

This lab has **seven (7)** checkpoints.

## Learning Outcomes

Upon completing this lab, students should be able to

- **Trace** recursive linked-list code.
- **Use** a set of JUnit tests to document the *design* of a non-trivial class.
- **Program** a line-focused Scanner-like that can ignore comments and permit line continuations.

## Introduction

1. Given a `LinkedList` class that holds `String` data with an `append` and a `length` method, what would the output of the following code be:

```
public class LinkedList {
    class ListNode { ... }

    private String toString(ListNode curr, int number) {
        if (curr != null) {
            return toString(curr.next, number - 1) +
                String.format("[%d] %s\n", number, curr.data);
        }
        return "";
    }
    public String toString() {
        return toString(head, length());
    }
}

public class Driver {
    public static void main(String args[]) {
        LinkedList linked = new LinkedList();
        linked.append("aardvark");
        linked.append("badger");
        linked.append("civet");
        linked.append("dromedary");
        linked.append("electric eel");
        System.out.println("<" + linked + ">");
    }
}
```

- ✓ Show the lab instructor the output and be prepared to explain what is happening.

### Solution:

Returns contents in reverse order but with smaller numbers:

```
toString("aardvark", 5) ->
  toString("badger", 4) + "[5] aardvark\n"
  toString("civet", 3) + "[4] badger\n"
```

```

    toString("dromedary", 2) + "[3] civet\n"
    toString("electric eel", 1) + "[2] dromedary\n"
    toString(null, 0) + "[1] electric eel\n"
    ""
    [1] electric eel
    [1] electric eel
    [2] dromedary
    [1] electric eel
    [2] dromedary
    [3] civet
    [1] electric eel
    [2] dromedary
    [3] civet
    [4] badger
    [1] electric eel
    [2] dromedary
    [3] civet
    [4] badger
    [5] aardvark

<[1] electric eel
[2] dromedary
[3] civet
[4] badger
[5] aardvark
>

```

2. How would you modify the *internal* version of `toString` to return items in the same format and same order but **separated** by dashes ("-") rather than **terminated** by new line characters?

Before you start: How many "\n" characters are in the string returned in the above code? Is that the same number of "-" characters *your* code will return?

- (a) Write the expected output with *your* code.

**Solution:**

```
<[1] electric eel-[2] dromedary-[3] civet-[4] badger-[5] aardvark>
```

- (b) Write the `toString` function to generate a dash-separated representation of the list.

**Solution:**

```

private String toString(LinkedNode curr, int number) {
    if (curr != null) {
        String prefix = toString(curr.next, number - 1);
        if (!prefix.isEmpty())
            prefix += "-";
        return prefix + String.format("[%d] %s",
    }
    return "";
}

```

```
}

```

✓ Show the lab instructor the *expected* output and your replacement code.

### 3. Your lead programmer sent you an email:

We need a `LineScanner` class built. It should be built with a `Scanner`, a `Reader`, or just a `String`. It should have two methods:

```
boolean hasNextLine()
String nextLine()
```

The `LineScanner` skips comments and concatenates continued lines into single lines returned by `nextLine`.

Attached is a test file. I need this by the end of the day.

Hopefully you have a lot of questions right now. But your lead just went for their weekly dip in the sensory-deprivation tank and will be unavailable until tomorrow morning.

Open `src/TestLineScanner.java` (you will be copying it into your Gradle project in a minute) and answer the following questions:

1. How many tests are there?
2. Are there any *annotations* (`\@...`) that you do not recognize? If so, figure out what they mean.
3. Just looking at the constructors you will need, what *fields* do you expect to need in `LineScanner`?
4. How are *comment lines* identified in the input read by `LineScanner`?
5. How are *line continuations* marked in the input read by `LineScanner`?
6. What happens if a comment line ends with a line continuation?

