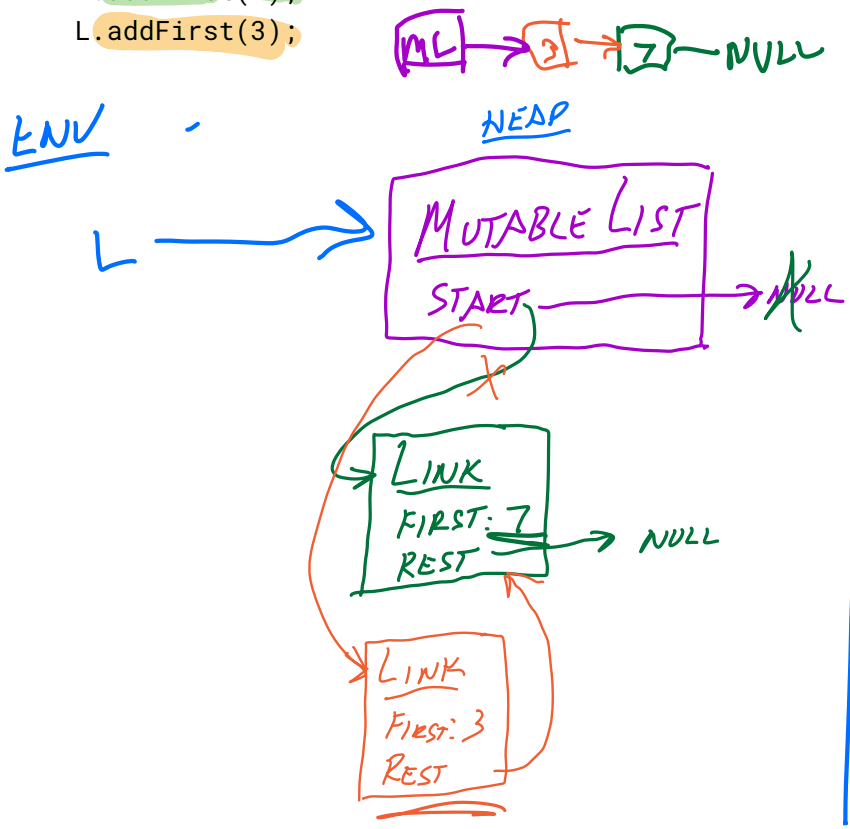
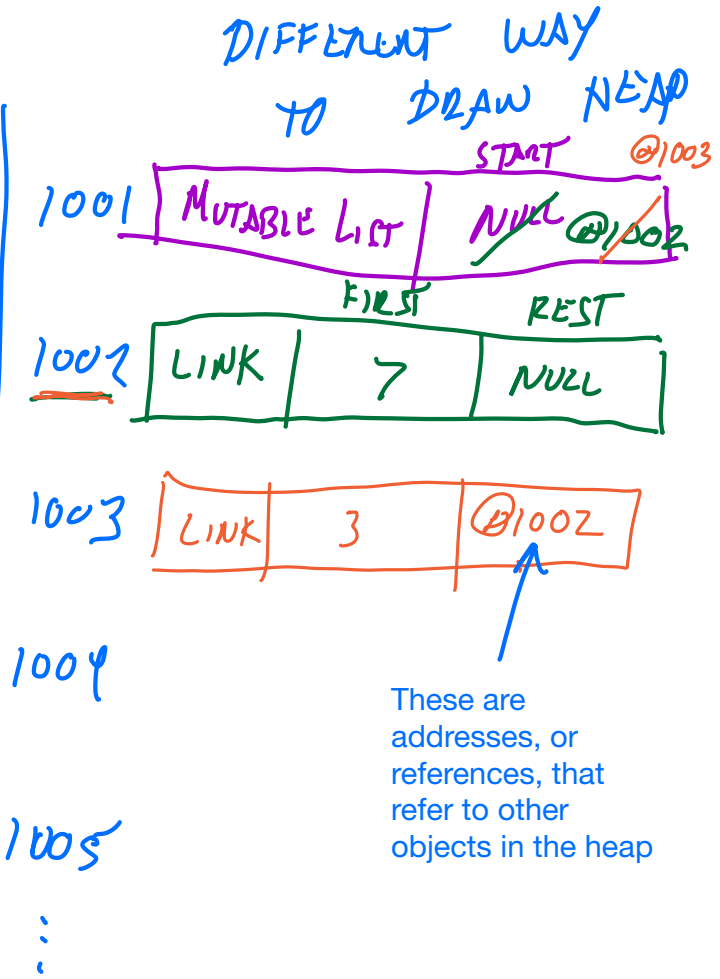
Lecture 10 – Addresses, Equality and ArrayLists

