Lecture 9 handout — generics, equality, lists in memory

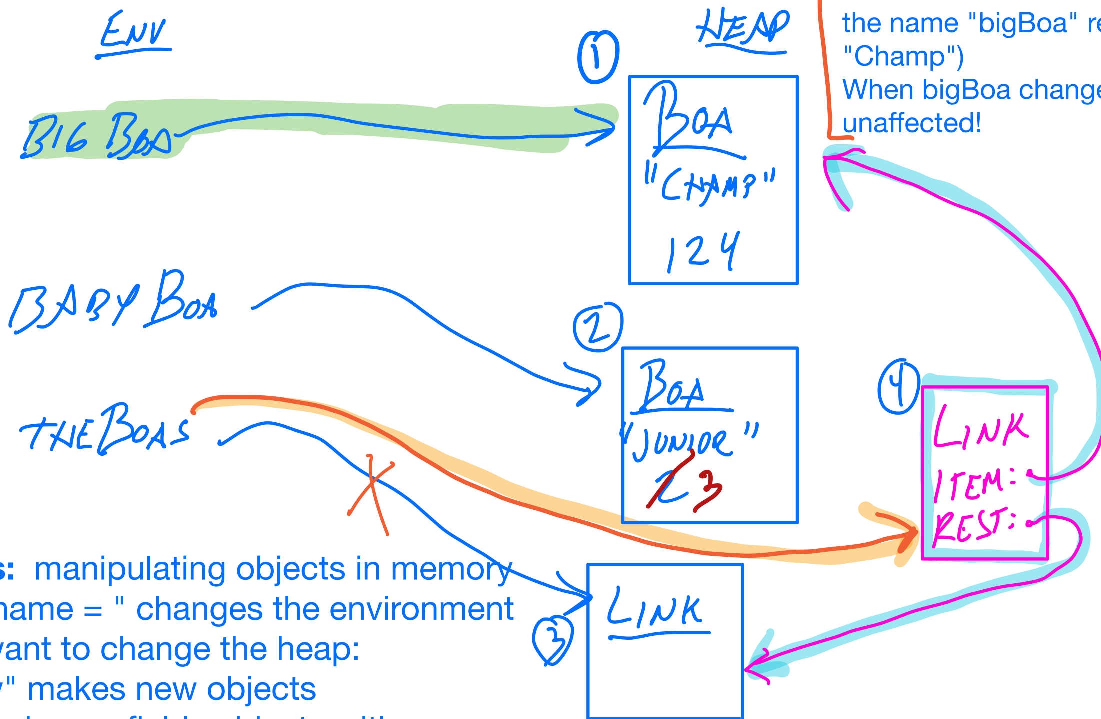
Review the Environment and Heap – how can each be changed?

```
// formerly NodeList
public class Link implements IList<T> {
    T first;
    IList<T> rest;
    ...

     public Link addFirst(T newElt) {
         return new Link(newElt, this);
     }

     public T getFirst() {
         return this.first;
     }
}
```

```
public class Boa {
     string name;
     int length;
     string eats;

     public growBy(int amt) {
       this.length = this.length + amt
     }
}
```

```
public void example() {
  ① Boa bigBoa = new Boa("Champ", 124, "peas");
  ② Boa babyBoa = new Boa("Junior", 2, "milk");
  ③ Link<Boa> theBoas = new Link<>(); // Note: IMMUTABLE LIST
  ④ theBoas = theBoas.addFirst(bigBoa);
  ⑤ babyBoa.growBy(1);   ] THIS.LENGTH += 1;

    System.out.println(theBoas.getFirst().length);
}
```

Note: the name bigBoa is NOT in the heap! when we run:
theBoas.addFirst(bigBoa),
we tell addFirst to use the OBJECT that the name "bigBoa" references (eg. "Champ")
When bigBoa changes later, the list node is unaffected!



Takeaways: manipulating objects in memory
=> Only "name = " changes the environment
=> If we want to change the heap:
- "new" makes new objects
- Can change fields objects with "obj.field"

Question: does list-immutability extend to the contents of the list elements?Defining our MutableList

No! Immutability refers to the *structure* (ie, the chain of NodeList/Link objects) => does not affect changes to objects inside the list (eg. "junior")!

