**Problem Statement:**

You are tasked with developing a simple **Banking System** using **Spring Core** to simulate basic banking functionalities such as **account creation**, **balance checking**, and **money transfer** between accounts. The banking system will handle user accounts, perform operations on those accounts, and integrate with a mock **Database** to persist the account data.

**Requirements:**

1. **Account Creation:**
   * The system should allow users to create a bank account by providing the account holder's name, account number, and an initial deposit amount.
   * Each account should have a unique account number.
2. **Balance Inquiry:**
   * Users should be able to check the balance of a given account using the account number.
3. **Money Transfer:**
   * The system should allow money transfers between two accounts.
   * If there are insufficient funds in the source account, an exception should be thrown (InsufficientFundsException).
   * If the source account or destination account does not exist, an appropriate exception should be thrown.
4. **Error Handling:**
   * The system should handle errors gracefully, such as insufficient funds or invalid account numbers.
   * Use custom exceptions where applicable (e.g., InsufficientFundsException, AccountNotFoundException).
5. **Dependency Injection (Spring Core):**
   * Use **Spring's Inversion of Control (IoC)** and **Dependency Injection** (DI) to manage the dependencies between different components of the system (e.g., BankService, AccountDAO, and BankAccount).
   * The **BankService** class should handle business logic like account creation and transfer, while the **AccountDAO** class will interact with the "database" (which could be an in-memory store or a mock database).
6. **Singleton Pattern (Database Connection):**
   * Implement the **Singleton pattern** to ensure that only one instance of the database connection (mocked) is created and shared across the system.
   * The **DatabaseConnection** class should follow the Singleton pattern to manage the connection.
7. **Spring Bean Configuration:**
   * Use **XML-based configuration** (or Java configuration) for Spring beans in the **applicationContext.xml** file.
   * Ensure proper wiring of beans such as BankService, AccountDAO, and BankAccount.
8. **Spring AOP (Optional):**
   * Optionally, implement **Aspect-Oriented Programming (AOP)** to log method calls and transactions for debugging and monitoring purposes.

**System Components:**

1. **BankAccount Class:**
   * Represents a bank account with an account number, holder's name, and balance.
2. **AccountDAO Class:**
   * Responsible for managing accounts in the "database" (a map or mock data structure).
   * Contains methods for adding, fetching, and updating accounts.
3. **BankService Class:**
   * Contains business logic like creating accounts, transferring funds, checking balances, and handling errors (e.g., insufficient funds).
4. **InsufficientFundsException Class:**
   * Custom exception for handling cases where an account does not have enough funds for a transaction.
5. **DatabaseConnection Class (Singleton):**
   * A mock database connection class following the Singleton pattern to ensure only one instance is created.
6. **BankingApp Class:**
   * The main class where the Spring context is loaded, and operations (like creating accounts, transferring money, etc.) are performed.
7. **applicationContext.xml (Spring Configuration):**
   * The XML configuration file that defines the Spring beans, their scopes, and dependencies.

**Implementation Steps:**

1. **Step 1: Set up Spring Core Project**
   * Create a Maven or Gradle project and add dependencies for Spring Core (using Spring Framework).
2. **Step 2: Create Java Classes**
   * Implement BankAccount, AccountDAO, BankService, InsufficientFundsException, and DatabaseConnection.
3. **Step 3: Configure Spring Beans**
   * Create applicationContext.xml for defining Spring beans and wiring dependencies. Make sure beans like BankService, AccountDAO, and DatabaseConnection are defined in the XML configuration.
4. **Step 4: Implement Business Logic in BankService**
   * Implement methods to create accounts, check balances, and transfer funds in BankService.
5. **Step 5: Handle Exceptions**
   * Throw and catch custom exceptions like InsufficientFundsException during money transfers and other operations.
6. **Step 6: Implement Main Class (BankingApp)**
   * Use Spring's ApplicationContext to load the beans and invoke operations like account creation, balance checking, and transfers.
7. **Step 7: Test and Debug**
   * Test your system by simulating various operations (like creating accounts, checking balances, transferring funds) and ensure exceptions are handled correctly.

**Example Use Case:**

1. **Creating a Bank Account:**
   * A user requests to create an account with the name "Alice", account number "1001", and an initial deposit of 1000.
   * The system creates the account and stores it in the database.
2. **Checking Account Balance:**
   * A user requests to check the balance for account number "1001".
   * The system fetches the account details from the database and returns the balance.
3. **Money Transfer:**
   * A user requests to transfer 500 from "1001" to "1002".
   * The system checks if there are enough funds in account "1001".
   * If there are sufficient funds, the transfer is processed and both account balances are updated.
4. **Handling Errors:**
   * If a user tries to transfer more money than is available in the source account, an InsufficientFundsException is thrown.

**Spring Beans Configuration (applicationContext.xml):**

xml

Copy

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- BankAccount Bean -->

<bean id="bankAccount" class="com.banking.BankAccount">

<constructor-arg value="1001"/>

<constructor-arg value="Alice"/>

<constructor-arg value="1000.00"/>

</bean>

<!-- AccountDAO Bean -->

<bean id="accountDAO" class="com.banking.AccountDAO"/>

<!-- BankService Bean -->

<bean id="bankService" class="com.banking.BankService">

<property name="accountDAO" ref="accountDAO"/>

</bean>

<!-- DatabaseConnection Bean (Singleton) -->

<bean id="databaseConnection" class="com.banking.DatabaseConnection" scope="singleton"/>

</beans>

**Conclusion:**

This **Spring Core** Banking System demonstrates how to use Spring's **IoC container** to manage beans and dependencies in a banking application. You will use **dependency injection** to decouple the components like BankAccount, BankService, AccountDAO, and DatabaseConnection. The application should allow for basic banking functionalities like account creation, balance checking, and money transfer, all managed by Spring beans.