Programming Assignment 1: Mini-Git

Specification

Background

Version control systems are software features or programs designed to track changes to documents or sets of documents over time. In most systems, each time changes are made to the documents being tracked, a new *version* or *revision* is logged. Usually some additional information, called *metadata*, is also tracked along with each revision. This metadata can include a timestamp for when the changes were made, one or more authors of the changes, comments or notes about the changes, and/or many other types of information. Version control systems also typically provide a way to review the *history* of the documents being tracked, along with operations to revert to previous points in history if necessary. The history tracking features of Google Docs are an example of a version control system.

Version control systems that are designed specifically for tracking source code for computer programs are often called *source control systems* and may include additional features useful for tracking source code. These features may include associating certain types of files with particular programming languages or running automated tests each time a new revision is created. One popular source control system in wide use today is Git, which was developed by Linus Torvalds (who also created the Linux operating system) and initially released in 2005.

In this assignment, we will implement our own, simplified version of a version control system similar to Git, using linked-lists.

> **Note:** Version control systems typically need to address at least two significant problems: how to track and manage the metadata for the revisions that make up the version history, and how to represent and track the actual changes to the documents themselves. We will focus only on the first problem (tracking metadata and history); for more information on how Git handles tracking the changes, see the free, online book Pro Git.

System Structure

In our system, as in Git, a set of documents and their histories are referred to as a *repository.* Each revision within a repository is referred to as a *commit.* You will implement a class called `Repository` that supports a subset of the operations supported by real Git repositories. (We will not be dealing with features such as branching or remote repositories. We will assume histories are fairly linear and mostly take place in a single, local repository.)

We will represent commits with following provided class. **You must not modify this class in any way.**

```
public static class Commit {
    public final long timeStamp;
    public final String id;
    public final String message;
    public Commit past;

    public Commit(String message, Commit past) {
        this.id = getNewId();
        this.message = message;
        this.timeStamp = System.currentTimeMillis();
        this.past = past;
    }

    public Commit(String message) {
        this(message, null);
    }

    @Override
    public String toString() {
        SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd 'at' HH:mm:ss z");
        Date date = new Date(timeStamp);

        return id + " at " + formatter.format(date) + ": " + message;
    }

    private static String getNewId() {
        return UUID.randomUUID().toString();
    }
}
```

Each commit consists of a unique identifier, a message describing the changes, the general time the commit was made, and a reference to the immediately previous commit. In our representation, identifiers will be strings.

Note: You may see some code you're unfamiliar with (namely the `SimpleDateFormat`, `Date` classes and `UUID`), but that's okay! You are not required to understand these, just know that `SimpleDateFormat` and `Date` allow us to print out the current date and `UUID` is used to generate random, unique ids for our commits. Feel free to explore these classes or ask the course staff if you'd like to learn more about them!

As we eventually did with our `LinkedIntList` and `LinkedNode` classes, we will implement the `Commit` class as a public static inner class within the `Repository` class. (Ideally, we would make this class private, but we leave it public for ease of testing.)

Notice that the `id` and `message` fields of the `Commit` class are all `final`, meaning that you will not be able to modify them. If you attempt to change the value of these fields after they have been initialized, you will get a compiler error such as the following:

```
error: cannot assign a value to final variable message
```

Required Operations

Your `Repository` class must include the following methods:

```
public Repository(String name)
```

- Create a new, empty repository with the specified name
  - If the name is null or empty, throw an `IllegalArgumentException`

```
public String getRepoHead()
```

- Return the ID of the current head of this repository.
  - If the head is `null`, return `null`

```
public int getRepoSize()
```

- Return the number of commits in the repository

```
public String toString()
```

- Return a string representation of this repository in the following format:
  - `<name> - Current head: <head>`
    - `<head>` should be the result of calling `toString()` on the head commit.
  - If there are no commits in this repository, instead return `<name> - No commits`

```
public boolean contains(String targetId)
```

- Return true if the commit with ID `targetId` is in the repository, false if not.

```
public String getHistory(int n)
```

