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

TYPE VARIABLE (USUALLY SINGLE CAPITAL LETTERS)

<T>

T

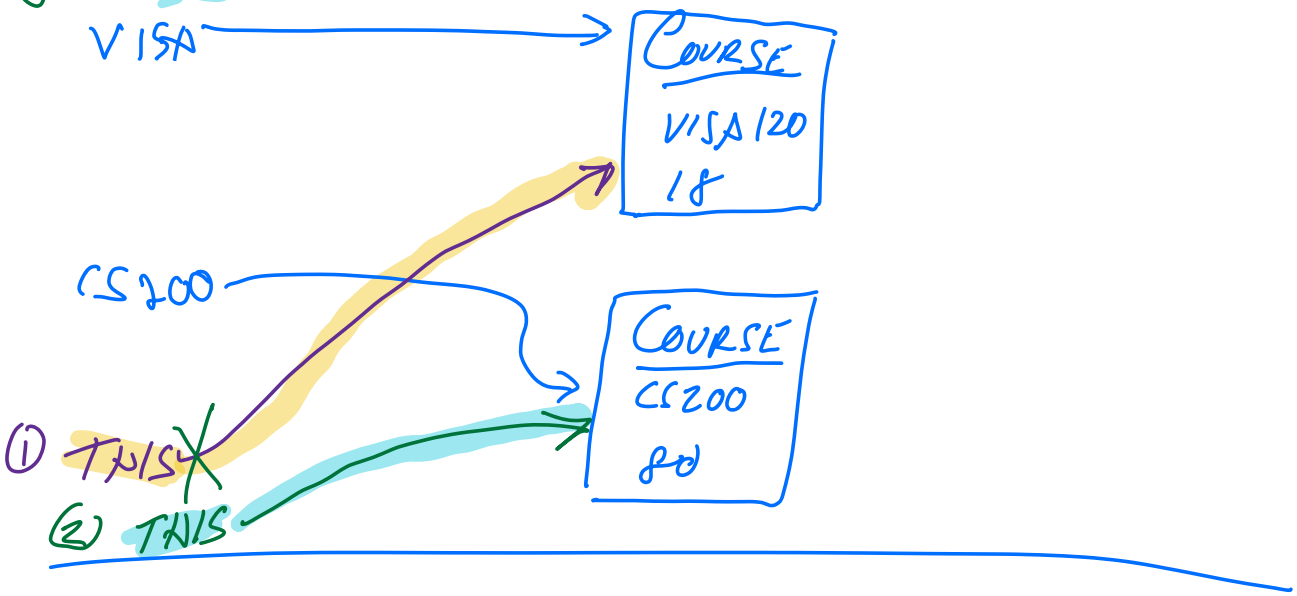
Node<T>

EX. FILLING IN TYPE PARAMETER
LinkedList<String>

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

BACKGROUND: HOW "THIS" WORKS

Consider the following code:
Course visa = new Course("visa120", 18)
Course cs200 = new Course("cs200" 80)
1. visa.enroll()
2. 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.

