**Exercise 1: Control Structures**

**Scenario 1:** The bank wants to apply a discount to loan interest rates for customers above 60 years old.

* + **Question:** Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.

**DECLARE**

**CURSOR** c\_customers **IS**

**SELECT** \* **FROM** Customers;

v\_customer c\_customers%ROWTYPE;

**BEGIN**

**FOR** v\_customer **IN** c\_customers LOOP

UPDATE Loans

**SET** InterestRate = InterestRate - (InterestRate \* 0.01)

**WHERE** CustomerID = v\_customer.CustomerID

**AND** DOB &lt; ADD\_MONTHS(SYSDATE, -60\*12);

**END** LOOP;

**END**;

/

**Scenario 2:** A customer can be promoted to VIP status based on their balance.

* + **Question:** Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.

**DECLARE**

**CURSOR** c\_customers **IS**

**SELECT** \* **FROM** Customers;

v\_customer c\_customers%ROWTYPE;

**BEGIN**

**FOR** v\_customer **IN** c\_customers LOOP

UPDATE Customers

**SET** IsVIP = **CASE** **WHEN** Balance &gt; 10000 **THEN** TRUE **ELSE** FALSE **END**

**WHERE** CustomerID = v\_customer.CustomerID;

**END** LOOP;

**END**;

/

**Scenario 3:** The bank wants to send reminders to customers whose loans are due within the next 30 days.

* + **Question:** Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

**DECLARE**

**CURSOR** c\_loans **IS**

**SELECT** c.Name, l.LoanID, l.LoanAmount, l.InterestRate, l.StartDate, l.EndDate

**FROM** Loans l

**INNER** **JOIN** Customers c **ON** l.CustomerID = c.CustomerID

**WHERE** EndDate **BETWEEN** SYSDATE **AND** ADD\_DAYS(SYSDATE, 30);

v\_loan c\_loans%ROWTYPE;

**BEGIN**

**FOR** v\_loan **IN** c\_loans LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ' || v\_loan.Name || ' has a loan (' || v\_loan.LoanID || ') with an amount of ' || v\_loan.LoanAmount || ' due in the next 30 days.');

**END** LOOP;

**END**;

/

**Exercise 2: Error Handling**

**Scenario 1:** Handle exceptions during fund transfers between accounts.

* + **Question:** Write a stored procedure **SafeTransferFunds** that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.

**CREATE** **OR** REPLACE **PROCEDURE** SafeTransferFunds (

p\_from\_account\_id NUMBER,

p\_to\_account\_id NUMBER,

p\_amount NUMBER

) **IS**

PRAGMA AUTONOMOUS\_TRANSACTION;

**BEGIN**

-- Begin transaction

**SAVEPOINT** transfer\_start;

-- Check for sufficient funds

**DECLARE**

v\_from\_balance Accounts.Balance%TYPE;

**BEGIN**

**SELECT** Balance **INTO** v\_from\_balance

**FROM** Accounts

**WHERE** AccountID = p\_from\_account\_id;

IF v\_from\_balance &lt; p\_amount **THEN**

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in the source account');

**END** IF;

**END**;

-- Transfer the funds

UPDATE Accounts

**SET** Balance = Balance - p\_amount

**WHERE** AccountID = p\_from\_account\_id;

UPDATE Accounts

**SET** Balance = Balance + p\_amount

**WHERE** AccountID = p\_to\_account\_id;

-- Commit the transaction

**COMMIT**;

EXCEPTION

**WHEN** OTHERS **THEN**

-- Rollback the transaction

**ROLLBACK** **TO** transfer\_start;

-- Log the error

**INSERT** **INTO** ErrorLog (Message) **VALUES** ('Transfer funds failed: ' || SQLERRM);

**END**;

/

**Scenario 2:** Manage errors when updating employee salaries.

* + **Question:** Write a stored procedure **UpdateSalary** that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.