- Return a string consisting of the String representations of the most recent `n` commits in this repository, with the most recent first. Commits should be separated by a newline (`\n`) character.
  - If there are fewer than `n` commits in this repository, return them all.
  - If there are no commits in this repository, return the empty string.
  - If n is non-positive, throw an `IllegalArgumentException`.

```
public String commit(String message)
```

- Create a new commit with the given message, add it to this repository.
  - The new commit should become the new head of this repository, preserving the history behind it.
- Return the ID of the new commit.

```
public boolean drop(String targetId)
```

- Remove the commit with ID `targetId` from this repository, maintaining the rest of the history.
- Returns `true` if the commit was successfully dropped, and `false` if there is no commit that matches the given ID in the repository.

```
public void synchronize(Repository other)
```

- Takes all the commits in the `other` repository and moves them into `this` repository, combining the two repository histories such that chronological order is preserved. That is, after executing this method, `this` repository should contain all commits that were from `this` and `other`, and the commits should be ordered in timestamp order from most recent to least recent.
    - If the `other` repository is empty, `this` repository should remain unchanged.
    - If `this` repository is empty, all commits in the `other` repository should be moved into `this` repository.
    - At the end of this method's execution, `other` should be an empty repository in all cases.
    - You should not construct any new `Commit` objects to implement this method. You may however create as many references as you like.

`synchronize` Explained

> Note that while the other operations are real Git ones, `synchronize` is not. `synchronize` is a great exercise, but does not mirror any functionality a real git repository would ever want to do.

Since this operation is somewhat complicated, consider an example. Assume we have the following two repositories with their own history:

```
Repository #1 (repo1)
f6df5ece-b614-4587-8c59-8530eb9c5f5e at 2023-07-02 at 18:28:39 PDT: Add initial scaffold
d3aa237c-0419-4fe8-b133-e4bd44f38f61 at 2023-07-02 at 16:25:43 PDT: Edit README
99eb454c-db6e-4a7e-ae01-e8ece007451b at 2023-06-30 at 03:45:12 PDT: Upload README

Repository #2 (repo2)
0f29d5e8-e79e-404e-8c3e-2e655e622a88 at 2023-07-01 at 12:04:28 PDT: Edit documentation
ea16b915-2f75-4648-9326-082b2beb4d3b at 2023-04-21 at 07:21:12 PDT: Upload documentation
```

Then, we synchronize Repository #2 into Repository #1 (`repo1.synchronize(repo2)`). Our repositories would now have the following histories:

```
Repository #1 (repo1)
f6df5ece-b614-4587-8c59-8530eb9c5f5e at 2023-07-02 at 18:28:39 PDT: Add initial scaffold
d3aa237c-0419-4fe8-b133-e4bd44f38f61 at 2023-07-02 at 16:25:43 PDT: Edit README
0f29d5e8-e79e-404e-8c3e-2e655e622a88 at 2023-07-01 at 12:04:28 PDT: Edit documentation
99eb454c-db6e-4a7e-ae01-e8ece007451b at 2023-06-30 at 03:45:12 PDT: Upload README
ea16b915-2f75-4648-9326-082b2beb4d3b at 2023-04-21 at 07:21:12 PDT: Upload documentation
```

```
Repository #2 (repo2)
```

Notice that Repository #1 now contains all of the commits, while Repository #2 is empty. Additionally, the order of commits in Repository #1 is based solely on their time stamps, from most recent to least recent. Since Repository #2 is now empty, we did not construct any new commits, only rearranging the ones that were initially present.

Client Program & Visualization

We have provided a client program that will allow you to test your `Repository` implementation by creating and manipulating repositories. The client program will directly call the methods you implement in your `Repository` class and will show you the resulting changes to the repositories. Click "Expand" below to see a sample execution of the client (user input is **bold and underlined**).

