Programming Assignment 1: Mini-Git

Specification

Background

Version control systems are software 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.

Learning Objectives

- Write a functionally correct Java class to represent a linked data structure.
- Define data structures to represent compound and complex data.
- Adhere method implementations to a specific runtime standard.
- Produce clear and effective documentation to improve comprehension and maintainability of classes.
- Write classes that are readable and maintainable, and that conform to the provided guidelines for style and implementation.

System Structure

In our system, as in Git, a set of documents and their histories is referred to as a *repository.* Each revision within a repository is referred to as a *commit.* The most recent commit within a repository is referred to as the *head* of the repository. 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 the following provided class. **You must not modify this class in any way.**

▼ Expand

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

    public Commit(String message, Commit past) {
        this.id = "" + currentCommitID++;
        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;
    }

    public static void resetIds() {
        Commit.currentCommitID = 0;
    }
}
```

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` and `Date` classes), 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. 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 `ListNode` 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.)

NOTE: Even though the inner class is `public`, a client is unaware of how we choose to represent the concept of a commit! Make sure to account for this when describing the functionality of the `Repository` class and its methods.

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.
- Throws an `IllegalArgumentException` if `targetId` is `null`
- Note that all elements are unique. Therefore, it should not continue looping unnecessarily once the element of interest is found.

```
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 with no trailing newline character at the end.
    - 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, and add it to this repository.
    - The new commit should become the new head of this repository, preserving the history behind it.
- Throws an `IllegalArgumentException` if `message` is `null`
- 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.
- Throws an `IllegalArgumentException` if `targetId` is `null`
- Returns `true` if the commit was successfully dropped, and `false` if there is no commit that matches the given ID in the repository.
