
```

```
index 0 1 2 3 4 5 6
value 7 5 2 12 14 14 9
```
References to same type

- What would happen if we had a class that declared one of its own type as a field?

```java
public class Strange {
    private String name;
    private Strange other;
}
```

- Will this compile?
  - If so, what is the behavior of the other field? What can it do?
  - If not, why not? What is the error and the reasoning behind it?

Array-Based List Review

- Array-based lists are what we’ve studied so far
  - ArrayIntList, ArrayList, SortedIntList all use arrays

- Arrays use a contiguous block of memory

  ```
  0 1 2 3 4 5 6 7 8
  6 2 5 3 7 1 4 -9 -8
  ```

- This means all elements are adjacent to each other

Advantages and Disadvantages

- Advantages of array-based lists
  - random access: can get any element in the entire array quickly
    - kind of like jumping to any scene on a DVD (no fast-forwarding required)

- Disadvantages of array-based lists
  - can’t insert/remove elements at the front/middle easily
    - have to shift the other elements
  - can’t resize array easily
    - have to create a new, bigger array

Linked Lists

- A linked list is a type of list

- But instead of a contiguous block of memory, like this:

  ```
  0 1 2 3 4
  5 6 7 8 9
  ```

- Linked list elements are scattered throughout memory:

  ```
  8 9 7 5 6
  ```

- But now the elements are unordered. How do linked lists keep track of everything?
Linked Lists

- Each element must have a reference to the next element:

```
8 → 9 → 7 → 5 → 6
```

- Now, so long as we keep track of the first element (the front), we can keep track of all the elements.

Linked Lists

- These references to the next element mean that linked lists have sequential access.
- This means that to get to elements in the middle of the list, we must first start at the front and follow all the links until we get to the middle:
  - kind of like fast-forwarding on a VHS tape
  - so getting elements from the middle/back is slow
- Linked lists also do some things well:
  - linked lists can insert elements quickly (no “shifting” needed)
  - linked lists can always add more elements (no set capacity)
- So there are tradeoffs between array lists and linked lists.

Linked data structures

- All of the collections we will use and implement in this course use one of the following two underlying data structures:
  - an array of all elements
    - ArrayList, Stack, HashSet, HashMap
  - a set of linked objects, each storing one element, and one or more reference(s) to other element(s)
    - LinkedList, TreeSet, TreeMap

A list node class

```
public class ListNode {
    int data;
    ListNode next;
}
```

- Each list node object stores:
  - one piece of integer data
  - a reference to another list node (it does NOT contain another ListNode object)
- ListNode can be "linked" into chains to store a list of values:

```
front → 42 → -3 → 17 → 9 null
```
List node client example

```java
public class ConstructList1 {
    public static void main(String[] args) {
        ListNode list = new ListNode();
        list.data = 42;
        list.next = new ListNode();
        list.next.data = -3;
        list.next.next = new ListNode();
        list.next.next.data = 17;
        list.next.next.next = null;
        System.out.println(list.data + " " + list.next.data + " " + list.next.next.data);
        // 42 -3 17
    }
}
```

List node w/ constructor

```java
public class ListNode {
    int data;
    ListNode next;
    public ListNode() {
        this.data = 0;
        this.next = null;
    }
    public ListNode(int data) {
        this.data = data;
        this.next = null;
    }
    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}
```

Exercise: Modify the previous client to use these constructors.

List node client example

- Two possible solutions:
  - ListNode list = new ListNode(42, new ListNode(-3, new ListNode(17)));
  - ListNode list = new ListNode(17);
    list = new ListNode(-3, list);
    list = new ListNode(42, list);

- NOTE: In the second solution, the nodes are added in reverse order!

- List creation is still somewhat tedious. More on that next time.

Linked node problem 1

- What set of statements turns this picture:
  - ListNode list = new ListNode(42, new ListNode(-3, new ListNode(17)));
  - list.next.next = new ListNode(30);

- Into this?
  - ListNode list = new ListNode(10);
    list.next = new ListNode(20);
    list.next.next = new ListNode(30);

- list.next.next = new ListNode(30);
Linked node problem 2

• What set of statements turns this picture:
  list → data | next: 10 | 20

• Into this?
  list → data | next: 30 | 10 | 20

list = new ListNode(30, list);

Linked node problem 3

• What set of statements turns this picture:
  list1 → data | next: 10 | 20
  list2 → data | next: 30 | 40

• Into this?
  list1 → data | next: 10 | 30 | 20
  list2 → data | next: 40

Linked node problem 3

• Two possible solutions:
  ListNode temp = list1.next;
  list1.next = list2;
  list2 = list2.next;
  list1.next.next = temp;

  ListNode temp = list2.next;
  list2.next = list1.next;
  list1.next = list2;
  list2 = temp;

Recall: A list node class

public class ListNode {
  int data;
  ListNode next;
}

• Each list node object stores:
  – one piece of integer data
  – a reference to another list node (it does NOT contain another ListNode object)

• ListNodes can be "linked" into chains to store a list of values:
  data | next: 42 | -3 | 17 | 9

• Warm-up: Write a method to find length of a linked list, given a reference to its first node.
References vs. objects

\texttt{variable = value;}

- \textit{a variable} (left side of \texttt{=}) is an arrow
- \textit{a value} (right side of \texttt{=}) is an object

\begin{itemize}
  \item For the list at right:
    \begin{itemize}
      \item \texttt{a.next = value;}
        \begin{itemize}
          \item means to adjust where 1 points
        \end{itemize}
      \item \texttt{variable = a.next;}
        \begin{itemize}
          \item means to make \texttt{variable} point at 2
        \end{itemize}
    \end{itemize}
\end{itemize}

Reassigning references

\begin{itemize}
  \item when you say:
    \begin{itemize}
      \item \texttt{a.next = b.next;}
    \end{itemize}
  \item you are saying:
    \begin{itemize}
      \item "Make the \texttt{variable} \texttt{a.next} refer to the same \texttt{value} as \texttt{b.next}"
      \item Or, "Make \texttt{a.next} point to the same place that \texttt{b.next} points."
    \end{itemize}
\end{itemize}

Basic Linked List Questions

\begin{itemize}
  \item Suppose you have two variables of type ListNode named \texttt{p} and \texttt{q}. Consider the following situation:
    \begin{itemize}
      \item \texttt{p}
      \item \texttt{q}
    \end{itemize}
  \item How many variables of type ListNode are there?
    \begin{itemize}
      \item 6, circled in green
    \end{itemize}
  \item How many ListNode objects are there?
\end{itemize}
Basic Linked List Questions

• How many ListNode objects are there?
  – 4, circled in green

Linked node question

• Suppose we have a long chain of list nodes:

  ![List nodes diagram]

  - We don’t know exactly how long the chain is.

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

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 = ...;
  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++;
  }
  ```
### Relationship to Array Code

