Imagine you were testing addFirst:

```
@Test
public void testAddFirst() {
    Link L1 = new EmptyList().addFirst(5);
    Assert.assertEquals(5, L1.getFirst())
}
```

Is this sufficient? What could have gone wrong?

In most assignments, we'll grade your tests in two ways:
1. We'll check your tests against our solution code.
   => This checks your understanding of the assignment/the method you're trying to write

=> WHEAT

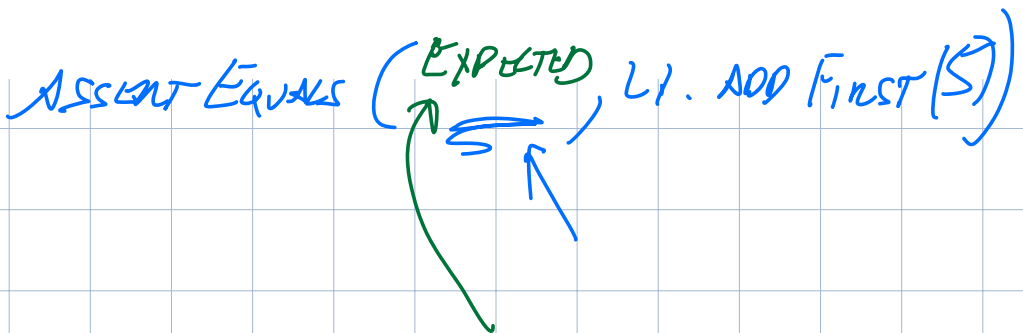
2. We'll check your test against broken solutions, to see if your tests can catch them
   => This checks if your tests are comprehensive enough to capture common failures

=> CHAFFS

Imagine you were testing addFirst:

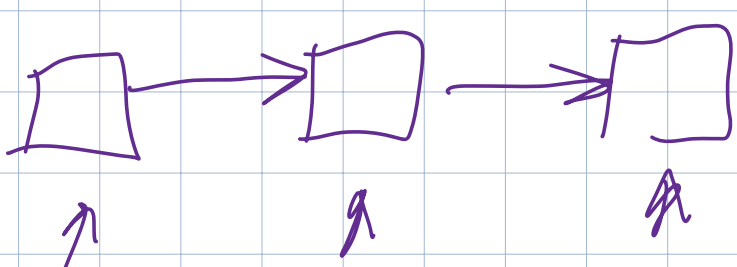
```
@Test
public void testAddFirst() {
    Link L1 = new EmptyList().addFirst(5);
    Assert.assertEquals(5, L1.getFirst())
}
```

Is this sufficient? What could have gone wrong?