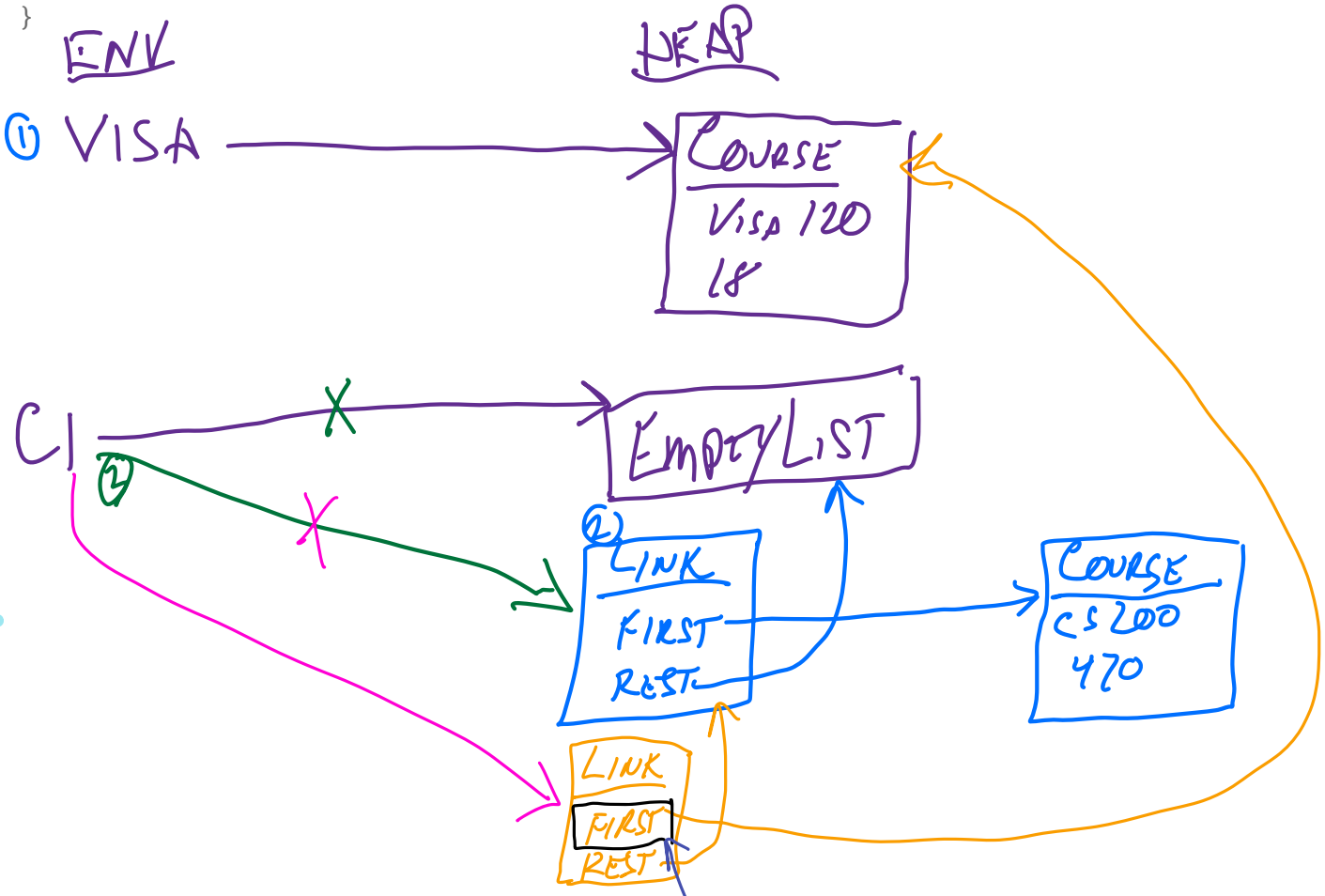
```
1  VISA.ENROLL() {
       THIS.ENROLLMENT += 1;
   }
```

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?

```
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);
}
```



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

---

[for reference for this question]