- Note that all elements are unique. Therefore, it should not continue looping unnecessarily once the element of interest is found.

```
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 null, throw an `IllegalArgumentException`
    - 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.

Run Time Requirement

- Unless otherwise stated, all methods must run in `O(n)` time where `n` is the size of the repository.
    - In the case of `synchronize`, the method should run in `O(n+m)` time where `n` is the size of this repository and `m` is the size of the other repository. In other words, the method should run in linear time.
- `getRepoHead`, `toString`, and `commit` must run in `O(1)` time.

`synchronize` Explained

▼ Expand

> **NOTE:** 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)
5 at 2023-07-02 at 18:28:39 PDT: Add initial scaffold
4 at 2023-07-02 at 16:25:43 PDT: Edit README
3 at 2023-06-30 at 03:45:12 PDT: Upload README

Repository #2 (repo2)
2 at 2023-07-01 at 12:04:28 PDT: Edit documentation
1 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)
5 at 2023-07-02 at 18:28:39 PDT: Add initial scaffold
4 at 2023-07-02 at 16:25:43 PDT: Edit README
2 at 2023-07-01 at 12:04:28 PDT: Edit documentation
3 at 2023-06-30 at 03:45:12 PDT: Upload README
1 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]
```

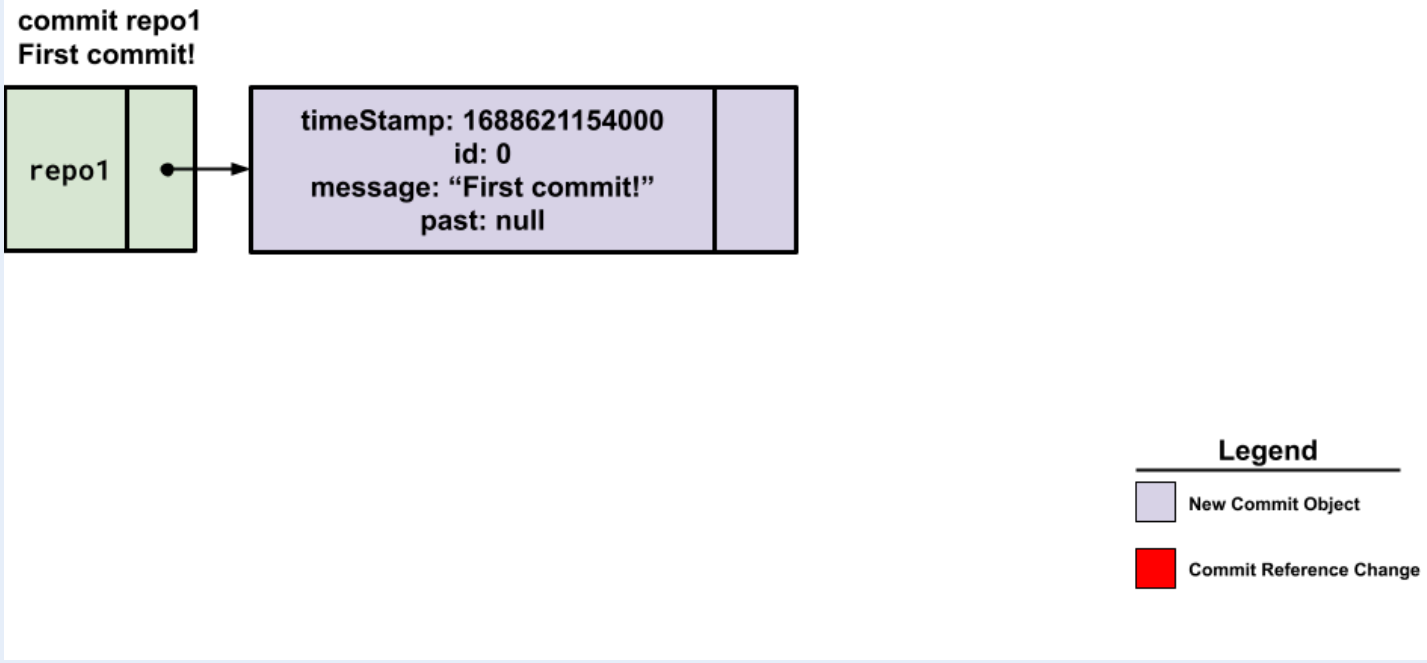
```
Enter operation and repository: commit repo2
Enter commit message: Commit the third
  New commit: 2

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 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 the images below, which visualize the repositories changing throughout the operations.

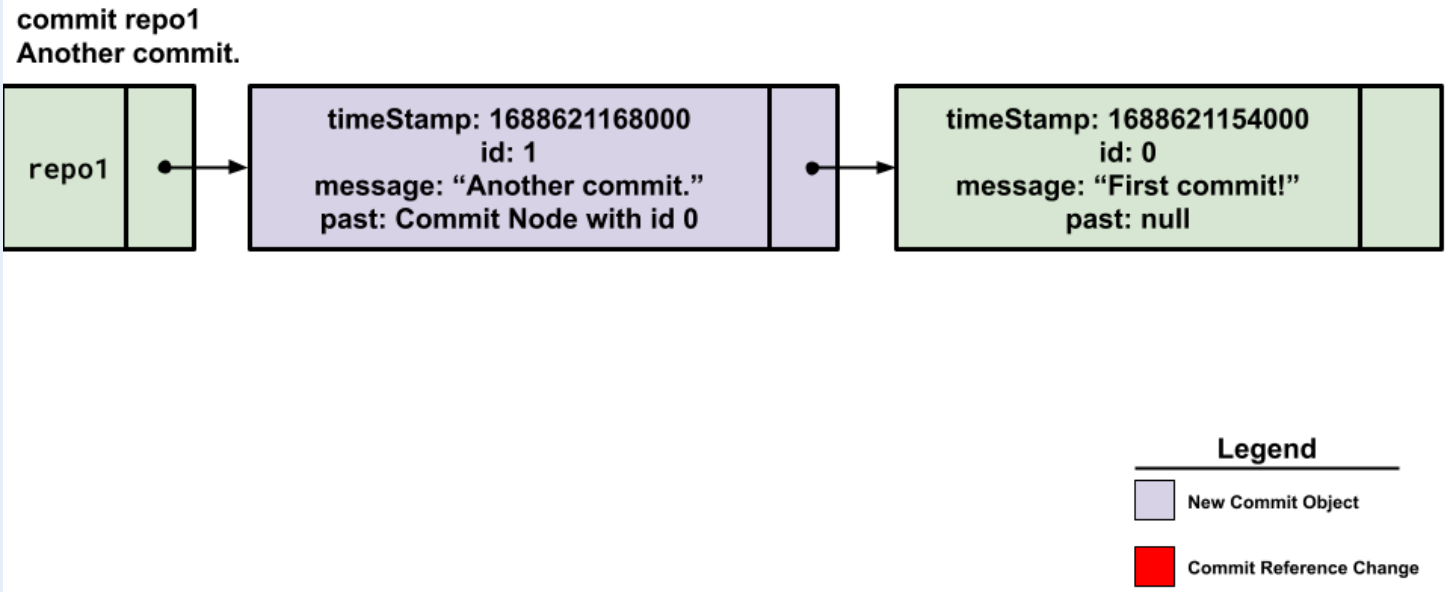
▼ Expand

- **After the user inputs `create repo1`**
  - `repo1` exists and is empty
  - `repo1.head = null`

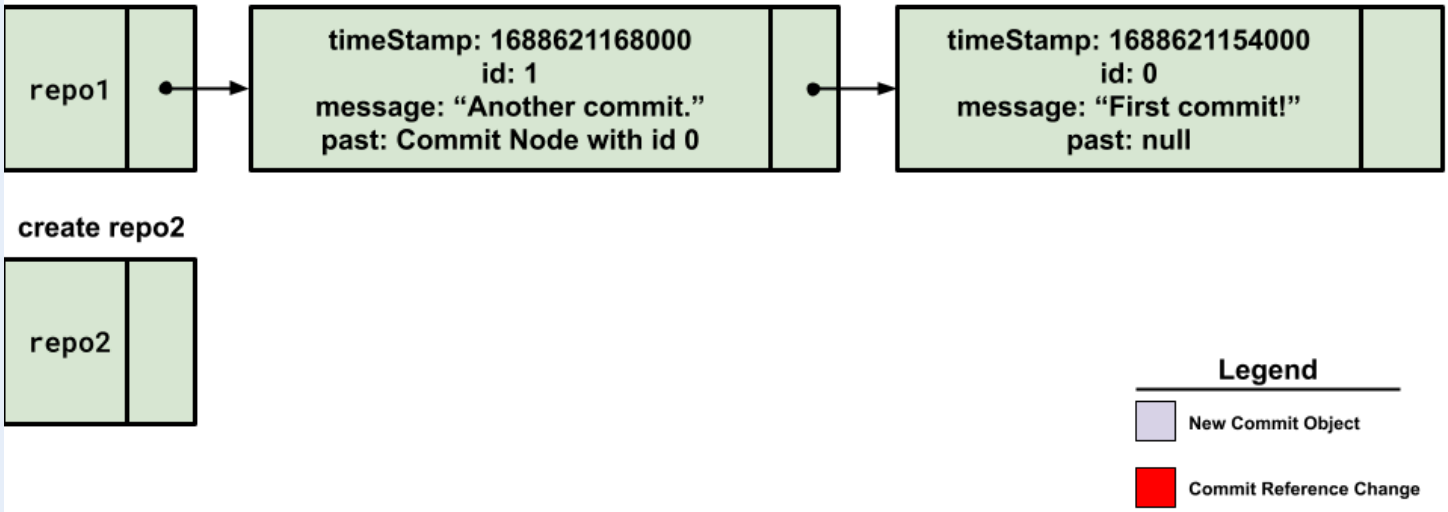


- **After the user inputs `commit repo1` with message `"First commit!"`**
  - A new `Commit` object with id `0` is created
  - `repo1.head` now refers to commit `0`
  - `commit 0.past = null`

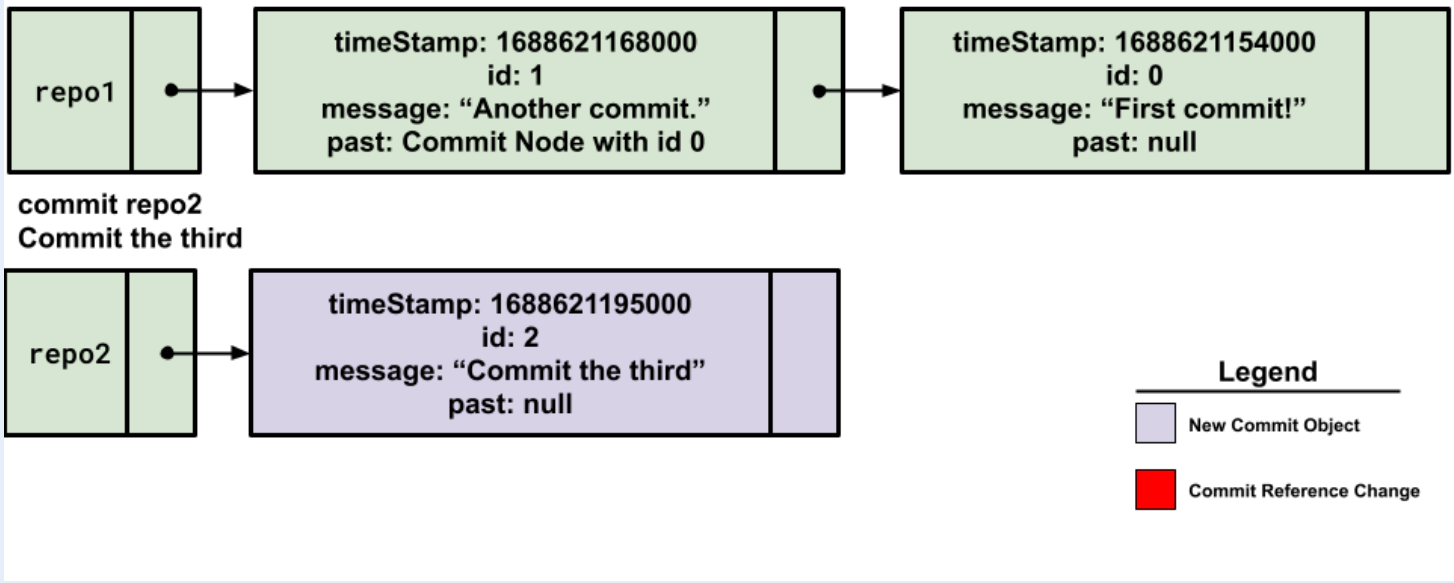
- State:
  - `repo1: head → 0 → null`



- **After the user inputs `commit repo1` with message `"Another commit."`**
  - A new `Commit` object with id `1` is created
  - `repo1.head` now refers to commit `1`
  - `commit 1.past = commit 0`
  - State:
    - `repo1: head → 1 → 0 → null`

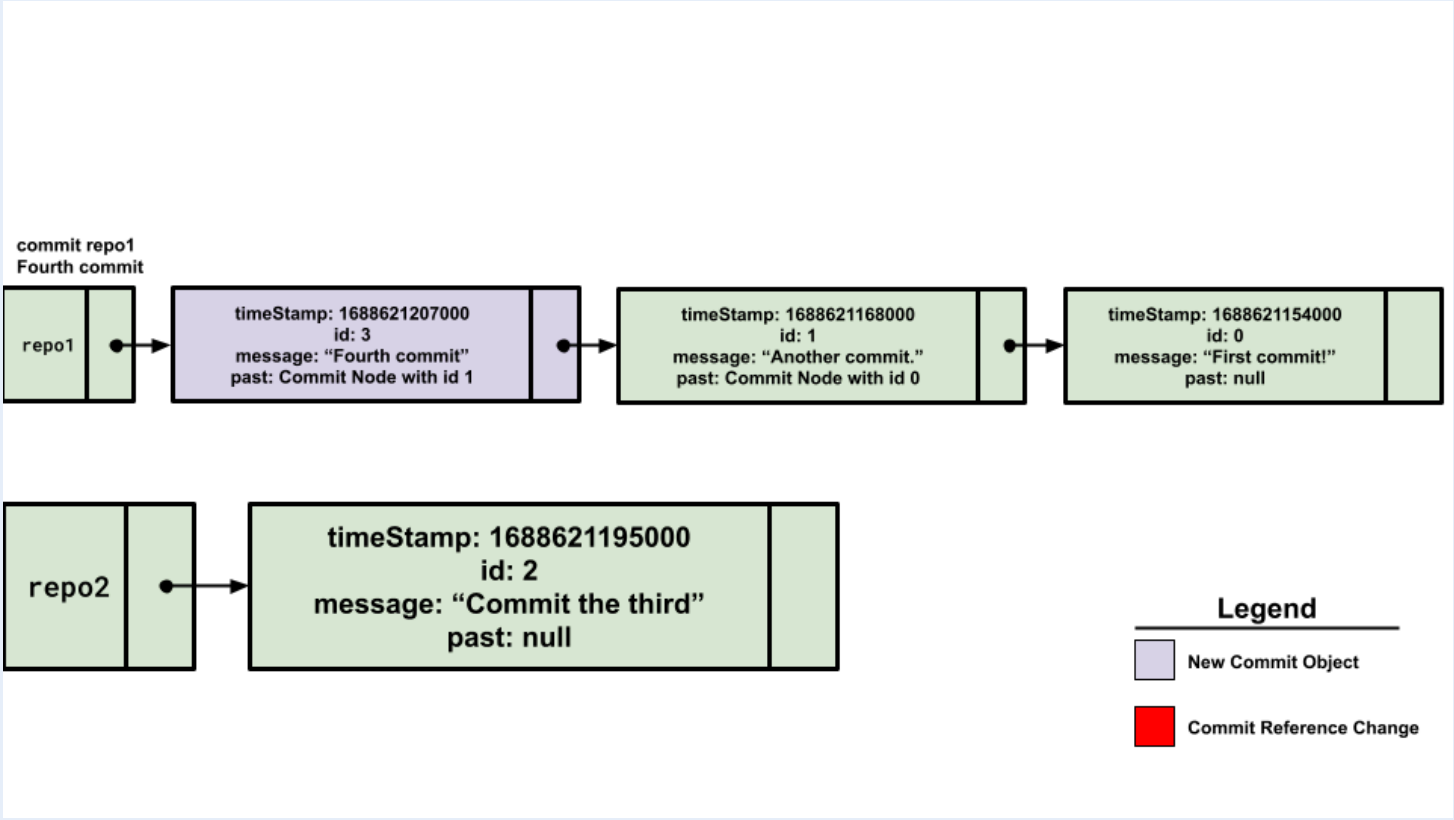


- **After the user inputs `create repo2`**
  - `repo2` exists and is empty
  - State:
    - `repo1: head → 1 → 0 → null`
    - `repo2: head → null`

