Bank Account Management

Project Overview

Complexity: Medium

Time Estimate: 6–8 hours

Technology Stack: Java 21 (LTS), IntelliJ IDEA Community Edition, JUnit 5, Git

Build a console-based **Bank Account Management System** that extends the Week 1 foundation.

Users can create accounts, perform transactions (deposit, withdrawal, transfer), handle invalid inputs gracefully, and record all operations in memory using arrays.

This week focuses on:

- **Clean Code Practices** (refactoring, meaningful names, modular methods)
- **Exception Handling** (using `try-catch`, `throws`, and custom exceptions)
- **Testing Fundamentals** (applying JUnit for unit and integration tests)
- **Version Control with Git** (creating branches, committing changes, and using `git cherry-pick` for code reuse).

Future labs will enhance this version with Java Collections and file-based persistence.

Learning Objectives

By completing this lab, you will be able to:

- Apply **clean code principles** for readability, maintainability, and scalability.
- Implement **robust exception handling** to manage invalid inputs and transaction errors.
- Write and execute **unit tests with JUnit 5** for critical methods like `deposit()`, `withdraw()`, and `transfer()`.
- Utilize **Git** for version control — initializing repositories, committing, branching, merging, and cherry-picking specific commits.
- Refactor Week 1 classes (`Account`, `TransactionManager`, etc.) to improve structure and reduce redundancy.
- Demonstrate **code review and testing cycles** to build confidence in software changes.
- Prepare the codebase for Week 3 enhancements (Collections API and File Storage).

System Features Overview

Your Bank Account Management System now includes five enhanced features:

Feature 1: Refactored Account and Transaction Classes

- Simplify methods with clear names and comments.
- Apply modular design to separate responsibilities (balance calculation vs. display).
- Introduce helper methods for common operations (e.g., `validateAmount()`).

Feature 2: Error Handling and Validation

- Handle invalid inputs with try-catch blocks.
- Throw custom exceptions (e.g., `InsufficientFundsException`, `InvalidAccountException`).
- Ensure withdrawals don't exceed overdraft limits or go below minimum balance.

Feature 3: Transaction Testing and Verification

- Write JUnit tests for `deposit()`, `withdraw()`, and `transfer()`.
- Validate balance updates, exception conditions, and transaction records.
- Log test results to console for clarity.

Feature 4: Git Version Control Integration

- Initialize a Git repository and track code changes.
- Use branches for features (e.g., `feature/error-handling`, `feature/testing`).
- Merge and cherry-pick commits across branches for controlled integration.

Feature 5: Enhanced Console User Experience

- Display error messages clearly.
- Add confirmation prompts for transactions.
- Simulate test outputs in the console (UI and JUnit summary).

Console UI Examples

1. Main Menu (Refactored)

```xml
╔═══════════════════════════════════════════╗
║      BANK ACCOUNT MANAGEMENT SYSTEM       ║
╚═══════════════════════════════════════════╝

Main Menu:
-----------
1. Manage Accounts
2. Perform Transactions
3. Generate Account Statements
4. Run Tests
5. Exit

Enter your choice: _

```

2. Error Handling Example – Invalid Deposit

```xml
Enter Account Number: ACC999
❌ Error: Account not found. Please check the account number and try again.

Enter Account Number: ACC001
Enter amount to deposit: -200
❌ Error: Invalid amount. Amount must be greater than 0.

```

3. Transaction Failure Example – Insufficient Funds

```xml
PROCESS TRANSACTION
──────────────────────────────────────
Enter Account Number: ACC002
Select type: 2 (Withdrawal)
Enter amount: $10,000
❌ Transaction Failed: Insufficient funds. Current balance: $2,950.00

```

4. JUnit Test Output Example

```xml
Running tests with JUnit...

Test: depositUpdatesBalance() .......... PASSED
Test: withdrawBelowMinimumThrowsException() .......... PASSED
Test: overdraftWithinLimitAllowed() .......... PASSED
Test: overdraftExceedThrowsException() .......... PASSED

✓ All 4 tests passed successfully!

```

5. Git Workflow Example

```xml
> git init
> git add .
> git commit -m "Initial refactoring for clean code"
> git branch feature/testing
> git checkout feature/testing
> git commit -m "Add JUnit tests for transactions"
> git checkout main
> git cherry-pick <commit-hash-of-tests>
> git push origin main
✓ Cherry-picked JUnit test changes successfully!

```

6. Statement Generation Example (with Error Handling)

```xml
GENERATE ACCOUNT STATEMENT
──────────────────────────────────────
Enter Account Number: ACC001

Account: John Smith (Savings)
Current Balance: $6,750.00

Transactions:
──────────────────────────────────────────────────
TXN001 | DEPOSIT    | +$1,500.00 | $6,750.00
TXN002 | WITHDRAWAL | -$750.00   | $5,250.00
──────────────────────────────────────────────────
Net Change: +$750.00

✓ Statement generated successfully.