**CREATE** **OR** REPLACE **PROCEDURE** UpdateSalary (

p\_employee\_id NUMBER,

p\_percentage NUMBER

) **IS**

PRAGMA AUTONOMOUS\_TRANSACTION;

**BEGIN**

-- Begin transaction

**SAVEPOINT** salary\_update\_start;

-- Update the salary

UPDATE Employees

**SET** Salary = Salary \* (1 + p\_percentage / 100)

**WHERE** EmployeeID = p\_employee\_id;

-- Check if any row was updated

IF **SQL**%ROWCOUNT = 0 **THEN**

-- Log the error

**INSERT** **INTO** ErrorLog (Message) **VALUES** ('Updating salary failed: Employee ID does not exist');

**END** IF;

-- Commit the transaction

**COMMIT**;

EXCEPTION

**WHEN** OTHERS **THEN**

-- Rollback the transaction

**ROLLBACK** **TO** salary\_update\_start;

-- Log the error

**INSERT** **INTO** ErrorLog (Message) **VALUES** ('Updating salary failed: ' || SQLERRM);

**END**;

/

**Scenario 3:** Ensure data integrity when adding a new customer.

* + **Question:** Write a stored procedure **AddNewCustomer** that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.

**CREATE** **OR** REPLACE **PROCEDURE** AddNewCustomer (

p\_customer\_id NUMBER,

p\_name VARCHAR2,

p\_dob DATE,

p\_balance NUMBER,

p\_last\_modified DATE

) **IS**

PRAGMA AUTONOMOUS\_TRANSACTION;

**BEGIN**

-- Begin transaction

**SAVEPOINT** customer\_insert\_start;

-- Check if the customer already exists

**DECLARE**

v\_existing\_customer NUMBER;

**BEGIN**

**SELECT** COUNT(\*) **INTO** v\_existing\_customer

**FROM** Customers

**WHERE** CustomerID = p\_customer\_id;

IF v\_existing\_customer &gt; 0 **THEN**

RAISE\_APPLICATION\_ERROR(-20002, 'Adding new customer failed: Customer ID already exists');

**END** IF;

**END**;

-- Add the new customer

**INSERT** **INTO** Customers (

CustomerID,

Name,

DOB,

Balance,

LastModified

) **VALUES** (

p\_customer\_id,

p\_name,

p\_dob,

p\_balance,

p\_last\_modified

);

-- Commit the transaction

**COMMIT**;

EXCEPTION

**WHEN** OTHERS **THEN**

-- Rollback the transaction

**ROLLBACK** **TO** customer\_insert\_start;

-- Log the error

**INSERT** **INTO** ErrorLog (Message) **VALUES** ('Adding new customer failed: ' || SQLERRM);

**END**;

/

**Exercise 3: Stored Procedures**

**Scenario 1:** The bank needs to process monthly interest for all savings accounts.

* + **Question:** Write a stored procedure **ProcessMonthlyInterest** that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.

**CREATE** **OR** REPLACE **PROCEDURE** ProcessMonthlyInterest

**IS**

**BEGIN**

**FOR** account **IN** (

**SELECT** AccountID, Balance

**FROM** Accounts

**WHERE** AccountType = 'Savings'

) LOOP

UPDATE Accounts

**SET** Balance = Balance \* 1.01

**WHERE** AccountID = account.AccountID;

**END** LOOP;

**COMMIT**;

**END**;

/

**Scenario 2:** The bank wants to implement a bonus scheme for employees based on their performance.

* + **Question:** Write a stored procedure **UpdateEmployeeBonus** that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

**CREATE** **OR** REPLACE **PROCEDURE** UpdateEmployeeBonus (

p\_department VARCHAR2,

p\_bonus\_percentage NUMBER

) **IS**

**BEGIN**

UPDATE Employees

**SET** Salary = Salary \* (1 + p\_bonus\_percentage / 100)

**WHERE** Department = p\_department;

**COMMIT**;

**END**;

/

**Scenario 3:** Customers should be able to transfer funds between their accounts.

* + **Question:** Write a stored procedure **TransferFunds** that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.