Building a Mutable List

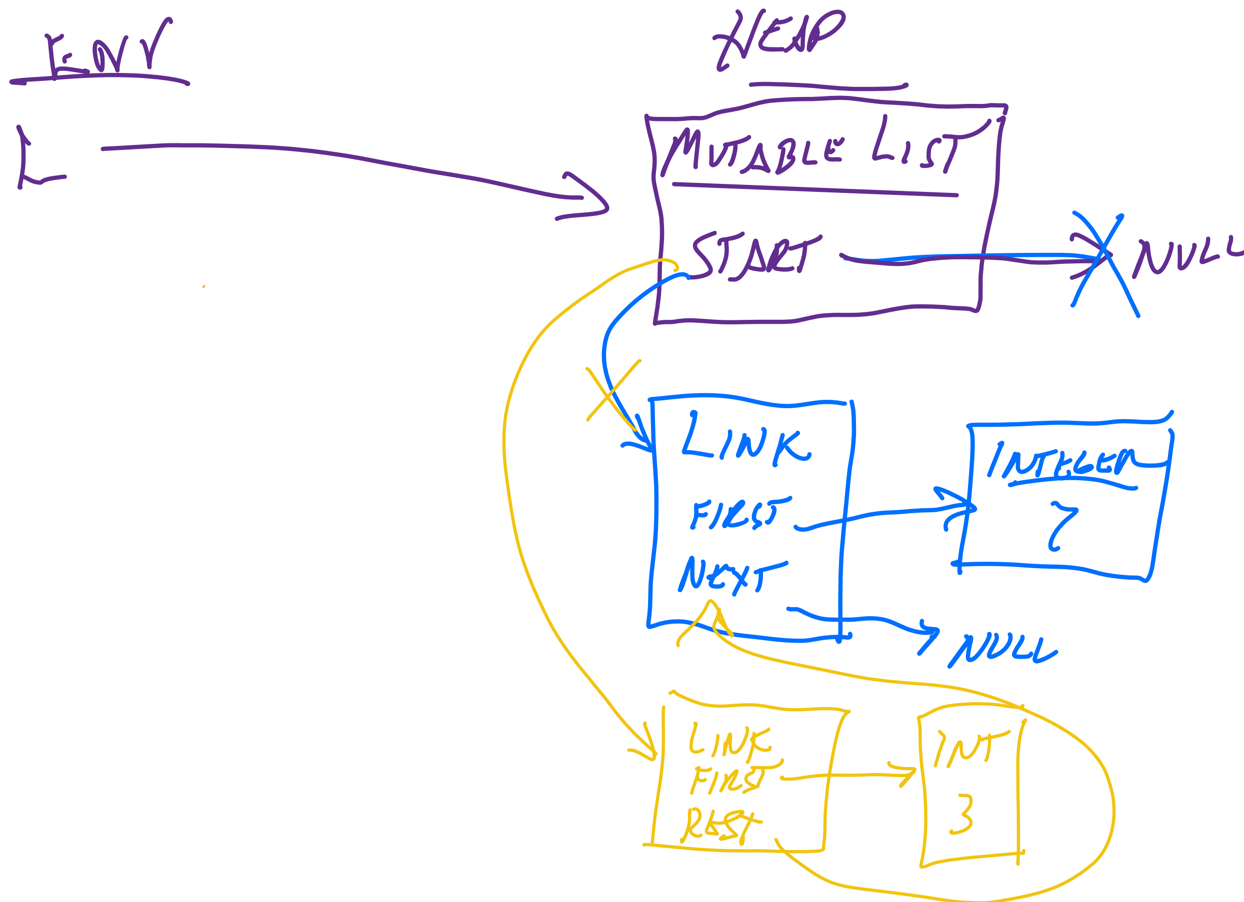
~~Lesson: Memory Diagrams with Addresses Explicit~~

Goal: want a list represented by a concrete object, which avoids some of the hassle of reassigning names with the NodeList/Link.

```
// the list [3, 7]
MutableList<Integer> L = new MutableList<>();
L.addFirst(7);
L.addFirst(3);
```

Idea: new class for MutableList, which has a field "Start" which points to the chain of nodes.

=> Each time we add to the list, start gets reassigned to point to the new "head" of the list
=> As we add nodes, "L" is unchanged!



For more info on why we have both mutable and immutable lists, see the notes for lecture 8.

(See next page, and posted code example, for the implementation)

Defining our MutableList

Here's our initial definition:

```
class MutableList {
       Node start;  // Front of the list


}


class Node {
   int item; // Data
   Node next; // Makes chain of nodes
}
```

How do we write addFirst? addFirst's goal is to create a new node at the beginning of the list. This would require three things:

- Make a new object for the new node
- The "next" field of the new object needs to point to the old start of the list (ie, this.start)
- this.start needs to be reassigned to point to this new object (which will now be the start of the list!

