Certainly! Below is the modified problem statement that includes the integration of **JDBC (Java Database Connectivity)** to the Bank Account Management System. This version adds database connectivity for storing and retrieving bank account data, handling transactions, and supporting multithreading.

**Java Banking Assignment: Bank Account Management System with JDBC Integration**

**Problem Statement:**

You are tasked with developing a **Bank Account Management System** that allows multiple users to perform various banking operations like deposit, withdraw, transfer funds, and check balance. The system must use **Object-Oriented Programming (OOP)** principles, **Collections**, **Exceptions**, **Multithreading**, and **JDBC (Java Database Connectivity)** to ensure smooth operation and efficiency in handling multiple users simultaneously, with data stored in a relational database.

**Functional Requirements:**

**1. Bank Account Class:**

* Represent a bank account with attributes such as **account number**, **account holder’s name**, **account balance**, and **account type**.
* Provide methods to:
  + **Deposit** money.
  + **Withdraw** money (ensure sufficient balance).
  + **Transfer** money between accounts.
  + **Check balance**.
* Use **JDBC** to store and retrieve account details from a relational database.

**2. Bank Class:**

* Store and manage a collection of bank accounts, which are retrieved from and saved to the database.
* Allow searching for a bank account by **account number**.
* Ensure that multiple users can interact with their accounts concurrently.
* Use **JDBC** to store account data and interact with the database for operations like account creation, withdrawal, deposit, and balance checking.

**3. Transaction Class:**

* Represent a transaction (deposit, withdraw, or transfer) with details such as **transaction ID**, **transaction type**, **amount**, and **timestamp**.
* Handle the processing of the transaction in the system.
* Record transactions in a **database** for auditing purposes.

**4. Exception Handling:**

* Handle errors such as:
  + **Insufficient funds** when performing withdrawals or transfers.
  + **Invalid account details** when searching or processing transactions.
  + **Invalid transaction types** (other than "Deposit", "Withdraw", or "Transfer").
* Use **custom exceptions** to manage these errors and ensure smooth system operation.

**5. Multithreading:**

* Use **multithreading** to simulate concurrent banking operations, such as multiple customers performing transactions on their accounts at the same time.
* Use **synchronized** methods or other thread safety mechanisms to ensure that balance updates are atomic.
* Use **ExecutorService** or a **thread pool** to manage concurrent processing of transactions.
* Ensure that transactions are processed sequentially (first-come, first-served) while maintaining thread safety for account balance updates.

**6. Collections:**

* Use a collection like **HashMap** or **ArrayList** to manage multiple bank accounts in memory during the runtime.
* Use a **Queue** or **BlockingQueue** to simulate transaction processing, ensuring transactions are handled one at a time while using multithreading.

**7. JDBC Integration:**

* Use **JDBC** to connect to a relational database and manage data persistence. The database will store accounts and transaction history.
* Use **PreparedStatement** to securely execute SQL queries (INSERT, UPDATE, SELECT) and **ResultSet** to fetch data.
* Implement database connectivity using **DriverManager** or **DataSource** for establishing connections to the database.
* Ensure that each account’s balance is updated in the database when deposits or withdrawals occur.

**Detailed Breakdown:**

**1. BankAccount Class:**

* **Attributes:**
  + accountNumber (String)
  + accountHolderName (String)
  + balance (double)
  + accountType (String - "Saving" or "Current")
* **Methods:**
  + deposit(double amount): Adds money to the account and updates the balance in the database.
  + withdraw(double amount): Deducts money from the account and ensures the balance does not go below zero.
  + transfer(BankAccount targetAccount, double amount): Transfers money between accounts and updates both account balances.
  + checkBalance(): Retrieves the current balance from the account.
  + **Database Operations** (via JDBC):
    - Store the account in the database.
    - Retrieve the account’s details from the database.
    - Update account balances in the database after transactions.

**2. Bank Class:**

* **Attributes:**
  + accountsMap (HashMap<String, BankAccount>) - Holds all accounts indexed by their account number (useful for in-memory operations).
* **Methods:**
  + addAccount(BankAccount account): Adds a new account to the database and the in-memory map.
  + getAccount(String accountNumber): Retrieves an account from the database or in-memory map by its account number.
  + processTransaction(Transaction transaction): Processes the transaction by invoking methods of BankAccount to deposit, withdraw, or transfer money.
  + **Database Operations** (via JDBC):
    - Perform account creation and deletion in the database.
    - Use JDBC to fetch and update account details based on transactions.

**3. Transaction Class:**

* **Attributes:**
  + transactionID (String)
  + transactionType (String - "Deposit", "Withdraw", "Transfer")
  + amount (double)
  + accountNumber (String)
  + timestamp (LocalDateTime)
* **Methods:**
  + process(): Executes the transaction (deposit, withdraw, or transfer) by interacting with the Bank class and relevant BankAccount objects.
  + **Database Operations** (via JDBC):
    - Log the transaction in the transaction history table in the database for future reference.

**4. Custom Exceptions:**

* **InsufficientFundsException**: Raised when a withdrawal or transfer exceeds the available balance in an account.
* **InvalidAccountException**: Raised when an invalid account number is provided (does not exist).
* **InvalidTransactionException**: Raised when an invalid transaction type is provided (other than "Deposit", "Withdraw", or "Transfer").
* **SQLException**: Raised for general database-related errors.

**5. Multithreading:**

* **Transaction Processing with Multithreading**:
  + Use ExecutorService or ThreadPoolExecutor to handle concurrent transaction processing.
  + Ensure that multiple threads can process transactions simultaneously without interfering with each other.
  + **Thread Safety**: Use synchronized methods or ReentrantLock to prevent race conditions when updating account balances.

**6. Collections & Queue:**

* Use a **HashMap** to store accounts in memory and allow quick access by account number.
* Use a **Queue** (such as a BlockingQueue) to manage transaction processing and ensure FIFO (First-In, First-Out) order for transactions.

**7. JDBC Database Operations:**

* **Database Schema**: Store bank account data in the accounts table and transaction history in a transactions table.
  + **Accounts Table**: Stores account details like account number, account holder, balance, and account type.
  + **Transactions Table**: Stores transaction details like transaction ID, type, amount, timestamp, and associated account number.