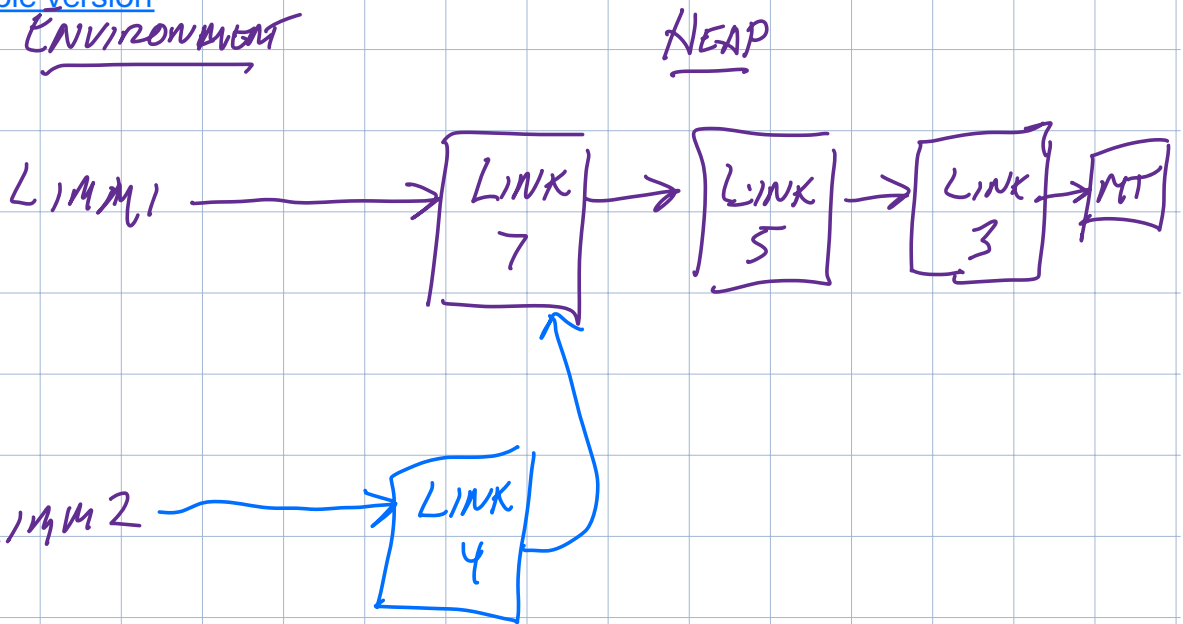
Could write: expected = new EmptyList().addFirst(5)
… but this is the same as L1.addFirst(5)!
We can't use addFirst to build our expected list, because this is what we're trying to tes in the first place

One way: use methods you've already built and tested to test new functionality:
- assertEquals(1, l1.size())
- assertEquals(5, l1.contains(5))



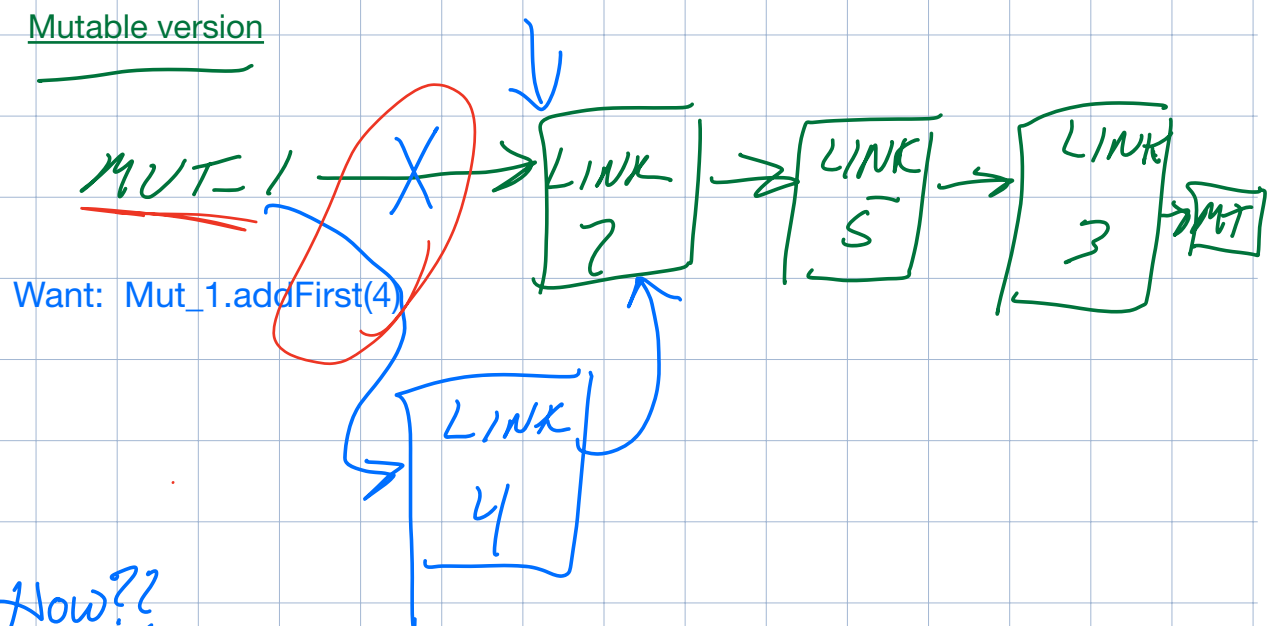
IMMUTABLE VS. MUTABLE LISTS

Immutable version



=> In an immutable list, the names for our lists always point to the same data
Even though we built L_im2 from L_im1, L_im1 has not changed.