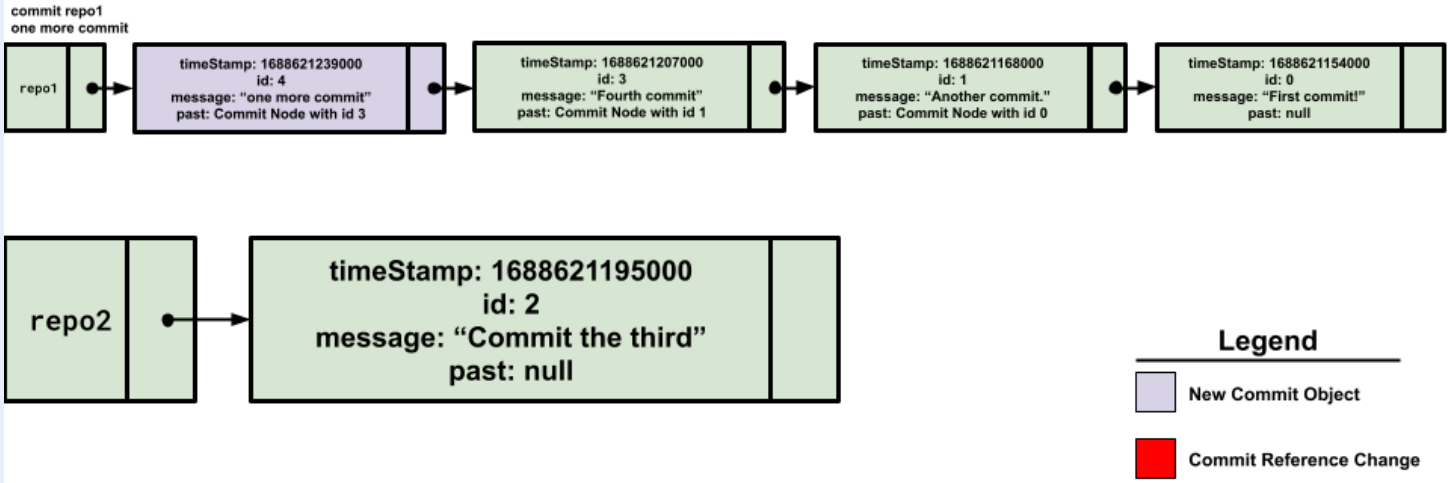


- **After the user inputs `commit repo2` with message `"Commit the third"`**
  - A new `Commit` object with id `2` is created

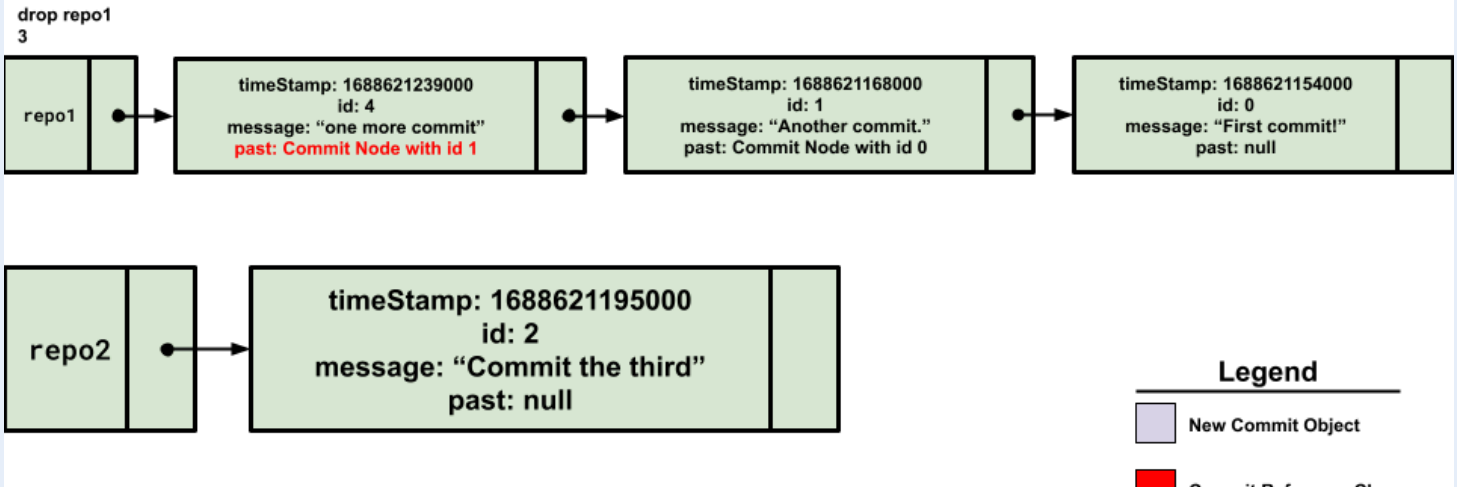
- `repo2.head` now refers to commit `2`
- `commit 2.past = null`
- State:
  - `repo1: head → 1 → 0 → null`
  - `repo2: head → 2 → null`



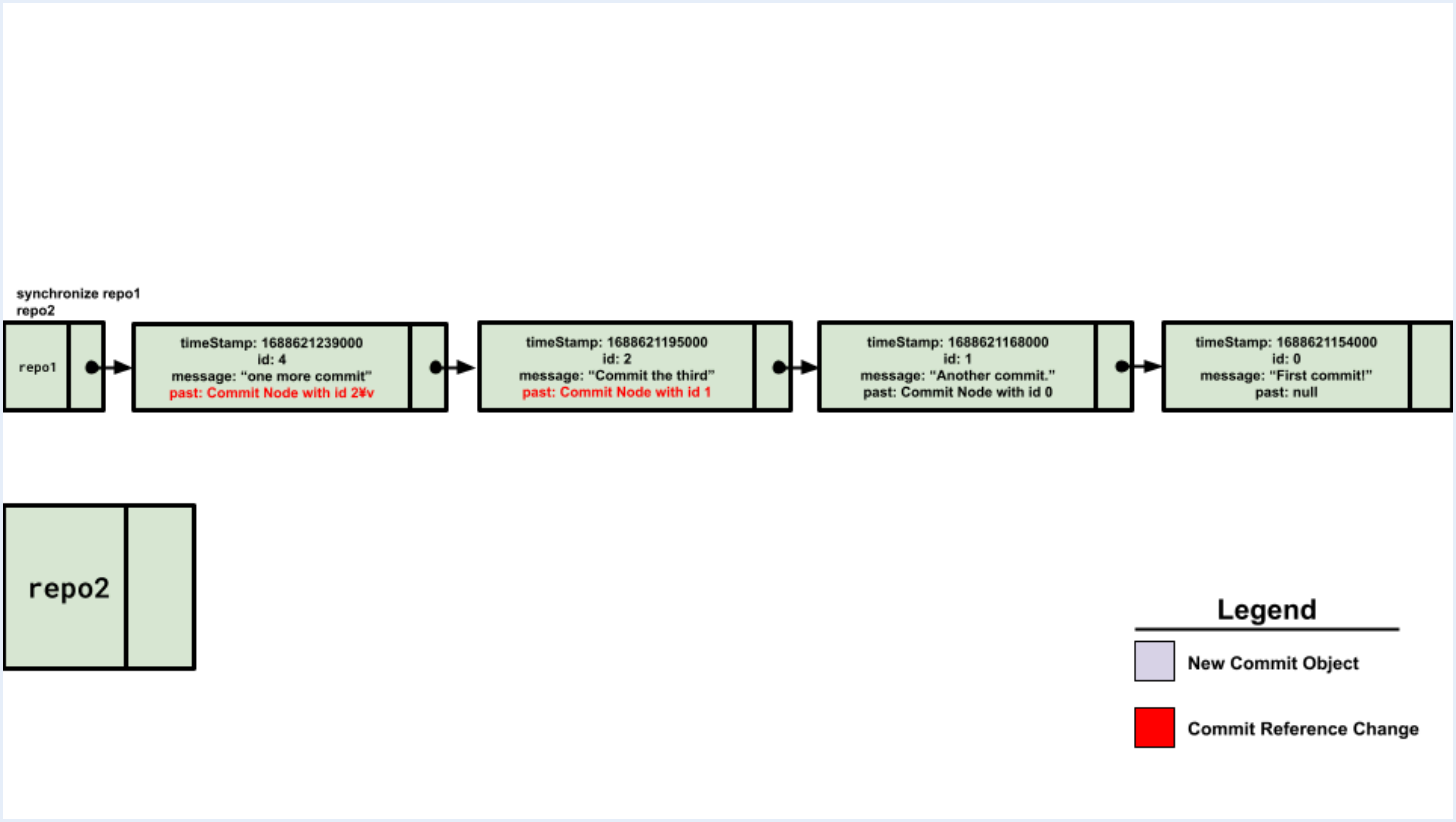
- **After the user inputs `commit repo1` with message `"Fourth commit"`**
  - A new `Commit` object with id `3` is created
  - `repo1.head` now refers to commit `3`
  - `commit 3.past = commit 1`
  - State:
    - `repo1: head → 3 → 1 → 0 → null`
    - `repo2: head → 2 → null`



- **After the user inputs `commit repo1` with message `"one more commit"`**
  - A new `Commit` object with id `4` is created
  - `repo1.head` now refers to commit `4`
  - `commit 4.past = commit 3`
  - State:
    - `repo1: head → 4 → 3 → 1 → 0 → null`
    - `repo2: head → 2 → null`



- **After the user inputs `drop repo1 3`**
  - No new `Commit` object is created
  - Commit `3` is removed from `repo1`
  - `commit 4.past` changes from commit `3` to commit `1`
  - State:
    - `repo1: head → 4 → 1 → 0 → null`
    - `repo2: head → 2 → null`



- **After `repo1.synchronize(repo2)`**
  - No new `Commit` object is created
  - All commits from `repo2` are moved into `repo1` in timestamp order