```

7. Application Exit

```xml
Thank you for using the Bank Account Management System!
All data saved in memory. Remember to commit your latest changes to Git!
Goodbye!

```

Expected User Workflows

Workflow 1: Handle Transaction Error

1. User selects "Perform Transactions."
2. Enters invalid account number → error message displayed.
3. Re-enters valid account number.
4. Attempts withdrawal beyond balance → custom exception shown.
5. Performs valid withdrawal → success confirmed.

Workflow 2: Run JUnit Tests

1. User selects "Run Tests."
2. System executes unit tests on core methods.
3. Results display as Passed/Failed.
4. User reviews Git commit to store test results.

Workflow 3: Version Control Integration

1. Developer creates `feature/error-handling` branch.
2. Implements exceptions and tests.
3. Commits and pushes to branch.
4. Merges into main using `git merge` .
5. Uses `git cherry-pick` to bring selected fix commits into testing branch.

Workflow 4: Statement Generation with Refactored Code

1. User selects "Generate Statement."
2. System fetches transactions from array.
3. Applies error handling for empty records.
4. Generates summary with totals and balances.

User Stories

Epic 1: Error Handling and Validation

US-1.1: Handle Invalid Deposits

- **As a user**, I want to see clear errors for negative deposit amounts so that I don't crash the program.
- **Acceptance Criteria:** Negative amounts throw `InvalidAmountException`.

US-1.2: Prevent Overdraft Exceeding Limit

- **Acceptance Criteria:** Withdrawals beyond limit trigger `OverdraftExceededException`.

Technical Requirements:

- Use `try-catch` for input validation.
- Define custom exception classes for specific errors.

Epic 2: Code Refactoring and Clean Design

US-2.1: Refactor TransactionManager for Readability

- Break long methods into smaller modular ones (e.g., `validateTransaction`, `applyTransaction`).
- Rename variables for clarity.

US-2.2: Apply Comments and Formatting Standards

- Follow Google Java Style Guide for naming and indentation.

Technical Requirements:

- Run manual review to ensure methods ≤ 25 lines.
- Add JavaDoc comments to each public method.

Epic 3: Testing and Verification

US-3.1: Write Unit Tests for Deposit and Withdraw**

- **Acceptance Criteria:** Tests pass for valid and invalid cases.

US-3.2: Test Transfer Between Accounts**

- Check balance updates in both accounts.

Technical Requirements:

- Use JUnit 5.
- Organize tests under `src/test/java`.
- Apply `@BeforeEach` to reset test data.

Epic 4: Git Version Control Workflows

US-4.1: Implement Feature Branching**

- Create and switch branches using `git branch` and `git checkout`.

US-4.2: Cherry-Pick Specific Commits**

- Selectively apply tested commits across branches.

Technical Requirements:

- Include Git commands in README.
- Perform at least 3 commits during lab progression.

Epic 5: Statement Generation Enhancement

US-5.1: Generate Error-Free Statements**

- Handle accounts with no transactions gracefully.
- Format output for clarity and totals.

Technical Requirements:

- Sort transactions by timestamp (newest first).
- Ensure balance summaries use 2-decimal precision.

Project Structure

XML

```
bank-account-management-system/
│
├── src/
│   ├── Main.java
│   ├── models/
│   │   ├── Account.java
│   │   ├── SavingsAccount.java
│   │   ├── CheckingAccount.java
│   │   ├── Customer.java
│   │   ├── RegularCustomer.java
│   │   ├── PremiumCustomer.java
│   │   ├── Transaction.java
│   │   ├── exceptions/
│   │   │   ├── InvalidAmountException.java
│   │   │   ├── InsufficientFundsException.java
│   │   │   └── OverdraftExceededException.java
│   │
│   ├── services/
│   │   ├── AccountManager.java
│   │   ├── TransactionManager.java
│   │   └── StatementGenerator.java
│   │
│   └── utils/
│       └── ValidationUtils.java
│
├── src/test/java/
│   ├── AccountTest.java
│   ├── TransactionManagerTest.java
│   └── ExceptionTest.java
│
├── docs/
│   └── git-workflow.md
│
└── README.md

```

Implementation Phases

Phase 1: Setup and Refactoring (1–2 hours)

Tasks:

- Fork Week 1 repo and create a new repo for week 2 then create a new branch `feature/refactor`.
- Refactor Account and TransactionManager for clarity.
- Add JavaDocs and consistent naming.

Git Commands:

```xml
git checkout -b feature/refactor
git add .
git commit -m "Refactored AccountManager and TransactionManager"

```

Phase 2: Exception Handling

Tasks:

- Create custom exceptions.
- Wrap input validation in try-catch blocks.
- Update UI to display errors gracefully.