▼ Expand

```
Welcome to the Mini-Git test client!
Use this program to test your Mini-Git repository implemenation.
Make sure to test all operations in all cases --
some cases are particularly tricky.

Available repositories:
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: create repo1
  New repository created: repo1 - No commits

Available repositories:
repo1 - No commits
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: commit repo1
Enter commit message: First commit!
  New commit: 0

Available repositories:
repo1 - Current head: 0 at 2023-10-25 at 06:53:42 AEDT: First commit!
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: commit repo1
Enter commit message: Another commit.
  New commit: 1

Available repositories:
repo1 - Current head: 1 at 2023-10-25 at 06:53:46 AEDT: Another commit.
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: history repo1
How many commits back? 2
1 at 2023-10-25 at 06:53:46 AEDT: Another commit.
0 at 2023-10-25 at 06:53:42 AEDT: First commit!
```

```
Available repositories:
repo1 - Current head: 1 at 2023-10-25 at 06:53:46 AEDT: Another commit.
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: create repo2
  New repository created: repo2 - No commits

Available repositories:
repo2 - No commits
repo1 - Current head: 1 at 2023-10-25 at 06:53:46 AEDT: Another commit.
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: commit repo2
Enter commit message: Commit the third
  New commit: 2

Available repositories:
repo2 - Current head: 2 at 2023-10-25 at 06:54:00 AEDT: Commit the third
repo1 - Current head: 1 at 2023-10-25 at 06:53:46 AEDT: Another commit.
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: commit repo1
Enter commit message: Fourth commit
  New commit: 3

Available repositories:
repo2 - Current head: 2 at 2023-10-25 at 06:54:00 AEDT: Commit the third
repo1 - Current head: 3 at 2023-10-25 at 06:54:05 AEDT: Fourth commit
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: history repo1
How many commits back? 4
3 at 2023-10-25 at 06:54:05 AEDT: Fourth commit
1 at 2023-10-25 at 06:53:46 AEDT: Another commit.
0 at 2023-10-25 at 06:53:42 AEDT: First commit!

Available repositories:
repo2 - Current head: 2 at 2023-10-25 at 06:54:00 AEDT: Commit the third
repo1 - Current head: 3 at 2023-10-25 at 06:54:05 AEDT: Fourth commit
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: head repo1
3

Available repositories:
repo2 - Current head: 2 at 2023-10-25 at 06:54:00 AEDT: Commit the third
repo1 - Current head: 3 at 2023-10-25 at 06:54:05 AEDT: Fourth commit
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: commit repo1
Enter commit message: one more commit
  New commit: 4

Available repositories:
repo2 - Current head: 2 at 2023-10-25 at 06:54:00 AEDT: Commit the third
repo1 - Current head: 4 at 2023-10-25 at 06:54:13 AEDT: one more commit
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: drop repo1
Enter ID to drop: 3
  Successfully dropped 3
```

```
Available repositories:
repo2 - Current head: 2 at 2023-10-25 at 06:54:00 AEDT: Commit the third
repo1 - Current head: 4 at 2023-10-25 at 06:54:13 AEDT: one more commit
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: history repo1
How many commits back? 3
4 at 2023-10-25 at 06:54:13 AEDT: one more commit
1 at 2023-10-25 at 06:53:46 AEDT: Another commit.
0 at 2023-10-25 at 06:53:42 AEDT: First commit!

Available repositories:
repo2 - Current head: 2 at 2023-10-25 at 06:54:00 AEDT: Commit the third
repo1 - Current head: 4 at 2023-10-25 at 06:54:13 AEDT: one more commit
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: synchronize repo1
Which repository would you like to synchronize into the given one? repo2

Available repositories:
repo2 - No commits
repo1 - Current head: 4 at 2023-10-25 at 06:54:13 AEDT: one more commit
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: history repo1
How many commits back? 6
4 at 2023-10-25 at 06:54:13 AEDT: one more commit
2 at 2023-10-25 at 06:54:00 AEDT: Commit the third
1 at 2023-10-25 at 06:53:46 AEDT: Another commit.
0 at 2023-10-25 at 06:53:42 AEDT: First commit!

Available repositories:
repo2 - No commits
repo1 - Current head: 4 at 2023-10-25 at 06:54:13 AEDT: one more commit
Operations: [create, head, history, commit, drop, synchronize, quit]
Enter operation and repository: quit
```

In addition to this, you may (and are *encouraged*) to create your own client programs to to test out your implementation on various cases. You may also modify the provided client if you find it helpful. However, **your `Repository` class must work with the provided client without modification and must meet all requirements above**. To better understand what is happening, you can reference these slides which visualize the repositories changing throughout the operations.

