Project Title: Terminal-Based Java Banking System
Your Name: Gregory Mitchell
Date: 07/18/25

What the App Does & How It's Useful

This project is a simple yet functional terminal-based banking system built in Java. It allows users to create accounts, deposit and withdraw funds, transfer money between accounts, calculate compound interest recursively, and view transaction history. The app is useful for understanding how financial systems work and for practicing object-oriented programming, recursion, and data structures in real-world scenarios.

Algorithms

1. Transfer Funds Between Accounts

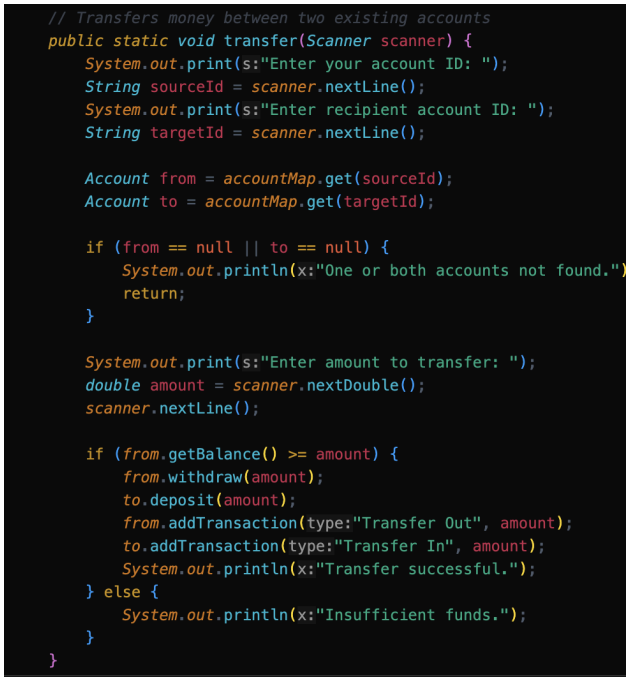
Purpose: Moves money from one account to another after validating account existence and sufficient balance.

How it's used: Option 4 in the menu asks for source and target account IDs, checks their existence, verifies balance, and then processes the transfer.

How I created it: I used a HashMap<String, Account> to access accounts in constant time and called withdraw() and deposit() on the respective accounts. I added a check to avoid null pointer crashes (if (from == null || to == null)). I also logged the transaction to each account.

Code Location: BankSystem.java → transfer()

Big-O Time: O(1) for account lookup, **O(1)** for balance checking, and **O(1)** for transaction logging.



```java
// Transfers money between two existing accounts
public static void transfer(Scanner scanner) {
    System.out.print(s:"Enter your account ID: ");
    String sourceId = scanner.nextLine();
    System.out.print(s:"Enter recipient account ID: ");
    String targetId = scanner.nextLine();

    Account from = accountMap.get(sourceId);
    Account to = accountMap.get(targetId);

    if (from == null || to == null) {
        System.out.println(x:"One or both accounts not found.")
        return;
    }

    System.out.print(s:"Enter amount to transfer: ");
    double amount = scanner.nextDouble();
    scanner.nextLine();

    if (from.getBalance() >= amount) {
        from.withdraw(amount);
        to.deposit(amount);
        from.addTransaction(type:"Transfer Out", amount);
        to.addTransaction(type:"Transfer In", amount);
        System.out.println(x:"Transfer successful.");
    } else {
        System.out.println(x:"Insufficient funds.");
    }
}
```

