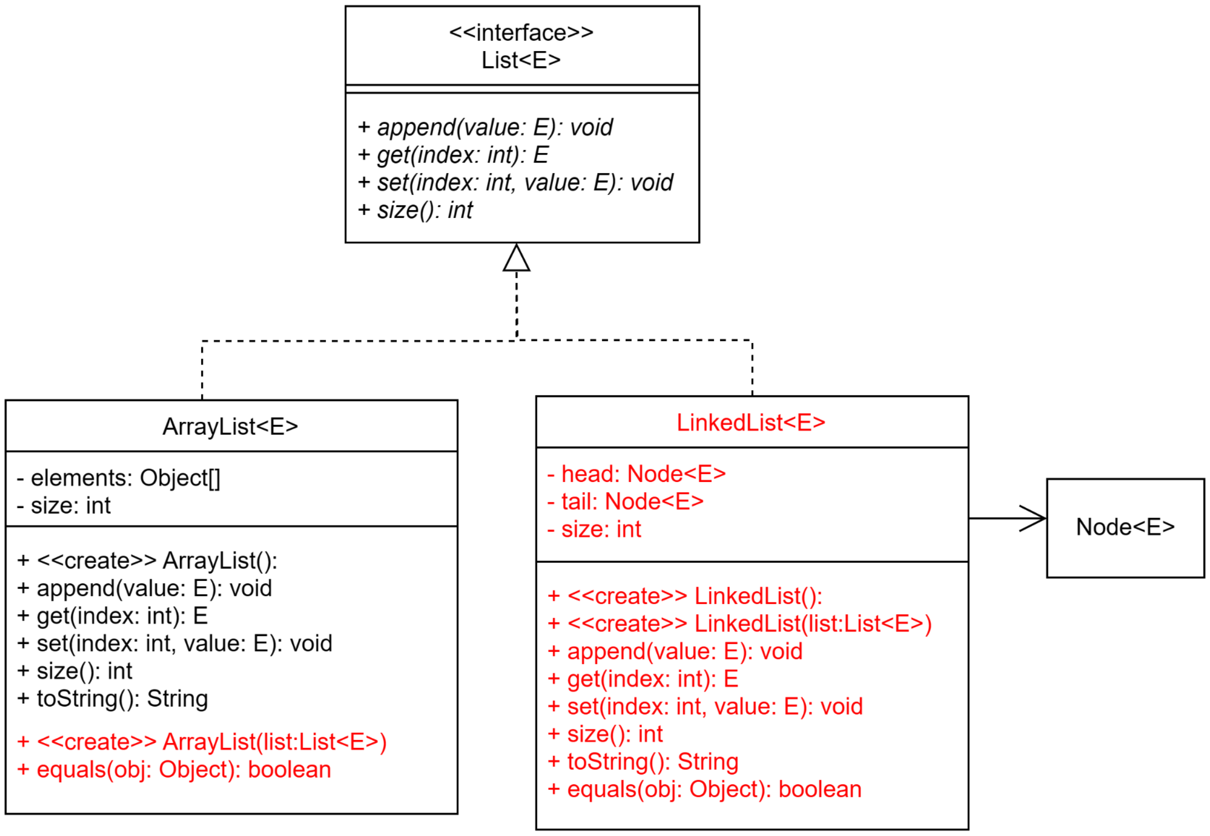
Lists

Goals of the Assignment

The goal of this assignment is to write an alternative implementation of the List abstract data type: the Linked List. First, you will upgrade the ArrayList implemented in class by adding special methods. Then you will write the alternative List implementation named LinkedList. Your new class will support all of the same functionality as the ArrayList. As always, you are expected to practice good software engineering, including unit testing and the Git workflow. Read this document ***in its entirety*** before asking for help from the course staff.

Activities

Here is a brief overview of this assignment.



1. Open your ArrayList class and add the following methods:
    a. `ArrayList(List<E> list)`, which creates an ArrayList with the specified list.
    a. `equals(Object o)`, which returns true if and only if the specified object is an instance of ArrayList and has the same size and *equal* elements as this list.

2. Begin by creating a new class, `LinkedList`, that implements the generic `List` interface that was created in class. Your new class must be fully generic. For now, just stub out all of the methods.

3. Unlike the `ArrayList` class you created during lecture, your `LinkedList` class will ***not*** use arrays. Instead, you will store values in a linked sequence of *nodes*. Use the same generic `Node` class that was used in the `NodeQueue` created in class. Add the following fields to your class:
    a. A `head` `Node`.
    b. A `tail` `Node`.
    c. The current `size` of the list.

4. Create a parameterless constructor that initializes all three fields. The `head` and `tail` should both initially be `null`, and the `size` should be `0`.

5. The easiest method to implement is `size()`; simply return the current size of the list.

6. The next easiest method to implement is the `append(E value)` method.
    a. Create a new `Node` to hold the new `value`.
    b. If the `size` of the list is `0`, the new `Node` becomes both the `head` and `tail` of the list.
    c. Otherwise, the new `Node` becomes the new `tail`. ***Hint***: don't forget to set the *new* `Node` as the *current* `tail`'s next `Node` *before* changing the `tail`.
    d. Don't forget to increment `size`.

7. The `get(int index)` method is slightly more complex than the other methods that you have implemented thus far. This is because a linked sequence of nodes does not support *random access* - there is no way to jump directly to a specific node in the sequence. Instead, you need to "walk the list" by starting at the head and counting nodes until you arrive at the correct index.

   You can accomplish this by creating a *counter* that starts at `0` and, beginning at the `head`, moving from one node to the next. Each time you move to the next node, increment the counter. When the counter is equal to the `index`, you have found the right node. If you reach the end of the list first, you should throw a `java.lang.IndexOutOfBoundsException`.

8. Implement the `set(int index, E value)` method. You will use an algorithm very similar to the one in the `get(int index)` method.

9. Implement the remaining methods based on the UML diagram shown above.

10. Write JUnit tests for each class and any non-trivial methods.

Submission Instructions

You must ensure that your solution to this assignment is pushed to GitHub *before* the start of the next lecture period.