Testing

On this assignment, we are expecting you to write tests covering the methods that you complete. To help facilitate this, we have provided you `RepositoryTest`, a scaffold file that contains testing examples you should reference when developing your own. Additionally, we have hidden the output to the Ed test covering `synchronize` with the intention that you are writing correctness tests for this method in particular. While *you'll still be able to see the overall test result in Ed*, you won't be able to use the testing error messages to help you debug this method as you might have done before. Note

that this means **there are no hidden tests,** just hidden output for the `synchronize` tests**.**

Implementation Guidelines

As always, your code should follow all guidelines in the Code Quality Guide and Commenting Guide. In particular, pay attention to these requirements and hints:

- The specified exceptions must be thrown correctly in the specified cases. Exceptions should be thrown as soon as possible, and no unnecessary work should be done when an exception is thrown. Exceptions should be documented in comments, including the type of exception thrown and under what conditions.
- You should not construct any unnecessary `Commit` objects. Specifically, you should only construct a `Commit` object when an entirely new commit is being created. If commits are being removed or rearranged, you should manipulate the existing `Commit` objects. (You may create as many *references* to `Commit` objects as you like.)
    - You should only need to construct `Commit` objects in the `commit()` method.
- Your `Repository` class should have exactly two fields as specified below and they should be declared `private`. You are not allowed to have any other fields.
    - A reference to the head of the repository.
    - A field to keep track of the repository's name.
- These methods can be quite challenging! It is valuable to take a look at resources from class, particularly the `LinkedIntList` pre-class work, in-class activities, and section problems. Notably, the `weave` problem in Section 8 will be a helpful starting point for implementing `synchronize`.
- **You should not modify the `id`, `message`, or `timeStamp`** fields directly. In particular if you run into the issue `error: cannot assign a value to final variable message`, it likely means that you are attempting to modify a `Commit` object's data, instead of rearranging the commits.
- Some notes on `synchronize`:
    - Note that you will have to compare the time stamps to determine which order they should appear in and that the `timeStamp` field is of type `long`. This is another primitive that you haven't seen before, but you can essentially treat it as an `int` when doing your comparisons. So, if you're trying to check if `commit1` is chronologically earlier than `commit2`, you can check if `commit1.timeStamp < commit2.timeStamp`.
- **You must use an iterative approach to this assignment**. While recursion is a powerful tool that we'll explore later in the course, we're specifically assessing your ability to reason about LinkedLists and the cases they generate.

Mini-Git

This code slide does not have a description.

Reflection

The following questions will ask you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

Please answer all questions.

Question 1

Have you ever used a version control system before? If so, in what contexts? What problems did it help you solve? If not, what are some situations in which it might be helpful to have a system like this?

No response

Question 2

How do you think a system like Mini-Git would need to be different if *multiple people* were committing to the same repository? Would the system still work? Would it need to be managed differently?

No response

Question 3

How do you think this assignment would have been different if we asked you to implement Mini-Git using an `ArrayList` of `Commit` objects instead of the linked structure we used? Which aspects would have been easier? Which aspects would have been more difficult? Which approach would you have preferred if given the choice?

No response

Question 4

Describe how you went about testing your implementation. What specific situations and/or test cases did you consider? Why were those cases important?

No response

Question 5

What skills did you learn and/or practice with working on this assignment?

No response

Question 6

What did you struggle with most on this assignment?

No response

Question 7

What questions do you still have about the concepts and skills you used in this assignment?

No response

Question 8

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

No response

Question 9

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

No response

Question 10

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

No response

Testing Comprehension

Building off of the last testing comprehension, below, you will fill in JUnit tests that we have written for Mini-Git. For syntax, take a look at the JUnit Cheat Sheet!