  - `commit 4.past` remains commit `2` only if `2` is the next most recent commit
  - `commit 2.past = commit 1`
  - `repo2.head = null`
  - Final state:
    - `repo1: head → 4 → 2 → 1 → 0 → null`
    - `repo2: head → null`

Testing

On this assignment, you are required to write **4 of your own test cases for synchronize** covering the different LinkedList cases we've introduced in Section 6 and 7: Front, Middle, Empty, and End. **Each of these test cases should be contained within its own method in your Testing class.**

To help facilitate this, we've provided you with two additional helper methods to help in this process, as well as `ExampleTesting.java` to give you an idea of how to use them. 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**.**

Each of the tests you write must include a `throws InterruptedException` in the method declaration (similar to how we write `throws FileNotFoundException` when doing File I/O) as we must temporarily interrupt execution via `Thread.sleep` to ensure individual commits have unique timestamps. Examples of this can be seen in `ExampleTesting.java`.

> ⚠ **WARNING:** If you choose to not use the provided helper methods, you must make sure to call `Thread.sleep(1)` between each of your commits. This will ensure each individual Commit node has a unique timestamp

You are not required to write test cases for any of the other instance methods for your `Repository` implementation, but we'd encourage you to do so to get more practice writing JUnit tests.

> ⚠ **WARNING:** We have provided a test case that simply tests to see if you've uploaded `Testing.java` and it fails no tests. It does not check whether or not you've met the testing requirements for this assignment.

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:

> ⚠ **WARNING: 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.

- 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.
- Aside from creating a new `Commit` in `commit()`, your implementation should not create any additional helper objects or data structures.

- Your `Repository` class should have the following 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.
  - (Optional) A size field to keep track of the size of the repository.
- Avoid redundant conditionals. If you already know that something is true, then you shouldn't bother to check for it. Such conditionals just make your code longer and harder to understand.
- When writing loops, choose loop bounds or loop conditions that help generalize code the best. Avoid hardcoding special cases outside the loop (e.g., handling the first or last iteration separately) when they could be covered naturally by the loop.
- 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 7 will be a helpful starting point for implementing `synchronize`.

> **NOTE:** Section 7 is not yet available when this assignment is released, so check back once that section is available!

- **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 do **not** need to account for situations where commits have the same timestamps.

Spec Walkthrough

⭐ [GRADED] Mini-Git

Download starter code:

P1_MiniGit.zip

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` for inspiration (`Commit[] elementData` and `int size`) 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 did you struggle with most on this assignment?

No response

Question 6

What skills did you learn and/or practice with working 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

Final Submission

Fill out the box below and click "Submit" in the upper-right corner of the window to submit your work.

Question

I attest that the work I am about to submit is my own and was completed according to the course Academic Honesty and Collaboration policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type "yes" as your response.)

No response

[SCAFFOLD] Mini-Git

This code slide does not have a description.