**CREATE** **OR** REPLACE **PROCEDURE** TransferFunds (

p\_from\_account\_id NUMBER,

p\_to\_account\_id NUMBER,

p\_amount NUMBER

) **IS**

**BEGIN**

UPDATE Accounts

**SET** Balance = Balance - p\_amount

**WHERE** AccountID = p\_from\_account\_id

**AND** Balance &gt;= p\_amount;

UPDATE Accounts

**SET** Balance = Balance + p\_amount

**WHERE** AccountID = p\_to\_account\_id;

**COMMIT**;

EXCEPTION

**WHEN** NO\_DATA\_FOUND **THEN**

DBMS\_OUTPUT.PUT\_LINE('Error: Source account not found');

**WHEN** INSUFFICIENT\_FUNDS **THEN**

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in the source account');

**END**;

/

**Exercise 4: Functions**

**Scenario 1:** Calculate the age of customers for eligibility checks.

* + **Question:** Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

**CREATE** **OR** REPLACE **FUNCTION** CalculateAge (

p\_dob DATE

) **RETURN** NUMBER **IS**

v\_age NUMBER;

**BEGIN**

**SELECT** MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12 **INTO** v\_age **FROM** dual;

**RETURN** FLOOR(v\_age);

**END**;

/

**SELECT** CalculateAge(TO\_DATE('1980-01-01', 'YYYY-MM-DD')) **FROM** dual;

**Scenario 2:** The bank needs to compute the monthly installment for a loan.

* + **Question:** Write a function **CalculateMonthlyInstallment** that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

**CREATE** **OR** REPLACE **FUNCTION** CalculateMonthlyInstallment (

p\_loan\_amount NUMBER,

p\_interest\_rate NUMBER,

p\_loan\_duration NUMBER

) **RETURN** NUMBER **IS**

v\_monthly\_interest\_rate NUMBER;

v\_number\_of\_payments NUMBER;

**BEGIN**

v\_monthly\_interest\_rate := p\_interest\_rate / 100 / 12;

v\_number\_of\_payments := p\_loan\_duration \* 12;

**RETURN** (p\_loan\_amount \* v\_monthly\_interest\_rate \* POWER(1 + v\_monthly\_interest\_rate, v\_number\_of\_payments)) / (POWER(1 + v\_monthly\_interest\_rate, v\_number\_of\_payments) - 1);

**END**;

/

**SELECT** CalculateMonthlyInstallment(

10000, -- Loan Amount

5, -- Interest Rate (e.g., 5% per annum)

5 -- Loan Duration in Years

)

**AS** MonthlyInstallment

**FROM** dual;

**Scenario 3:** Check if a customer has sufficient balance before making a transaction.

* + **Question:** Write a function **HasSufficientBalance** that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.

**CREATE** **OR** REPLACE **FUNCTION** HasSufficientBalance (

p\_account\_id NUMBER,

p\_amount NUMBER

) **RETURN** BOOLEAN **IS**

v\_balance NUMBER;

**BEGIN**

**SELECT** Balance **INTO** v\_balance **FROM** Accounts **WHERE** AccountID = p\_account\_id;

**RETURN** v\_balance &gt;= p\_amount;

**END**;

/

**SELECT** HasSufficientBalance(1, 1000) **AS** SufficientBalance

**FROM** dual;

**Exercise 5: Triggers**

**Scenario 1:** Automatically update the last modified date when a customer's record is updated.

* + **Question:** Write a trigger **UpdateCustomerLastModified** that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE OF Name, DOB, Balance, LastModified ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

UPDATE Customers

SET Name = 'Arijit Das (updated)'

WHERE CustomerID = 1;

SELECT \* FROM Customers WHERE CustomerID = 1;

**Scenario 2:** Maintain an audit log for all transactions.

* + **Question:** Write a trigger **LogTransaction** that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

CREATE TABLE AuditLog (

TransactionID NUMBER,

TransactionDate DATE,

AccountID NUMBER,

Amount NUMBER,

TransactionType VARCHAR2(10),

ModifiedDate DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (

TransactionID,

TransactionDate,

AccountID,

Amount,

TransactionType,

ModifiedDate

) VALUES (

:NEW.TransactionID,

SYSDATE,

:NEW.AccountID,

:NEW.Amount,

:NEW.TransactionType,

SYSDATE

);

END;

/

INSERT INTO Transactions (

TransactionID,

AccountID,

TransactionDate,

Amount,

TransactionType

) VALUES (

10,

1,

SYSDATE,

1000,

'Deposit'

);

SELECT \* FROM AuditLog;

**Scenario 3:** Enforce business rules on deposits and withdrawals.

* + **Question:** Write a trigger **CheckTransactionRules** that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = :NEW.AccountID;

IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Withdrawal amount exceeds account balance');