2. Recursive Compound Interest Calculator

Purpose: Recursively computes how much money a balance will grow over time using the compound interest formula.
How it's used: Option 5 asks for the rate and number of years, then recursively multiplies the balance until the base case is reached.
How I created it: Instead of using Math.pow() or a loop, I wrote a recursive function, calculateCompoundInterest(), that multiplies the principal by (1 + rate) and decrements years until it reaches 0. This satisfied the recursion requirement.
Code Location: BankSystem.java → calculateCompoundInterest()
Big-O Time: O(n) where n is the number of years. Each recursive call goes one level deeper until n == 0.

```java
public static void interestCalculator(Scanner scanner) {
    System.out.print(s:"Enter account ID: ");
    String id = scanner.nextLine();
    Account acc = accountMap.get(id);

    if (acc != null) {
        System.out.print(s:"Enter annual rate (e.g., 0.05 for 5%): ");
        double rate = scanner.nextDouble();
        System.out.print(s:"Enter number of years: ");
        int years = scanner.nextInt();
        scanner.nextLine();

        double result = calculateCompoundInterest(acc.getBalance(), rate, years);
        System.out.printf(format:"Future Value: $%.2f%n", result);
    } else {
        System.out.println(x:"Account not found.");
    }
}

// Recursive method to calculate compound interest
public static double calculateCompoundInterest(double principal, double rate, int years) {
    if (years == 0) return principal;
    return calculateCompoundInterest(principal * (1 + rate), rate, years - 1);
}

// Displays all transactions for a given account
```

3. Transaction History Search & Display

Purpose: Displays all transactions linked to an account.
How it's used: Option 6 allows the user to input their account ID and prints the full history.
How I created it: I used a LinkedList<Transaction> to store transactions inside each Account. The method printTransactionHistory() simply loops through the list and prints each one using toString(). It handles an empty list gracefully.
Code Location: Account.java → printTransactionHistory()
Big-O Time: O(n) where n is the number of transactions.

```java
// Displays all transactions for a given account
public static void viewHistory(Scanner scanner) {
    System.out.print(s:"Enter account ID: ");
    String id = scanner.nextLine();
    Account acc = accountMap.get(id);

    if (acc != null) {
        System.out.println(x:"Transaction History:");
        acc.printTransactionHistory();
    } else {
        System.out.println(x:"Account not found.");
    }
}
```

Algorithm Recap

- The transfer algorithm checks the balance and uses both the HashMap and object methods from Account to ensure funds move securely between users.
- The compound interest function was implemented recursively instead of using Math.pow() to fulfill the recursion SLO.
- The transaction history is printed by iterating over a LinkedList<Transaction>, showcasing proper dynamic storage and loop-based output.

Data Structures Used (3 Total)

Structure	Where it's used	Why was it chosen
ArrayList<Account>	Global list of all user accounts	Easy to store all users in one place and iterate if needed. It also has an O(1) append time, which is good.
HashMap<String, Account>	Maps account IDs to account objects	Critical for performance: O(1) lookup time makes

		access to user accounts fast for actions like deposit, withdraw, and transfer. Much better than looping through an array.
LinkedList<Transaction>	Stores transaction history in each account	LinkedList makes it efficient to keep adding new transactions with no array resizing. Since we only ever iterate or append, it's ideal for this use case.

Opportunity During Development

While building the interest calculator, I realized I could use recursion. Instead of a formula, I implemented a recursive method that multiplies the principal by the rate and reduces the year counter until it reaches 0. This was a creative opportunity to apply recursion meaningfully.

Error Encountered & Fixed

Originally, my transfer() function did not check if either account was null, which caused a NullPointerException. I fixed this by adding a check using:
if (from == null || to == null) { ... } This stopped the crash and provided user feedback instead.

How I Used ChatGPT

I used ChatGPT as a support tool to refine my logic and translate pseudocode into efficient and readable Java. I didn’t copy full methods or rely on auto-generated code; instead, I asked targeted questions to better understand what design patterns I should use, such as how to structure the main menu loop, how to apply recursion for compound interest, and how to organize the Account and Transaction classes effectively. I wrote all the algorithms and functionality myself and used ChatGPT at the end to review my implementations for clarity, correctness, and best practices.

What I’d Improve or Add Next Version

If I had more time, I would: Add user authentication (username + PIN), save/load accounts from a file (using serialization or file I/O), build a GUI with JavaFX or Swing, and add interest calculation that auto-applies monthly/yearly.