Git Commands:

```xml
git checkout -b feature/exceptions
git commit -m "Commit Message"

```

Phase 3: Testing and Verification (2 hours)

Tasks:

- Add JUnit 5 to project.
- Write unit tests for deposit, withdraw, transfer.
- Run and document results.

Git Commands:

```xml
git checkout -b feature/testing
git add file to-be-added
git commit -m "Commit Message"
git cherry-pick <refactor-commit-hash>
```

Phase 4: Merge and Documentation

Tasks:

- Merge branches and resolve conflicts.
- Document Git workflow in README.
- Submit final repository.

Minimum Requirements Checklist

- ◯ All custom exceptions implemented.
- ◯ JUnit tests created and passing.
- ◯ Code refactored for clean structure.
- ◯ Git repository initialized with branching and cherry-pick usage.
- ◯ README includes Git workflow and test results.
- ◯ All Week 1 features are still functional.

Grading Rubric

	Criteria	Points	Excellent (90–100%)	Good (75–89%)	Satisfactory (60–74%)
1	Clean Code & Refactoring	20	Consistent naming, JavaDocs, DRY methods	Mostly refactored, minor redundancy	Basic clean-up applied
2	Exception Handling	20	Custom exceptions, robust try-catch	Handled core cases	Limited validation
3	Testing & Verification (JUnit)	20	Comprehensive unit tests, edge cases covered	Core tests only	Minimal testing
4	Git Version Control	15	Branches, merges, and cherry-picks used effectively	Basic branch workflow	Single branch
5	Functionality & Stability	15	All Week 1 features enhanced and stable	Minor bugs	Partially functional

6	DSA (Use of Arrays & Algorithms)	10	Efficient array management and search/sort logic	Functional but not optimized	Inefficient iteration
7	Documentation	10	Clear README + Git docs included	Partial docs	Limited notes
8	**Total**	**100**			

Submission Requirements

Deliverables:

- Public GitHub repository with:
  - Source code (`/src`)
  - JUnit tests (`/src/test/java`)
  - Git workflow documentation (`/docs/git-workflow.md`)
  - README with setup, testing, and branching instructions.
- At least 5 Git commits showing progress (refactor, exceptions, testing, merge).

Submission Link:

(Insert Google Form or LMS link here)

Testing the Application

Test Scenario 1: Refactored Account Creation

- Run the refactored application
- Select option 1 (Create Account)
- Enter valid details for a Savings Account (Regular Customer)
- Verify constructors and encapsulation work correctly (no direct field access)
- Confirm auto-generated Account ID (e.g., ACC001)
- Check console output for clean, formatted messages and no redundant lines

Test Scenario 2: Deposit Operation with Exception Handling

- Select option 2 (Perform Transactions)
- Enter invalid account number (e.g., ACC999)
- Verify custom exception (`InvalidAccountException`) displays a user-friendly message
- Enter valid account number and positive deposit amount
- Confirm balance updates correctly and transaction logs are recorded

Test Scenario 3: Withdrawal and Overdraft Validation

- Select option 2 (Perform Transactions)
- Choose Withdrawal on a Checking Account
- Enter an amount exceeding balance but within overdraft limit → should succeed
- Enter amount beyond overdraft limit → should throw `OverdraftExceededException`
- Confirm final balance accuracy and transaction count increment

Test Scenario 4: Statement Generation After Refactoring

- Select option 3 (Generate Account Statement)
- Enter valid account number
- Verify clean, formatted output using refactored methods
- Confirm transactions display in reverse chronological order
- Ensure summary totals (deposits, withdrawals, net change) are correct

Test Scenario 5: Exception Handling for Invalid Inputs

- At main menu, enter invalid choice (e.g., letters instead of numbers)
- Attempt deposit with negative amount or zero
- Verify custom exceptions (`InvalidAmountException` , `InputMismatchException`) are caught
- Check program continues running without crashing
- Confirm retry prompts appear correctly

Test Scenario 6: Run JUnit Tests

- Open JUnit test classes (`AccountTest` , `TransactionManagerTest` , `ExceptionTest`)
- Run unit tests for deposit, withdraw, and transfer methods
- Verify all assertions pass (expected vs actual balances)
- Confirm exception-based tests (`assertThrows`) behave as intended
- Check JUnit summary shows all tests successful

Test Scenario 7: Git Cherry-Pick, Push, and Code Quality Verification

- Switch to branch `feature/error-handling` and use `git cherry-pick <commit-hash>` to apply selected tested commits.
- Resolve conflicts if any, verify the app compiles and runs correctly.
- Push final code to remote (`git push origin main`) and confirm commits and branches appear on GitHub.
- Check `.gitignore` , `README.md` , and code formatting for consistency.
- Ensure JavaDocs, clean indentation, and meaningful commit messages are present.