ELSIF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Deposit amount must be positive');

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Account not found');

END;

/

-- Testing deposit with a positive amount

INSERT INTO Transactions (

TransactionID,

AccountID,

TransactionDate,

Amount,

TransactionType

) VALUES (

11,

1,

SYSDATE,

2000,

'Deposit'

);

-- Testing withdrawal with an amount less than the balance

INSERT INTO Transactions (

TransactionID,

AccountID,

TransactionDate,

Amount,

TransactionType

) VALUES (

12,

1,

SYSDATE,

500,

'Withdrawal'

);

-- Testing withdrawal with an amount greater than the balance

INSERT INTO Transactions (

TransactionID,

AccountID,

TransactionDate,

Amount,

TransactionType

) VALUES (

13,

1,

SYSDATE,

2500,

'Withdrawal'

);

-- Testing deposit with a non-positive amount

INSERT INTO Transactions (

TransactionID,

AccountID,

TransactionDate,

Amount,

TransactionType

) VALUES (

14,

1,

SYSDATE,

-500,

'Deposit'

);

SELECT \* FROM USER\_ERRORS;

**Exercise 6: Cursors**

**Scenario 1:** Generate monthly statements for all customers.

* + **Question:** Write a PL/SQL block using an explicit cursor **GenerateMonthlyStatements** that retrieves all transactions for the current month and prints a statement for each customer.

DECLARE

CURSOR GenerateMonthlyStatements IS

SELECT c.CustomerID, c.Name, t.TransactionID, t.Amount, t.TransactionType, t.TransactionDate

FROM Customers c

JOIN Accounts a ON c.CustomerID = a.CustomerID

JOIN Transactions t ON a.AccountID = t.AccountID

WHERE EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE)

ORDER BY c.CustomerID, t.TransactionDate;

v\_customer\_id NUMBER;

v\_name VARCHAR2(100);

v\_transaction\_id NUMBER;

v\_amount NUMBER;

v\_transaction\_type VARCHAR2(10);

v\_transaction\_date DATE;

BEGIN

FOR statement IN GenerateMonthlyStatements LOOP

v\_customer\_id := statement.CustomerID;

v\_name := statement.Name;

IF v\_transaction\_type = 'Deposit' THEN

v\_amount := v\_amount + statement.Amount;

ELSE

v\_amount := v\_amount - statement.Amount;

END IF;

IF v\_transaction\_id = 1 OR v\_transaction\_id IS NULL THEN

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || v\_customer\_id);

DBMS\_OUTPUT.PUT\_LINE('Name: ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('---------------------');

END IF;

DBMS\_OUTPUT.PUT\_LINE('Transaction ID: ' || v\_transaction\_id);

DBMS\_OUTPUT.PUT\_LINE('Amount: ' || v\_amount);

DBMS\_OUTPUT.PUT\_LINE('Transaction Type: ' || v\_transaction\_type);

DBMS\_OUTPUT.PUT\_LINE('Transaction Date: ' || statement.TransactionDate);

DBMS\_OUTPUT.PUT\_LINE('---------------------');

v\_transaction\_id := statement.TransactionID;

END LOOP;

END;

/

**Scenario 2:** Apply annual fee to all accounts.

* + **Question:** Write a PL/SQL block using an explicit cursor **ApplyAnnualFee** that deducts an annual maintenance fee from the balance of all accounts.

**DECLARE**

**CURSOR** ApplyAnnualFee **IS**

**SELECT** AccountID, Balance

**FROM** Accounts;

v\_account\_id NUMBER;

v\_balance NUMBER;

**BEGIN**

**FOR** account **IN** ApplyAnnualFee LOOP

UPDATE Accounts

**SET** Balance = Balance - 10 -- Annual maintenance fee of 10

**WHERE** AccountID = v\_account\_id;

**END** LOOP;

**END**;

/

**Scenario 3:** Update the interest rate for all loans based on a new policy.

* + **Question:** Write a PL/SQL block using an explicit cursor **UpdateLoanInterestRates** that fetches all loans and updates their interest rates based on the new policy.