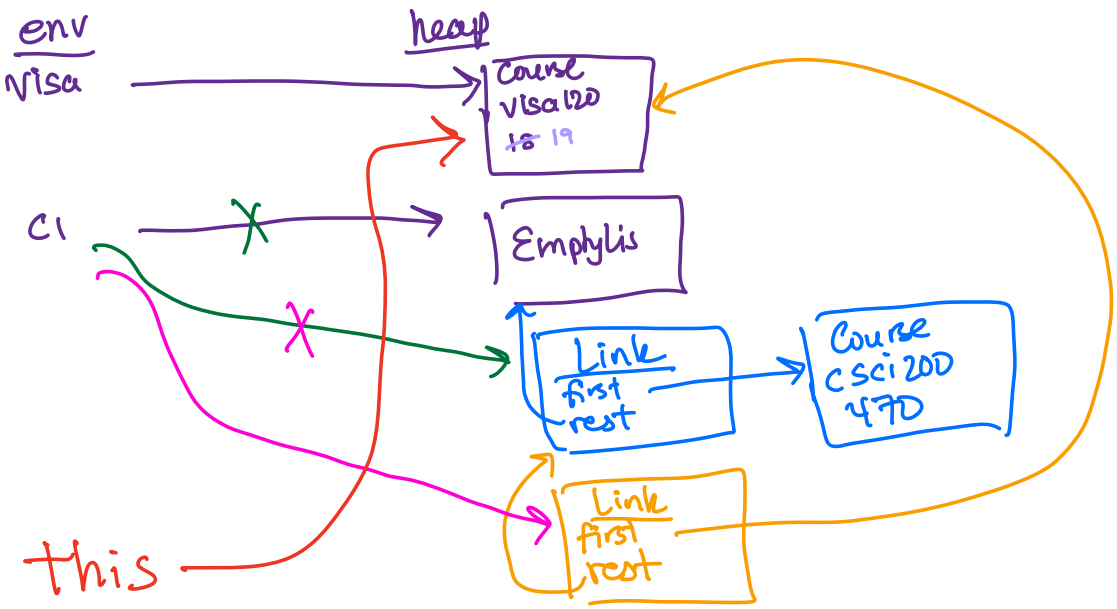
```
public class Link<T> implements IList<T> {
   T first;
   IList<T> rest;

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

How does the call visa.enroll() work?

- When we call a method, the name this goes into the environment, referring to the object on which the method was called
- then, the body of the method gets evaluated

visa.enroll()

this.enrollment = this.enrollment + 1

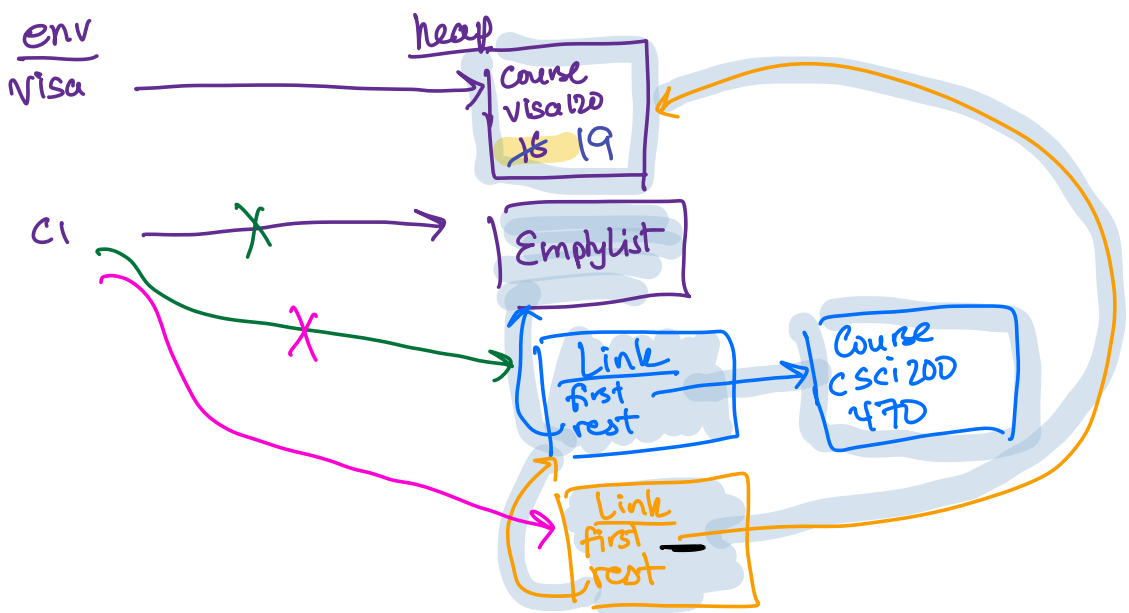


Note that we have an immutable list, but the contents changed. what sense does that make?

The real question is what does immutability of a list even cover?