Given what you have figured out, what would be the sequence of lines returned by calling `LineScanner.nextLine()` repeatedly with the following underlying text:

```
// Dwarves are quite the singers
Far over the misty mountains cold
To dungeons deep\
and caverns old
    // but not too deep:
// Balrog!
We must away ere break of day
To find our long-forgotten gold

// The Elves and pine trees, amiright?\
The pines were roaring on the height
The winds were moaning in the night
The fire was red,\
    it flaming spread
The trees like torches blazed with light
// The End!
```

✓ Show the lab instructor your answers to these questions. Expect to be asked about annotations.

**Solution:**

There are ten (10) methods marked with `@Test`.

`@Tag` is new: it permits different tests labeled, run, and, most importantly here, filtered. We can run just tests w/ certain tags.

Fields: something to read from. A `Scanner` so `nextLine` works.

Comments begin with the regex `\s*//` (whitespace and then two slashes; Java single line format).

A line ending with `"\"` marks a continued line.

Comments are not continued. Being a comment takes precedence.

```
Far over the misty mountains cold
To dungeons deepand caverns old
We must away ere break of day
To find our long-forgotten gold
```

```
The pines were roaring on the height
The winds were moaning in the night
The fire was red, it flaming spread
The trees like torches blazed with light
```

4. Create a Gradle project. The main program should be `driver.LineScannerDriver` and you need two packages: `driver` and `io`.

`TestLineScanner.java` goes in `src/test/java/io/TestLineScanner.java`.

`build.gradle`:

Add the following to the `plugins` section:

```
id 'com.adarshr.test-logger' version '4.0.0'
```

Replace `test` section with the following:

```
test {
    useJUnitPlatform {
        includeTags("constructor")
        // includeTags("constructor","standard","comments","continued")
    }

    testLogging {
        showExceptions true
        showCauses true
        showStandardStreams true
    }
}
```

Notice the `includeTags` line (and following comment): it does what you think it would.

Add `io.LineScanner.java` in the main code. To get it to compile with the tests, add three *empty* constructors with the appropriate parameters.

Also add *stub* (do-nothing) versions of `nextLine` and `hasNextLine`.

The **constructor** tagged tests should compile and pass.

✓ Show your working code to the lab instructor.

5. What *fields* does `LineScanner` need? The question comes down to what does such an object **do** with the parameters with which it was constructed. The `LineScanner`, with suggestively named methods like `nextLine` behaves like a `Scanner`: it keeps track of some source of *input* and *reads* it.

For a starter, wrapping a `Scanner` inside the class and forwarding the two methods to that `Scanner` might work: fill in the `Scanner`-based constructor as the primary constructor. Then have the other two constructors forward the work to the primary constructor after constructing a new `Scanner`; fill-in the stub methods to return the results of calling the routine on the `Scanner` with the same name.

Add the standard tests in `build.gradle`. Get the code to compile, run, and pass the tests.

✓ Show the lab instructor your passing tests and code.

6. Ignoring comments requires “reading ahead” in the wrapped `Scanner`. In class we looked at `ScanPastComments`. It had a *next-line* field that held the next line. Whenever a line was needed and that field was `null`, the inner `Scanner` was read, skipping comments, until a line was found or EOF was reached.

How do you know a line is a comment? `String.startsWith` is your friend.

Modify your code to read the next line (when necessary), keeping it around and returning it, and skipping comment lines. Tests tagged with `comments` should now be able to pass. Feel free to add additional, smaller, test functions if you need more, finer-grained information on what parts of your code works.

✓ Show the lab instructor your passing tests and code.

7. Finally, turn on the `continued` test tag. Run the tests and make them pass. Make sure you understand why each has the given expected value.

✓ Show the lab instructor your passing tests and code.

8. [Not a separate checkpoint but necessary] Add and commit your full `gradle` project (if you `git add` just the root directory, `., git` will add all subdirectories recursively).

Create a repository in your Gitea turn-in organization, associate it with your lab work, and push the repo. This will be done at the end of every lab.