DECLARE

CURSOR UpdateLoanInterestRates IS

SELECT LoanID, InterestRate

FROM Loans;

v\_loan\_id NUMBER;

v\_interest\_rate NUMBER;

BEGIN

FOR loan IN UpdateLoanInterestRates LOOP

UPDATE Loans

SET InterestRate = CASE

WHEN InterestRate < 5 THEN InterestRate + 0.25

WHEN InterestRate >= 5 AND InterestRate < 8 THEN InterestRate + 0.1

ELSE InterestRate

END

WHERE LoanID = v\_loan\_id;

END LOOP;

END;

/

SELECT LoanID, InterestRate FROM Loans;

**Exercise 7: Packages**

**Scenario 1:** Group all customer-related procedures and functions into a package.

* + **Question:** Create a package **CustomerManagement** with procedures for adding a new customer, updating customer details, and a function to get customer balance.

**CREATE** **OR** REPLACE PACKAGE CustomerManagement

**AS**

**PROCEDURE** AddNewCustomer (p\_customer\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER, p\_last\_modified DATE);

**PROCEDURE** UpdateCustomerDetails (p\_customer\_id NUMBER, p\_new\_name VARCHAR2, p\_new\_dob DATE, p\_new\_balance NUMBER, p\_new\_last\_modified DATE);

**FUNCTION** GetCustomerBalance (p\_customer\_id NUMBER) **RETURN** NUMBER;

**END** CustomerManagement;

/

**CREATE** **OR** REPLACE PACKAGE BODY CustomerManagement

**AS**

**PROCEDURE** AddNewCustomer (p\_customer\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER, p\_last\_modified DATE)

**IS**

**BEGIN**

**INSERT** **INTO** Customers (

CustomerID,

Name,

DOB,

Balance,

LastModified

) **VALUES** (

p\_customer\_id,

p\_name,

p\_dob,

p\_balance,

p\_last\_modified

);

**END**;

**PROCEDURE** UpdateCustomerDetails (p\_customer\_id NUMBER, p\_new\_name VARCHAR2, p\_new\_dob DATE, p\_new\_balance NUMBER, p\_new\_last\_modified DATE)

**IS**

**BEGIN**

UPDATE Customers

**SET**

Name = p\_new\_name,

DOB = p\_new\_dob,

Balance = p\_new\_balance,

LastModified = p\_new\_last\_modified

**WHERE** CustomerID = p\_customer\_id;

**END**;

**FUNCTION** GetCustomerBalance (p\_customer\_id NUMBER) **RETURN** NUMBER

**IS**

v\_balance NUMBER;

**BEGIN**

**SELECT** Balance **INTO** v\_balance

**FROM** Customers

**WHERE** CustomerID = p\_customer\_id;

**RETURN** v\_balance;

**END**;

**END** CustomerManagement;

/

**Scenario 2:** Create a package to manage employee data.

* + **Question:** Write a package **EmployeeManagement** with procedures to hire new employees, update employee details, and a function to calculate annual salary.

**CREATE** **OR** REPLACE PACKAGE EmployeeManagement

**AS**

**PROCEDURE** HireNewEmployee (p\_employee\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2, p\_hire\_date DATE);

**PROCEDURE** UpdateEmployeeDetails (p\_employee\_id NUMBER, p\_new\_name VARCHAR2, p\_new\_position VARCHAR2, p\_new\_salary NUMBER, p\_new\_department VARCHAR2, p\_new\_hire\_date DATE);

**FUNCTION** CalculateAnnualSalary (p\_salary NUMBER) **RETURN** NUMBER;

**END** EmployeeManagement;

/

**CREATE** **OR** REPLACE PACKAGE BODY EmployeeManagement

**AS**

**PROCEDURE** HireNewEmployee (p\_employee\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2, p\_hire\_date DATE)

**IS**

**BEGIN**

**INSERT** **INTO** Employees (

EmployeeID,

Name,

Position,

Salary,

Department,

HireDate

) **VALUES** (

p\_employee\_id,

p\_name,

p\_position,

p\_salary,

p\_department,

p\_hire\_date

);

**END**;

**PROCEDURE** UpdateEmployeeDetails (p\_employee\_id NUMBER, p\_new\_name VARCHAR2, p\_new\_position VARCHAR2, p\_new\_salary NUMBER, p\_new\_department VARCHAR2, p\_new\_hire\_date DATE)

