# Programming Assignment 1: Mini-Git

## ⭐ [GRADED] Implementing Data Structures

**WARNING:** This slide *IS* graded

This slide should be completed in Ed, there is no zip file for downloading to complete it in VSCode

### Problem Description

Extend the functionality of the provided `ArrayIntList` class to support a new method called `synchronize()` (foreshadowing :0). The method should have the following behavior:

```
public void synchronize(ArrayIntList other)
```

- Takes all the elements in the `other` list and moves them into `this` list, combining them such that the final list should still be sorted in ascending. For example, if `list1 = [1, 4, 5]` and `list2 = [2, 3, 6]`, calling `list1.synchronize(list2)` will result in `list1 = [1, 2, 3, 4, 5, 6]` and `list2 = []`.
  - If the `other` list is null, throw an `IllegalArgumentException`
  - If the `other` list is empty, `this` repository should remain unchanged.
  - If `this` list is empty, all elements in the `other` list should be moved into `this` list.
  - At the end of this method's execution, `other` should be an empty list in all cases.
  - You should not construct any new `ArrayIntList` objects to implement this method. You may, however, create a single new array object if the current array size is not big enough.
- You may assume that `this` and `other` will both be sorted in ascending order before calling `synchronize`.

Hint: there's a very similar problem that we ask you to write in the Mini-Git slides. It might be of interest to you to see which method is more complex!

# ⭐ [GRADED] JUnit Testing

**WARNING:** This slide *IS* graded

This slide should be completed in Ed, there is no zip file for downloading to complete it in VSCode

Cornbear has entered the financial sector!



Cornbear's bank is designing and implementing a new `BankAccount` class.

On the right, you can find Cornbear's complete `BankAccount` implementation. Cornbear also tasked his 12 interns to implement the class, and while he is confident his implementation is correct, he's not sure about his interns'. He would like you to identify what is wrong with their code. Since Cornbear makes all his interns sign NDAs (Non-Disclosure Agreements), **you won't be able to access his interns' code—only Cornbear's**.

Unfortunately, each of Cornbear's interns made a mistake in their implementation! Here's what Cornbear knows about the bugs:

- **1 intern** has a bug in their constructor
- **1 intern** has a bug in their `applyInterest` method
- **1 intern** has a bug in their `deposit` method
- **6 interns** have different bugs in their `transfer` method
- **2 interns** have different bugs in their `withdraw` method
- **1 intern** has a bug in their `validateAmounts` private method

You will write JUnit tests in `BankAccountTesting.java` to check the behavior of the 13 total implementations. Your tests should pass on Cornbear's reference implementation, but catch all the sneaky mistakes his interns made!

**NOTE**: The provided implementation of `BankAccount` is correct, and nothing should be changed in `BankAccount.java!`

One test has been provided already; you may edit it, but you must write at least 5 additional tests. You may have to write more; our advice is to write at least one test per public method in

`BankAccount`, including the constructor.

You may find this JUnit cheatsheet and practice problems helpful.

# ⭐[GRADED] Recursive Tracing

**WARNING:** This slide *IS* graded

This slide should be completed in Ed, there is no zip file for downloading to complete it in VSCode

## Problem Description

`Mysteries.java` contains two recursive methods, `mystery1` and `mystery2`.

Check out `MysteriesTesting.java`. We have provided two test methods with incomplete assertions. Your task is to replace each `TODO` with a value you find by tracing through the corresponding mystery method call. You may not run the code to determine the result. Instead, put your recursive tracing skills to use! Once you have replaced each `TODO`, you can click the "Test" button to check your work.

You may find Section 7: Recursive Tracing helpful as you work through this slide.

## 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

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.

### `synchronize` Explained

▶ Expand

## 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

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

# 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`.

ℹ **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**

Compare your experience implementing `synchronize` with an array list vs. a linked list. Which aspects were easier? Which aspects were 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*