CSCE 156 – Lab 11: Linked Lists

0. Prior to the Laboratory

1. Review the laboratory handout
2. Read the following wiki entry on linked lists:
   http://en.wikipedia.org/wiki/Linked_list

1. Lab Objectives& Topics

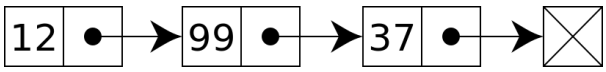
Following the lab, you should be able to:

- Use Linked Lists to store/retrieve/manipulate large collections of objects
- Implement Java interfaces

2. Problem Statement

List ADTs provide functionality for dealing with collections of objects in an object-oriented manner. In contrast to "static" arrays that have a fixed size and require the client code to "book-keep" the array, a List ADT provides an interface to add, remove, and retrieve elements while encapsulating (hiding) the details of how it does it. In an array-based list, the list would internally resize and retract the array as necessary. In a linked list,elements are added by creating nodes and manipulating pointers.

A linked list is typically implemented using *nodes* which contain elements and a reference to a another node (the "next" node). A linked list maintains a reference only to a *head* node. A small example of a linked list containing integers:



In this lab, you will implement a Linked-List ADT that holds `Truck` objects and implements several standard methods. Your list implementation is intended to be used in a larger inventory and truck management application, so you need to thoroughly test your implementation before you deliver it.

Activity 1: Implement a Linked List

Download and import the Eclipse project available on Canvas. Most of the code has already been provided for you. A`Truck` and `TruckListNode`representing Trucks and Linked List nodes holding trucks respectively have been provided for you. What you will need to do is finish the implementation of the `TruckList` class. Specifically you will need to implement the following methods:

- `clear` – This method will clear the entire list. After calling it, the list should be empty
- `addToStart` – This method should add the given truck to the front of the list
- `addToEnd` – This method should add the given truck to the end of the list

- `remove` – This method should remove the truck at the specified position, assuming the list is indexed starting at 0. This method should throw an `IndexOutOfBoundsException` if an invalid position is provided
- `getTruck` – This method should return the truck at the specified position, assuming the list is indexed starting at 0. This method should throw an `IndexOutOfBoundsException` if an invalid position is provided
- `print` – This method should print the list to the standard output in a human readable format (hint: make use of the `Truck.toString()` method).

Hint: You should start by implementing the "helper" methods so that you can utilize them in these functions.

Activity 2: Testing Your Implementation

To make sure that your implementation works, you should utilize the utilities and other tools provided to design and write several test cases. You will place these test cases into the Driver class and make sure that the results are as expected. You will need to write your own test cases. As you write your test cases, keep the following in mind.

- What are the "corner case(s)" that should be tested? A corner case is a pathological case that would occur only under special circumstances and may require special functionality.
- Is it a good idea to test cases in which you know an exception will be thrown? Why or why not? How could you test them?
- For this activity, a visual inspection suffices, but how might you *automate* such testing to eliminate human error in the process?

To help you write test cases, a few tools have been provided to you.

- The Driver class gives you an example of how to instantiate and use your `TruckList` class
- The Truck class has a static "factory" method that creates a truck with a random license plate that you can use in your test cases
- The Truck class has a special idiom (software design pattern) built into it: the Builder Pattern. The more member fields that an object has, the more difficult it is to write consistent and readable code to call its various constructor(s). The builder pattern allows you to use a Fluent Pattern style to build an object by calling "setters" on an inner-builder class prior to actually building the object. Objects that have a builder pattern are easier to use and construct. The Driver class contains an example on how to use the builder pattern.

Advanced Activity (Optional)

The linked list you have implemented is constructed of nodes which can only contain instances of the Truck class. Modify the linked list to accommodate any type using generics. In simple terms, a generic can be thought of as a variable type. An example of a generic for an *ArrayList* is:

```
ArrayList<MyType> listOfMyType = new ArrayList<MyType>();
```

This statement constructs an *ArrayList* which can contain objects of *MyType.*You should rename your TruckList and TruckListNode to MyList and MyListNode. You then need to add a generic type to the implementation of *MyList* and *MyListNode* as follows.

```
class MyList<T> { … }
```

```
class MyListNode<T> { … }
```

The extra `<T>` at the end of these class definition indicated a generic *T* will be used in throughout each of those class definitions. When you need to construct a type of *T* in one such class you write:

```
class MyListNode<T> {

private T item;

//...

}
```

The *T* generic is a place holder for the type which a use specifies. In the following code snippet a *MyListNode* is generated such that it can store objects of *MyType*:

```
MyListNode<MyType> listNode = new listNode<MyType>();
```

in your implementation of *print()* simply print out the string representation of the objects in your linked list using the *toString()* method of each object.