Follow-up questions to Lecture 8-2

The purpose of this document is to close some loose ends of lecture 8-2.

In some of the lecture slides, the client output was displayed as, for example, `(3,5,9)`. Is this an error?

Yes, this was a Powerpoint error. The correct output, as generated by the `toString()` method, is `(3 5 9)`. It was corrected in the current lecture 8-2 slides in the course website.

The `Node` class begins with a declaration of a field whose type is `Node`. How can a definition of X include a referece to X?

The answer to this question is both simple and deep. Indeed, we see here a recursive definition. So, we should ask: What is the *base case* of this recursive definition? The answer is based on two unrelated issues: (1) In object-oriented programming, the `null` value is considered an object, *of any type*. (2) In Java, when you declare an object variable, say, "`Obj foo;`", the compiler treats it by default as "`Obj foo = null;`". Taken together, these two observations imply that declaring "`Node next;`" has the effect of creating a `Node` object that points to nothing, which is exactly what we want.

The `List` class feature a `first` pointer. Can we also have a `last` pointer that points to the last element in the list?

Yes, this makes sense. A `last` pointer will make some of the list operations more efficient. The price that we'll have to pay is updating this pointer, when needed.

The linked lists that we saw in the lecture allow going only forward in the list. Why can't we have lists in which we can go both forward and backward?

We can, and this will require writing another class that we can all `DoublyConnectedList`. You can think for yourself how such a list can be implemented. Tip: It must have the `last` field that we mentioned in the previous question, as well as other changes in both the `List` and `Node` classes. The price is that all the list management methods will need a few more lines of code, and will take more time to execute. So, should we do it? This a classical cost-benefit dilemma, and the answer is that we have to do what the application requires.

Does a listiterator advances the iteration? How does it do it?

In the original lecture 8-2 slides, the `ListIterator` API said nothing about advancing the iteration/iterator. This indeed was an omission / sloppiness error. In the current lecture slides, the documentation of the `next()` method was modifed to include a comment about the fact that `next()` indeed moves the iteration forward. How does it do it? See the `ListIterator` class implementation.

The code that was shown in the lecture 8-2 slides includes `package` and `import` statements. The code that was given in the course website does not use packages. What is the reason for this discrepancy.

The code that was shown in the lecture 8-2 slides is the right way to create a package of classes that enable client programs to manage lists. However, working with packaged classes involves various compilation nuances, and students often spend hours figuring out technical compilation errors that have nothing to do with the code proper. Therefore, to minimize headache, we dropped the `package` and `import` declarations from the hand-out lecture 8-2 code.

Why did Shimon end the lecture at 9:15 instead of 9:30?

Because he got confused with the Monday lecture, that ends at 11:15.