`Prologue: interfaces

Example: the CarlStack interface

- Yes, CarlStack.java has all the methods you would expect for class Stack.
- CarlStack.java is an interface because of the use of the interface keyword but the code inside the curly braces is set up for CarlStack to be a Class.
- Yes, CarlStack.java compiles but it shouldn't.
- Adding a hello-world main works but once again it shouldn't because interfaces can only access methods if they are implemented by a class beforehand, which in this code they are not.

The CarlStackLL implementation:

1. Yes, one can tell the CarlStackLL is an implementation of the CarlStack interface because CarlStackLL implements all the methods used in the CarlStack interface.
2. Yes, CarlStackLL does implement all the methods used in the CarlStack interface.
3. CarlStackLL uses a LinkedList to store all the items in the stack.
4. Yes, CarlStackLL.java does compile.
5. Compiling CarlStackLL.java once commenting out the peek method does not work and produces the error: "CarlStackLL is not abstract and does not override abstract method peek() in CarlStack /public class CarlStackLL<T> implements CarlStack<T> / ^ / where T is a type-variable: / T extends Object declared in class CarlStackLL" Trying to compile CarlStackLL.java after commenting out the peek method returns this error because there is a peek method in the CarlStack interface called at the beginning of CarlStackLL but no peek method is implemented in the CarlStackLL file, therefor there is an error because the "T" Object (which is peek) cast as a type-variable in CarlStack has no method declared in CarlStackLL.

The CarlStackAL and CarlStackALBad

6. CarlStackAL and CarlStackALBad are classes, not interfaces.
7. CarlStackAL and CarlStackALBad store their stack data in ArrayLists.
8. CarlStackAL and CarlStackALBad differ in that CarlStackAL pushes its data to the top bottom of its stack and executes its push, peek, and pop commands on the last object in the list first whereas CarlStackALBad pushes its new data to the top of its list (at position 0) and executes its push, peek, and pop commands on the first object in the list (at the bottom) first.
9. The initialCapacity parameter to one of the constructors in CarlStackAL and CarlStackALBad signifies the initial number of objects able to be stored in CarlStackAL and CarlStackALBad.

CarlStackTester and the complexity of implementations

10. You can execute CarlStackTester's if-block by including no arguments after entering "java CarlStackTester" in your command line, oppositely you can make its else block execute by including some number of arguments after entering "java CarlStackTester" in your command line.

11. The *testStack* method contains a parameter of type String. You are allowed to pass Strings into *testStack*.
12. The CarlStackAL did not do what I expected it to do, I expected it to pop first ant, then bat, cat, dog, emu, in that order.
13.
    a. N = 200,000
        i. CarlStackAL took 0.0080 seconds
        ii. CarlStackALBad took 2.1610 seconds
    b. N = 400,000
        i. CarlStackAL took 0.0110 seconds
        ii. CarlStackALBad took 8.9090 seconds
    c. N = 600,000
        i. CarlStackAL took 0.0150 seconds
        ii. CarlStackALBad took 23.2350 seconds
    d. N = 800,000
        i. CarlStackAL took 0.0170 seconds
        ii. CarlStackALBad took 49.3020 seconds
14. The tests are run faster for CarlStackAL because the way CarlStackAL runs, the objects entered into CarlStackAL's ArrayList are given the last position (on the top of the list) and are much easier to access than the objects put in CarlStackALBad's ArrayList, which are pushed to the bottom of each stack at position 0, and are therefore harder to access.
15. One call of push("*something*") in CarlStackAL will take O(1) time because it always adds the data to the end of the stack, meaning it should not change no matter how many items are already in the stack.
16. One call of push("*something*") in CarlStackALBad will take $O(N^2)$ time because it always adds the data to the front of the stack, meaning it will have to switch back all the N items in the stack testing for position before inserting the data at position 0, so each iteration of push in CarlStackALBad will grow exponentially in direct proportion to the size of the input data set.
17. One run through of *timeTestStack(new CarlStackAL(N), N)* in CarlStackTest will take O(N) time because it runs its for loop for every item in the stack (N), therefor the time it takes will grow linearly with respect to N.
18. My predictions to question 17 matches what I saw while running the program because running CarlStackTest with N = 200,000 took 0.0080 seconds on CarlStackAL and running CarlStackAL with three times the number of items (N = 600,000) took 0.0150 seconds which is around 0.0080 *3 plus whatever fraction of N is added outside the exponent ().
19. One run through of *timeTestStack(new CarlStackALBad(N), N)* in CarlStackTest will take $O(N^2)$ time because it runs its for loop for every item in the stack (N), therefor the time it takes will grow linearly with respect to N. This predictions matches what I saw while running the program because running CarlStackALBad with N = 200,000 took 2.1610 seconds and running CarlStackALBad with three times the number of items (N = 600,000) took 23.2350 seconds which is around 2.1610 $*3^2$ plus whatever fraction of N is added outside the exponent (½N).