Mutable version



Want: Mut_1.addFirst(4)

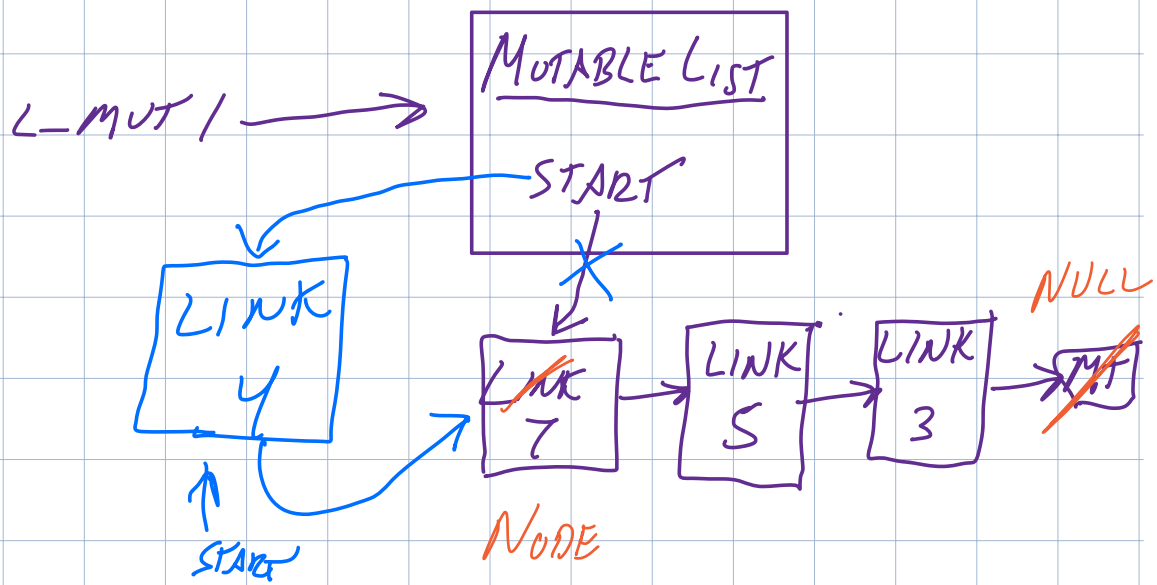
How??

L_mut1 = L_mut1.addFirst(4)

=> In the mutable version, the original name (Mut_1) changes to point to the new updated list!

How do we think about building this?

Can create a new class, MutableList, that has a field "start". This field points to the start of the chain of links.



When we call addFirst, we make a new link and update "start" to point to it.

=> This way, the name for the list (L_mut1) doesn't need to get updated when we change the list.

Why do we do this? When we build a data structure like this, we're building it for other developers to use it in their code—we don't want other developers to need to worry about the internal structure of the list, or think about updating names, we just want them to be able to call addFirst (just like you do for LinkedList and other classes in Java).

The idea is that we, as creators of the data structure, can write this tricky and annoying code and then "abstract it away" so others don't need to deal with it. This is an important principle when developing code for other people to use.

Extra note: key differences between this code and HW2:

- HW2 calls the "Link" objects "Node"
- People often don't like to make objects to represent the empty list (why use memory on this?). In HW2, the end of the list is represented by null
- HW2's mutable list isn't just for integers. It uses "generic types" to make the list *parameterizable*, meaning it can be used to hold any single type of object. We'll talk about what this means soon. For now, just know that this means that you should think of HW2's MutableList as behaving like other generic Java data structures, like LinkedList<Integer>, LinkedList<String>, …