Immutability covers the structure of a list: the link objects it contains and what objects are accessed via first. The contents of those objects, however, are not part of the list and can be changed.

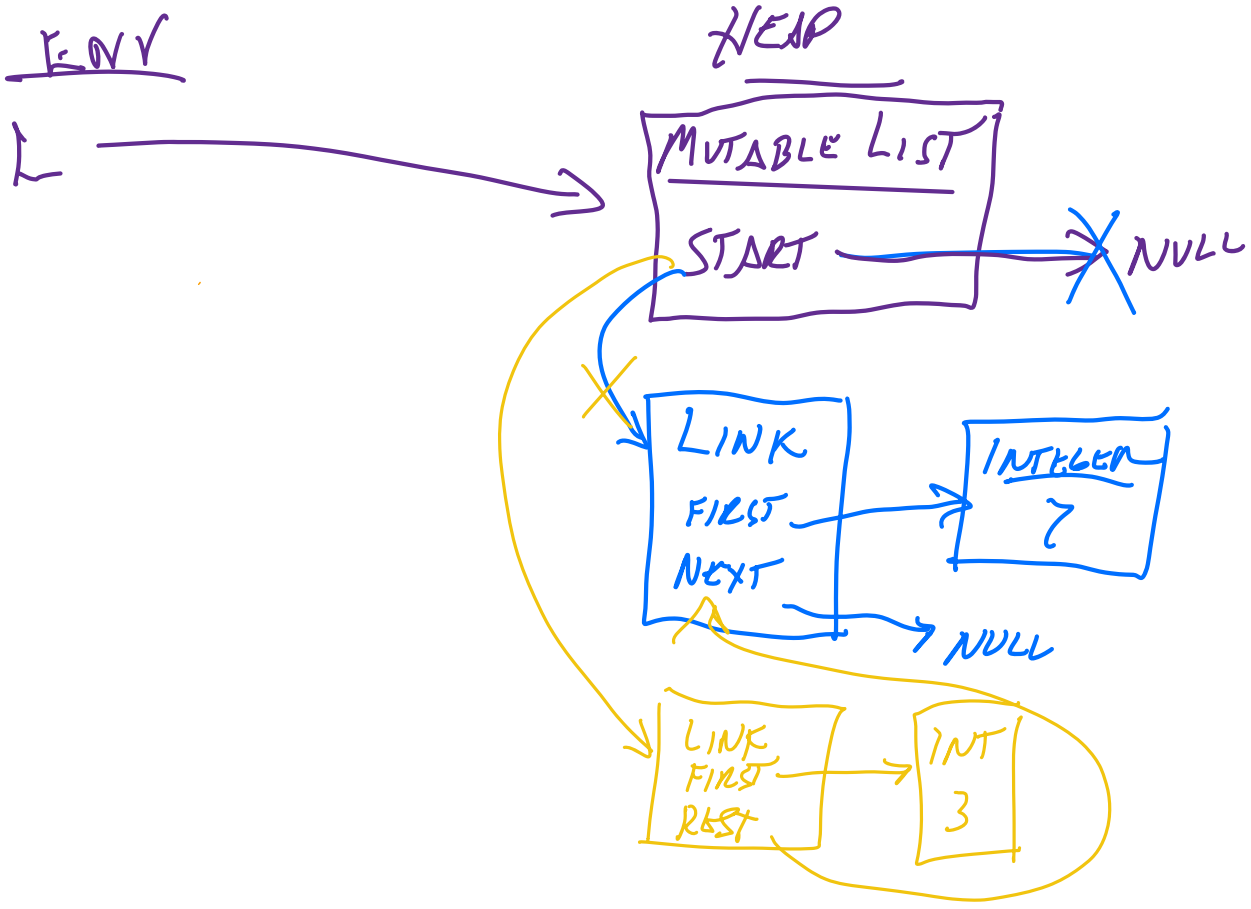
(the bluish highlights show what cannot change in an immutable list)



How would this look for a mutable list?

Lesson: ~~Memory Diagrams with Addresses Explicit~~

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



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

We'll see more about how mutable lists look in memory next class.