Lesson: Memory Diagrams with Addresses Explicit

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



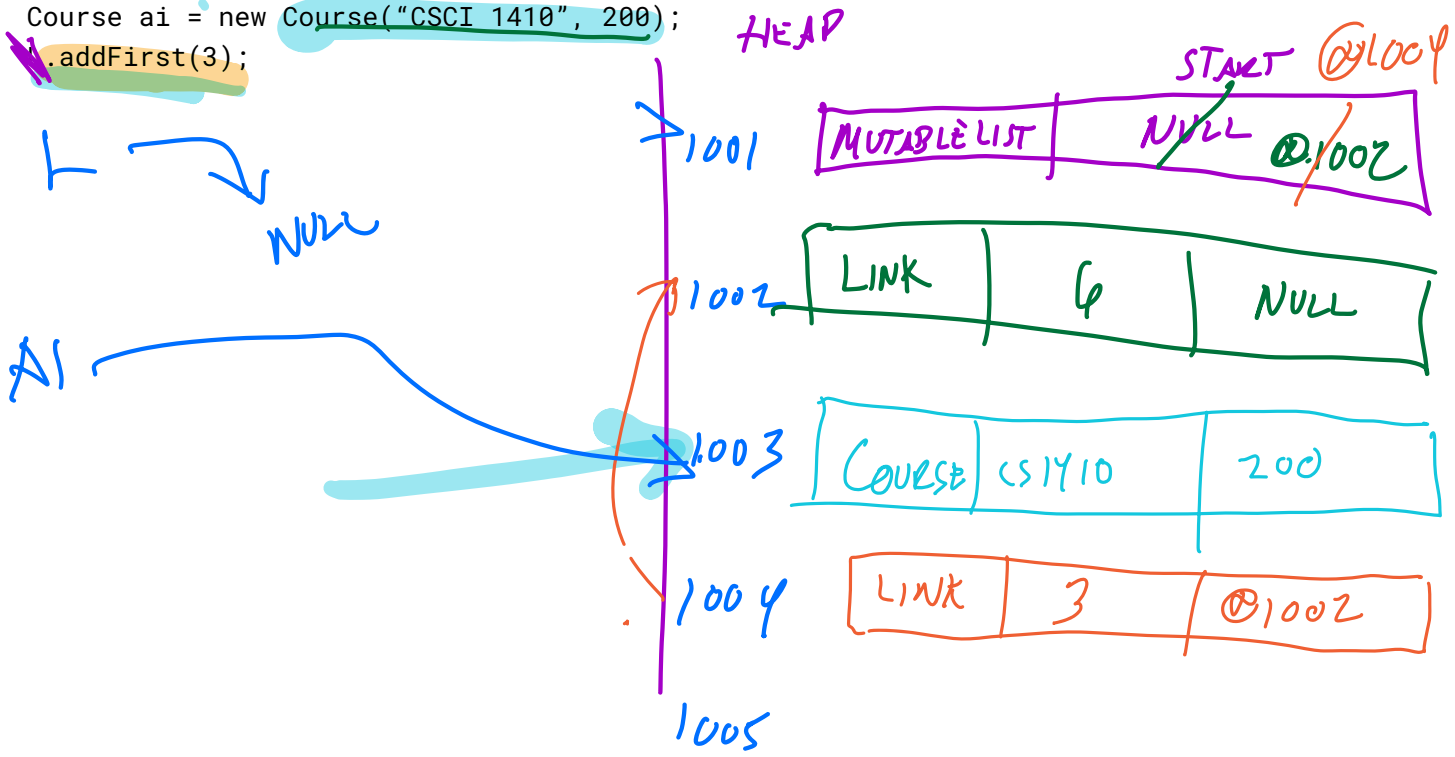
Can think of the heap as a series of addresses
- An address is a label for a specific spot in computer memory
- Every object lives at one address



These are addresses, or references, that refer to other objects in the heap

Activity: Draw the memory diagram with addresses for the following program

```
public void Example2() {
    MutableList<Integer> L = new MutableList<>();
    L.addFirst(6);
    Course ai = new Course("CSCI 1410", 200);
    L.addFirst(3);
}
```



=> When we make new objects ("new") we use the next space in the heap
=> Addresses (or slots) in the heap are used ("allocated") in the order in which the code is run (when we call "new")

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

```
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()));
}
```

DIFF NOTATIONS FOR WHAT IT MEANS FOR OBJECTS TO BE EQUAL?

STRING == STRING

L1 →

L2 →

@1020	MutableList(start: @1022)
@1021	Node(item: 6, next: null)
@1022	Node(item: 8, next: @1021)
@1023	MutableList(start: @1025)
@1024	Node(item: 6, next: null)
@1025	Node(item: 8, next: @1024)

(1) L1 == L2: "are L1 and L2 at the same location in memory". (Also called "Address comparison" "pointer comparison")
=> No, this is false

(2) .equals: Allows programmer to control what equality should mean for this type of object. ("Structural comparison")
=> Programmer would need to write equals method in MutableList (look at all elements, make sure data is the same…).