- A table explaining this relationship:

<table>
<thead>
<tr>
<th>Description</th>
<th>Array Code</th>
<th>Linked List Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>go to front of list</td>
<td>int i = 0;</td>
<td>ListNode current = front;</td>
</tr>
<tr>
<td>continue?</td>
<td>i &lt; size</td>
<td>current != null</td>
</tr>
<tr>
<td>get current value</td>
<td>elementData[i]</td>
<td>current.data</td>
</tr>
<tr>
<td>go to next element</td>
<td>i++;</td>
<td>current = current.next;</td>
</tr>
</tbody>
</table>

- This may be helpful if you are comfortable with arrays

### For Loops

- Of course, we usually write the array code in a for loop:
  ```java
  for (int i = 0; i < size; i++) {
    System.out.println(elementData[i]);
  }
  ```

- And we can still do this with the linked list code:
  ```java
  for (ListNode current = front; current != null; current = current.next) {
    System.out.println(current.data);
  }
  ```

- Whether you use a for loop or a while loop to traverse the linked list is up to you.

### A LinkedList class

- Let's write a collection class named `LinkedList`.  
  - 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 `LinkedList` keeps a reference to its `front` as a field
    - `null` is the end of the list; a `null` front signifies an empty list

### LinkedList class v1

```java
public class LinkedList {
  private ListNode front;

  public LinkedList() {
    front = null;
  }

  methods go here
}
```
Implementing `add`

// 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?

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 an empty list

• Before adding 20:

• After:

- We must create a new node and attach it to the list.

Adding to non-empty list

• Before adding value 20 to end of list:

• After:

- The first try of the `add` method is as follows:
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

// 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.
Implementing `add` (2)

```
// Inserts the given value at the given index.
public void add(int index, int value) {
    ...
}

- Exercise: Implement the two-parameter `add` method.
```

```
front = 42

data
next

front = -3

data
next

front = 17

data
next
```

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);
    }
}
```

Implementing `get`

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

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

```
front = 42

data
next

front = -3

data
next

front = 17

data
next
```

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;
}
```
Implementing `remove` (2)

// Removes value at given index from list.
// Precondition: 0 <= index < size
public void remove(int index) {
  ...
}

- How do we remove any node in general from a list?
- Does it matter what the list's contents are before the remove?

Removing from a list

- Before removing element at index 1:

```
front = [data: 42, next: -3]  // element 0
[data: -3, next: 42]  // element 1
[null, next: 20]  // element 2
```

- After:

```
front = [null, next: 42]  // element 0
[null, next: 20]  // element 1
```

Removing from the front

- Before removing element at index 0:

```
front = [null, next: 42]  // element 0
[null, next: -3]  // element 1
[null, next: 20]  // element 2
```

- After:

```
front = [null, next: -3]  // element 0
[null, next: 20]  // element 1
```

Removing the only element

- Before:

```
front = [null, next: 20]  // element 0
```

- After:

```
front = [null, next: null]  // element 0
```

- We must change the front field to store `null` instead of a node.
- Do we need a special case to handle this?
remove (2) solution

// Removes value at given index from list.
// Precondition: 0 <= index < size()
public void remove(int index) {
    if (index == 0) {
        // special case: removing first element
        front = front.next;
    } else {
        // removing from elsewhere in the list
        ListNode current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current.next;
        }
        current.next = current.next.next;
    }
}

Linked list iterator

• The following code is particularly slow on linked lists:
  List<Integer> list = new LinkedList<Integer>();
  ...
  for (int i = 0; i < list.size(); i++) {
      int value = list.get(i);
      if (value % 2 == 1) {
          list.remove(i);
      }
  }
  ...
  – Why?
  – What can we do to improve the runtime?