Here are some important ones for the problems below:

```
assertEquals(EXPECTED VALUE, TESTED VALUE, ERROR MESSAGE IF NOT EQUALS)
```

```
assertTrue(TESTED BOOLEAN VALUE, ERROR MESSAGE IF NOT TRUE)
```

For an example problem, let's take a look at `testDropFalse`, which tests the drop method in the cases where it should return `false`. The first two parameters in both assertions have been taken out!

```
@Test
@DisplayName("drop - false case")
public void testDropFalse() {
    Repository r = new Repository("r");
    assertEquals(???,
    "drop did not return false for an incorrect id and an empty repository");
    r.commit("c1");
    assertEquals(???,
    "drop did not return false for an incorrect id and a non empty repository");

}
```

In order to find out what to put instead of the "???", let's take a look at each message: in both cases, our test is outputting the message that `drop` is not returning `false` for an incorrect id (for some reason).

In other words, the test was expecting `false` to be returned when an incorrect id is being tested for the method `drop` . So, the answer for both of these is actually

```
false, r.drop("no id")
```

Where you can put any string instead of `"no id"` .

Now, here is a couple similar problems to try on your own... good luck!

Question 1

Choose the correct type of assertion for the testToString method below:

```java
@Test
@DisplayName("toString")
//Dependencies: commit
public void testToString() {
    Repository r = new Repository("r");
    YOURANSWER("r - No commits", r.toString(),
    "toString on an empty repository incorrect"); //does not depend on commit
    String id = r.commit("c");
    YOURANSWER("r - Current head: " + id + ": c", r.toString(),
    "toString after a commit incorrect");
}
```

- assertEquals
- assertTrue
- assertThrows
- assertNotEquals

Question 2

Fill in the parameters for the `assertEquals` below. Make sure to include the comma!

```java
@Test
@DisplayName("Constructor")
//Dependencies: getRepoHead
public void testConstructor() {
    Repository r = new Repository("r");
    assertEquals(/* ANSWER HERE */,
    "getRepoHead does not return null directly after constructor");
}
```

No response

Question 3

Fill in the constructor exceptions method below. Recall that the second parameter in assertThrows is called an *executable*, and is usually formatted like

```
() -> {/*CODE THAT SHOULD CAUSE EXCEPTION TO BE THROWN*/}
```

For example, we could test that dividing by zero throws an exception using

```
assertThrows(ArithmeticException.class, () -> {int x = 5 / 0;}, "Dividing by zero did not throw an
```

Fill in the two assertions below, with each answer on a **separate line**!

```
@Test
@DisplayName("Constructor Exceptions")
public void testConstructorExceptions() {
    assertThrows(IllegalArgumentException.class, /* ANSWER HERE */,
    "null passed into constructor does not throw exception");
    assertThrows(IllegalArgumentException.class, /* ANSWER HERE */,
    "\"\" passed into constructor does not throw exception");
}
```

No response

JUnit Cheatsheet

Learning a new concept can be overwhelming so we've compiled a cheatsheet you can reference while you write your own JUnit tests!

Creating a JUnit Test Class and JUnit Test Case

To create a JUnit test class, make sure you import `org.junit.jupiter.api.*` and `static org.junit.jupiter.api.Assertions.*`. These will give you access to method annotations like `@Test` and `@BeforeEach` and assertion methods like `assertTrue()` and `assertFalse()`.

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class ExampleTestClass {
    @Test
    public void yourTestCase() {
        // Assertion methods called here
    }
}
```

JUnit Method Annotations

Method annotations are used by JUnit so that JUnit knows how to treat your methods. Before you write a method, you must attach a method annotation. This special syntax is then interpreted by JUnit to know how to execute your method.

`@Test`: Turns a public method into a JUnit test case.

```
@Test
public void test() {
    ...
}
```

`@Timeout(time)`: Times the test such that the test will fail after `time` milliseconds. Thus, the code must finish execution before `time`. Note that you still need the `@Test`.

```
// Test will fail after 1000 ms
@Test
@Timeout(1000)
public void test() {
    ...
}
```

`@BeforeEach`: The method will be executed before each `@Test`

```
private int num; // This is a field

// This method will execute before each @Test
@BeforeEach
public void setUp() {
    num = 0;
}

@Test
public void test() {
    assertSame(0, num);
    num++;
    assertSame(1, num);
}
```

JUnit Assertion Methods

Assertion methods are the building blocks of JUnit and how you will write testing code inside your test methods. When you use an assertion method, the result needs to match up with what the assertion method expects, otherwise, your test will fail. Below are the most common types of assertion methods that you will use:

`assertTrue(test)`: Fails if the `test` is `false`