(3) Comparing strings with == will almost always fail => strings are objects, they live at different locations in memory. (== is okay for int, bool, float, …)
=> Should compare strings with .equals, ie. str1.equals(str2). This checks if the strings have the same characters

Course c1 = new Course("cs200", 80)
Course c2 = new Course("cs200", 84)

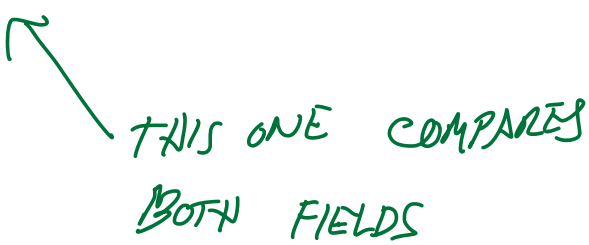
Should c1.equals(c2) be true?

=> As programmers, we COULD define .equals to just compare the course name and not the enrollment. This is a decision we would need to make when we write the equals method

Here's an example of writing a Course class with an equals method:

```
public class Course {
   private String name;
   private int enrolled;

   @Override
   // Example of an equals method. Since equals can be called with any
   // other object as the argument, we use type Object for the parameter
   public boolean equals(Object otherObj) {
       if (!(otherObj instanceof Course)) {
           // if otherObj isn't a Course, this and otherObj aren't equal
           return false;
       } else {
           Course otherC = (Course)otherObj; // tell Java otherObj is a Course
           return (this.name.equals(otherC.name)) &&
                   // eliminate next line if only want to compare on course numbers
                   (this.enrolled == otherC.enrolled);
       }
   }
}
```



Review: Continuing from last lecture–memory layouts of lists

WHAT SEQUENCE OF ADDFIRST/ADDLAST CALLS

Consider the following layouts for the list [8, 3, 6, 4] – what program might generate this heap layout?

@1012	**MutableList**(start:@1017)
@1013	Node(item:6, next:@1016)
@1014	Node(item:3, next:@1013)
@1015	Course(name: "CSCI1410", enrollment: 200)
@1016	Node(item:4, next:null)
@1017	Node(item:8, next:@1014)
@1018	

Can follow references to see order of elements in list

ORDER IN HEAP ≠ ORDER IN WHICH OBJECTS WERE CREATED.

ADDFIRST(6) OR ADDLAST(6) [6]

ADDFIRST(3) [3,6]

ADDLAST(4) [3,6,4]

ADDFIRST(8) [8,3,6,4]

Question: How would this memory layout be different if we were making an *immutable* list with the same sequence of addLast/addFirst calls?

Question: Imagine this list were named `L` in the environment. What sequence of memory objects get visited to compute `L.get(2)` [which should return 6]?

INDEX 0 1 2 3

[8, 3, 6, 4]

Can't just see which element is element 2 by looking at the heap => need to follow the chain of references (many colors or arrows above) to find out!

=> **This means we need to search the whole list, which has linear runtime!**

=> $\Theta(N)$

Activity: Now imagine the list had the following layout in memory (all the items consecutive and in order). What sequence of memory objects would get visited to compute `L.get(2)`?

@1012	**ConsecList**
@1013	8
@1014	3
@1015	6
@1016	4
@1017	
@1018	

[8,3,6,4]

Where is element 2 in this list?
Just from the picture we can see it's at @1015

Because this implementation has the array elements in consecutive slots, we can figure out element 2's address just by taking the address where the list starts and adding to it:

@1015 = @1012 + 2 + 1

ADDRESS OF ELEMENT 2 — START OF LIST

Therefore, we can implement get(i) by looking up the element at (address of list) + i + 1

=> This just involves adding a constant value to an address => **constant runtime!!**

1 + 1 + 1 => $\Theta(1)$

These two kinds of lists organize elements differently. As such, they will need different implementations of the core list methods (get, addLast, etc). For now, consider get, which takes a position in the list and returns the element at that index.

- In the first organization (MutableList, but also applies to LinkList), the elements aren't in order, so we have to follow references until we count off to the position we asked for. This means get(index) is linear in the size of the list (since we could have asked for the last index).

- In the second (ConsecList), we know exactly where each position is stored. Better still, we can compute the address where a position lies. If we want position p, for example, it is in address

  ```
  ConsecList-addr + 1 + p
  ```

  This means that get is constant time in a ConsecList.

The rest of this lecture and most of next will be on how to build ConsecList.