**IS**

**BEGIN**

UPDATE Employees

**SET**

Name = p\_new\_name,

Position = p\_new\_position,

Salary = p\_new\_salary,

Department = p\_new\_department,

HireDate = p\_new\_hire\_date

**WHERE** EmployeeID = p\_employee\_id;

**END**;

**FUNCTION** CalculateAnnualSalary (p\_salary NUMBER) **RETURN** NUMBER

**IS**

v\_annual\_salary NUMBER;

**BEGIN**

v\_annual\_salary := p\_salary \* 12;

**RETURN** v\_annual\_salary;

**END**;

**END** EmployeeManagement;

/

**Scenario 3:** Group all account-related operations into a package.

* + **Question:** Create a package **AccountOperations** with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.

**CREATE** **OR** REPLACE PACKAGE AccountOperations

**AS**

**PROCEDURE** OpenNewAccount (p\_account\_id NUMBER, p\_customer\_id NUMBER, p\_account\_type VARCHAR2, p\_balance NUMBER, p\_last\_modified DATE);

**PROCEDURE** CloseAccount (p\_account\_id NUMBER);

**FUNCTION** GetTotalCustomerBalance (p\_customer\_id NUMBER) **RETURN** NUMBER;

**END** AccountOperations;

/

**CREATE** **OR** REPLACE PACKAGE BODY AccountOperations

**AS**

**PROCEDURE** OpenNewAccount (p\_account\_id NUMBER, p\_customer\_id NUMBER, p\_account\_type VARCHAR2, p\_balance NUMBER, p\_last\_modified DATE)

**IS**

**BEGIN**

**INSERT** **INTO** Accounts (

AccountID,

CustomerID,

AccountType,

Balance,

LastModified

) **VALUES** (

p\_account\_id,

p\_customer\_id,

p\_account\_type,

p\_balance,

p\_last\_modified

);

**END**;

**PROCEDURE** CloseAccount (p\_account\_id NUMBER)

**IS**

**BEGIN**

UPDATE Accounts

**SET** Balance = 0, LastModified = SYSDATE

**WHERE** AccountID = p\_account\_id;

**END**;

**FUNCTION** GetTotalCustomerBalance (p\_customer\_id NUMBER) **RETURN** NUMBER

**IS**

v\_total\_balance NUMBER;

**BEGIN**

**SELECT** SUM(Balance) **INTO** v\_total\_balance

**FROM** Accounts

**WHERE** CustomerID = p\_customer\_id;

**RETURN** v\_total\_balance;

**END**;

**END** AccountOperations;

/

**Schema to be Created**

*CREATE TABLE Customers (*

*CustomerID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*DOB DATE,*

*Balance NUMBER,*

*LastModified DATE*

*);*

*CREATE TABLE Accounts (*

*AccountID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*AccountType VARCHAR2(20),*

*Balance NUMBER,*

*LastModified DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Transactions (*

*TransactionID NUMBER PRIMARY KEY,*

*AccountID NUMBER,*

*TransactionDate DATE,*

*Amount NUMBER,*

*TransactionType VARCHAR2(10),*

*FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)*

*);*

*CREATE TABLE Loans (*

*LoanID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*LoanAmount NUMBER,*

*InterestRate NUMBER,*

*StartDate DATE,*

*EndDate DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Employees (*

*EmployeeID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*Position VARCHAR2(50),*

*Salary NUMBER,*

*Department VARCHAR2(50),*

*HireDate DATE*

*);*

**Example Scripts for Sample Data Insertion**

*INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)*

*VALUES (1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);*

*INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)*

*VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);*

*INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)*

*VALUES (1, 1, 'Savings', 1000, SYSDATE);*

*INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)*

*VALUES (2, 2, 'Checking', 1500, SYSDATE);*

*INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)*

*VALUES (1, 1, SYSDATE, 200, 'Deposit');*

*INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)*

*VALUES (2, 2, SYSDATE, 300, 'Withdrawal');*

*INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)*

*VALUES (1, 1, 5000, 5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));*

*INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)*

*VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO\_DATE('2015-06-15', 'YYYY-MM-DD'));*

*INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)*

*VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO\_DATE('2017-03-20', 'YYYY-MM-DD'));*