```
@Test
public void test() {
    int x = 2;
    String s = "Hello World";
    assertTrue(true);
    assertTrue(x == 2);
    assertTrue(s.equalsIgnoreCase("Hello World"));
}
```

`assertFalse(test)`: Fails if the `test` is `true`

```
@Test
public void test() {
    int x = 2;
    String s = "Hello World";
    assertFalse(false);
    assertFalse(x != 2);
    assertFalse(s.contains("a"));
}
```

`assertEquals(expected, actual)`: Fails if the `expected` and `actual` are not equal

```
@Test
public void test() {
```

```
    String s1 = "Hello World";
    String s2 = "Hello World";
    String s3 = "Hello World";
    assertEquals(s1, s2);
    assertEquals(s2, s3);
    assertEquals(s3, s1);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();
    for (int i = 1; i <= 5; i++) {
        list1.add(i);
        list2.add(i);
    }
    assertEquals(list1, list2);
}
```

`assertSame(expected, actual)` : Fails if the `expected` and `actual` are not equal using reference semantics (==)

```
@Test
public void test() {
    int x = 2;
    assertSame(2, x);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = list1;
    assertSame(list1, list2);
}
```

`assertNotSame(expected, actual)` : Fails if `expected` and `actual` are equal using reference semantics (==)

```
@Test
public void test() {
    int x = 2;
    assertNotSame(3, x);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();
    assertNotSame(list1, list2);
}
```

`assertNull(value)` : Fails if `value` is non-null

```
@Test
public void test() {
    Map<String, Integer> map = new HashMap<>();
    map.put("cse122", 1);
    assertNull(map.get("Hello World"));

    String s = null;
    assertNull(s);
```

```
}
```

`assertNotNull(value)` : Fails if `value` is null

```
@Test
public void test() {
    Map<String, Integer> map = new HashMap<>();
    map.put("cse122", 1);
    assertNotNull(map.get("cse122"));

    String s = "Hello World";
    assertNotNull(s);
}
```

`assertArrayEquals(Any[] expectedValues, Any[] actualValues)` : Fails if `expectedValues` and `actualValues` do not have the same elements, in the same order.

```
@Test
public void test() {
    int[] a = new int[] {1, 2, 3};
    int[] b = new int[] {1, 2, 3};
    assertArrayEquals(a, b);
}
```

`assertThrows(exception.class, () -> {code})` : Fails if `code` does not throw `exception`

```
@Test
public void test() {
    List<Integer> list = new ArrayList<>();
    assertThrows(IndexOutOfBoundsException.class, () -> {
        list.get(2); // List is currently: []
    });

    assertThrows(IndexOutOfBoundsException.class, () -> {
        list.add(1); // List is currently: [1]
        list.add(2); // List is currently: [1, 2]
        list.add(3); // List is currently: [1, 2, 3]
        list.remove(3); // Index 3
    });

}
```

Using JUnit to test Java's ArrayList Implementation:

Below is an example of a JUnit testing class that tests Java's ArrayList implementation:

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
```

```java
import java.util.*;

public class ArrayListTest {
    private static final int TIMEOUT = 2000;
    private List<String> list;

    @BeforeEach
    public void setUp() {
        list = new ArrayList<>();
    }

    @Test
    @Timeout(TIMEOUT)
    public void testAddingElements() {
        assertTrue(list.isEmpty());
        list.add("Hunter Schafer");
        list.add("Miya Natsuhara");
        list.add("CSE 122");

        assertEquals("Hunter Schafer", list.get(0));
        assertEquals("Miya Natsuhara", list.get(1));
        assertEquals("CSE 122", list.get(2));

        assertTrue(list.size() == 3);
    }

    @Test
    public void testContains() {
        assertTrue(list.isEmpty());
        list.add("CSE 122");

        assertTrue(list.contains("CSE 122"));
        assertFalse(list.contains("Hello World"));
    }

    @Test
    public void testNegativeIndexGet() {
        assertTrue(list.isEmpty());
        assertThrows(IndexOutOfBoundsException.class, () -> list.get(-1));
    }
}
```