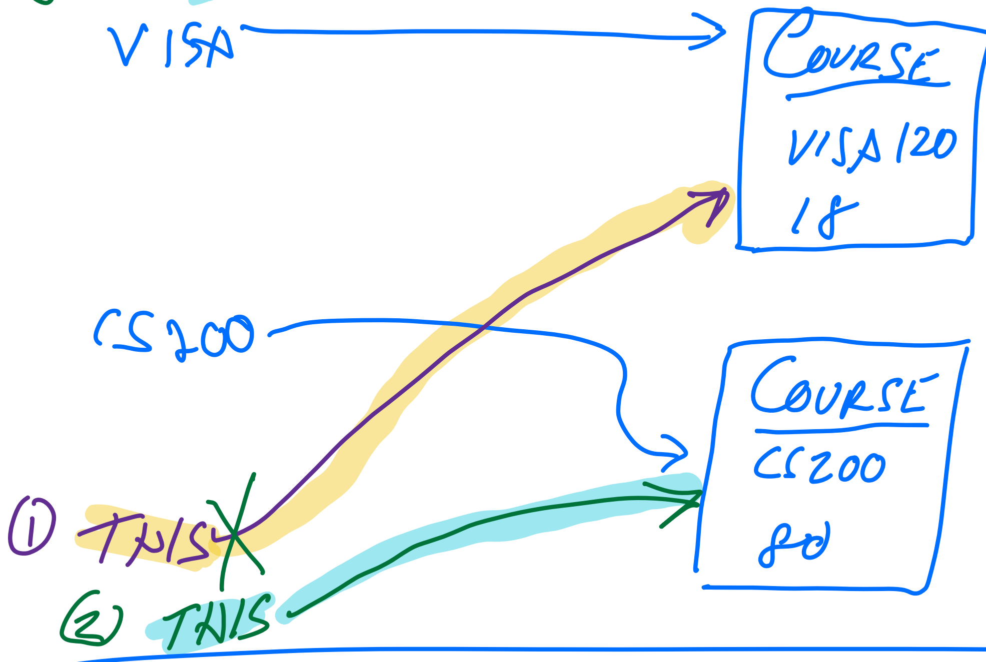
```
class MutableList {
 // . . .

  public void addFirst(int newElt) {
          this.start = new Node(newElt, this.start)

  }
}
```

For more on how "this" works, see the next few pages.

BACKGROUND: HOW "THIS" WORKS

Consider the following code:

```
Course visa = new Course("visa120", 18)
Course cs200 = new Course("cs200" 80)
visa.enroll()
cs200.enroll()
```



When we call enroll() on each object, Java will set up the name "this" to point to the object on which it was called.

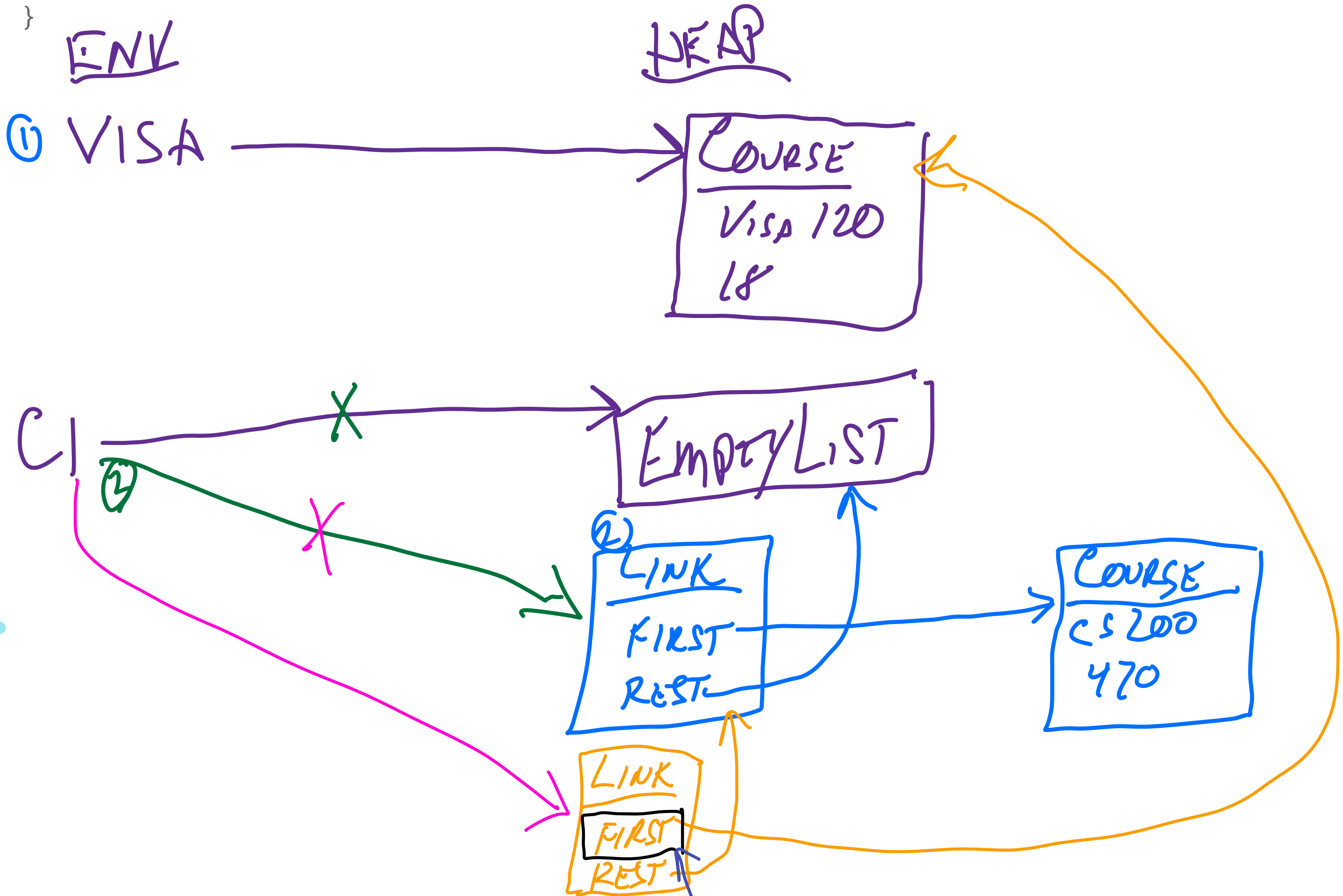
```
① VISA.ENROLL() {
      THIS.ENROLLMENT += 1;
  }
```