Iterators (11.1)

• iterator: An object that allows a client to traverse the
elements of a collection, regardless of its implementation.
  – Remembers a position within a collection, and allows you to:
    • get the element at that position
    • advance to the next position
    • (possibly) remove or change the element at that position
  – Benefit: A common way to examine any collection's elements.

    | index | 0 | 1 | 2 |
    |------|---|---|---|
    | value| 42 | -3 | 17 |

    front          data      next
    42             -3        17

    current element: -3
    current index: 1

    current element: -3
    current index: 1

    Iterator methods

<table>
<thead>
<tr>
<th>hasNext()</th>
<th>returns true if there are more elements to examine</th>
</tr>
</thead>
<tbody>
<tr>
<td>next()</td>
<td>returns the next element from the collection (throws a NoSuchElementException if there are none left to examine)</td>
</tr>
<tr>
<td>remove()</td>
<td>removes from the collection the last value returned by next() (throws IllegalStateException if you have not called next() yet)</td>
</tr>
</tbody>
</table>

  – every provided collection has an iterator method

    Set<String> set = new HashSet<String>();
    ...
    Iterator<String> itr = set.iterator();
    ...

  • Exercise: Write iterators for our array list and linked list.
    – You don't need to support the remove operation.
for-each loop and Iterable

- Java’s collections can be iterated using a “for-each” loop:
  ```java
  List<String> list = new LinkedList<String>();
  ...
  for (String s : list) {
    System.out.println(s);
  }
  ```
  - Our collections do not work in this way.

- To fix this, your list must implement the `Iterable` interface.
  ```java
  public interface Iterable<E> {
    public Iterator<E> iterator();
  }
  ```

Our list classes

- We have implemented the following two list collection classes:
  ```java
  public class ArrayIntList {
    private int[] data;
    private int front;
    ...
  }
  ```
  ```java
  public class LinkedIntList {
    private Node front;
    private Node data;
    private Node next;
    ...
  }
  ```

- Problems:
  - We should be able to treat them the same way in client code.
  - Some of their methods are implemented the same way (redundancy).
  - Linked list carries around a clunky extra node class.
  - They can store only `int` elements, not any type of value.
  - It is inefficient to get or remove each element of a linked list.

Inner classes

- **inner class**: A class defined inside of another class.
  - can be created as static or non-static
  - we will focus on standard non-static (“nested”) inner classes

- usefulness:
  - inner classes are hidden from other classes (encapsulated)
  - inner objects can access/modify the fields of the outer object

Inner class syntax

```java
// outer (enclosing) class
public class Name {
  ...

  // inner (nested) class
  private class Name {
    ...
  }
}
```

- Only this file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object’s methods/fields.
  - If necessary, can refer to outer object as `OuterClassName.this`

- Exercise: Convert the linked node into an inner class.
Our list classes

- We have implemented the following two list collection classes:
  - ArrayIntList
    
    | index | value |
    |-------|-------|
    | 0     | 42    |
    | 1     | -3    |
    | 2     | 17    |
  - LinkedIntList
    
    front → data next
    | 42   | -3   | 17 |

- Problems:
  - We should be able to treat them the same way in client code.
  - Some of their methods are implemented the same way (redundancy).
  - Linked list carries around a clunky extra node class.
  - They can store only int elements, not any type of value.
  - It is inefficient to get or remove each element of a linked list.

Type Parameters (Generics)

ArrayList<Type> name = new ArrayList<Type>();

- Recall: When constructing a java.util.ArrayList, you specify the type of elements it will contain between < and >.
  - We say that the ArrayList class accepts a type parameter, or that it is a generic class.

ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");

Implementing generics

// a parameterized (generic) class
public class Name<Type> {
    ...
}

- By putting the Type in < >, you are demanding that any client that constructs your object must supply a type parameter.
  - You can require multiple type parameters separated by commas.
- The rest of your class's code can refer to that type by name.

Exercise: Convert our list classes to use generics.

Generics and arrays (15.4)

public class Foo<T> {
    private T myField; // ok
    public void method1(T param) {
        myField = new T(); // error
        T[] a = new T[10]; // error
        myField = param;
        T[] a2 = (T[]) (new Object[10]); // ok
    }
}

- You cannot create objects or arrays of a parameterized type.
- You can create variables of that type, accept them as parameters, return them, or create arrays by casting from Object[].
Generics and inner classes

```java
public class Foo<T> {
    private class Inner<T> {} // incorrect
    private class Inner {}     // correct
}
```

- If an outer class declares a type parameter, inner classes can also use that type parameter.
- Inner class should NOT redeclare the type parameter. (If you do, it will create a second type parameter with the same name.)