When visa.enroll() returns, the name this is removed.
When we call cs200.enroll(), Java again sets up "this"—now it points to the cs200 object.

```
② CS200.ENROLL() {
      THIS.ENROLLMENT += 1;
  }
```

Question: Does list-immutability extend to the contents within list elements?

```java
public static void courseExample() {
    Course visa = new Course("visa120", 18);
    IList<Course> C1 = new EmptyList<Course>(); // NOTE -- IMMUTABLE LIST
    C1 = C1.addFirst(new Course("csci200", 470));
    C1 = C1.addFirst(visa);
    visa.enroll();

    // what do we expect to see here?
    System.out.println(C1);
}
```



[for reference for this question]

Common mistake: the link does not contain the name (from the end) visa. First just "points to" or "refers to" the visa object itself.
=> The heap can't look back to the environment, it only refers to objects

```java
public class Link<T> implements IList<T> {
    T first;
    IList<T> rest;

    public IList addFirst(T newElt) {
        return new Link(newElt, this);
    }
}
```

Lecture 9 handout — generics, equality, lists in memory, lists with addresses

Question: How do we make our List classes have elements of any type (not just int)?

```
public class Node     {
    int first;
    Node next;
}

public class MutableList {
    Node start; // front of the list

    public void addFirst(int newItem) {
        newNode = new Node(newItem, this.start);
        this.start = newNode;
        return this;
    }
}
```

<T> T <T> T

TYPE VARIABLE (USUALLY SINGLE CAPITAL LETTERS)

Node<T>

WHENEVER WE USE A GENERIC TYPE, NEED TO FILL IN TYPE PARAMETER!

EX. FILLING IN TYPE PARAMETER
LINKED LIST<STRING>

Question: What does it mean for lists to be "the same"

DISCUSS: ARE THESE LISTS EQUAL?

```
public static void equalityExample() {
    MutableList<Integer> L1 = new MutableList<Integer>();
    L1.addFirst(6);
    L1.addFirst(8);
    System.out.println("L1 is " + L1);

    MutableList<Integer> L2 = new MutableList<Integer>();
    L2.addFirst(6);
    L2.addFirst(8);
    System.out.println("L2 is " + L2);


    // what do you expect each of these to produce? (what do == and .equals mean?)
    System.out.println(L1 == L2);
    System.out.println(L1.equals(L2));
    System.out.println(L1.toString() == L2.toString());
    System.out.println(L1.toString().equals(L2.toString()));
}
```

→ SAME OBJECTS IN HEAP?

The == operator checks if two names refer to the same object
=> Object equality

.equals() method: Programmer (of MutableList in this case) will tell us what equality means
=> This is more flexible, and could let us check structural equality: e.g., could compare, eg. all elements are the same content, same order, etc.
=> As the developers of the MutableList class, we could decide which constraints to pick, and therefore what it means for two objects to be considered equal!