**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 cust\_cursor IS*

*SELECT customer\_id, age, loan\_interest\_rate*

*FROM customers*

*WHERE age > 60;*

*v\_customer\_id customers.customer\_id%TYPE;*

*v\_age customers.age%TYPE;*

*v\_loan\_interest\_rate customers.loan\_interest\_rate%TYPE;*

*BEGIN*

*OPEN cust\_cursor;*

*LOOP*

*FETCH cust\_cursor INTO v\_customer\_id, v\_age, v\_loan\_interest\_rate;*

*EXIT WHEN cust\_cursor%NOTFOUND;*

*UPDATE customers*

*SET loan\_interest\_rate = loan\_interest\_rate \* 0.99*

*WHERE customer\_id = v\_customer\_id;*

*END LOOP;*

*CLOSE cust\_cursor;*

*COMMIT;*

*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 cust\_cursor IS*

*SELECT customer\_id, balance*

*FROM customers*

*WHERE balance > 10000;*

*v\_customer\_id customers.customer\_id%TYPE;*

*v\_balance customers.balance%TYPE;*

*BEGIN*

*OPEN cust\_cursor;*

*LOOP*

*FETCH cust\_cursor INTO v\_customer\_id, v\_balance;*

*EXIT WHEN cust\_cursor%NOTFOUND;*

*UPDATE customers*

*SET IsVIP = TRUE*

*WHERE customer\_id = v\_customer\_id;*

*END LOOP;*

*CLOSE cust\_cursor;*

*COMMIT;*

*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 loan\_cursor IS*

*SELECT customer\_id, loan\_due\_date*

*FROM loans*

*WHERE loan\_due\_date BETWEEN SYSDATE AND SYSDATE + 30;*

*v\_customer\_id loans.customer\_id%TYPE;*

*v\_loan\_due\_date loans.loan\_due\_date%TYPE;*

*BEGIN*

*OPEN loan\_cursor;*

*LOOP*

*FETCH loan\_cursor INTO v\_customer\_id, v\_loan\_due\_date;*

*EXIT WHEN loan\_cursor%NOTFOUND;*

*DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ' || v\_customer\_id ||*

*', your loan is due on ' || TO\_CHAR(v\_loan\_due\_date, 'DD-MON-YYYY'));*

*END LOOP;*

*CLOSE loan\_cursor;*

*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 IN NUMBER,*

*p\_to\_account\_id IN NUMBER,*

*p\_amount IN NUMBER*

*) IS*

*insufficient\_funds EXCEPTION;*

*insufficient\_funds\_message VARCHAR2(100);*

*BEGIN*

*-- Check if the from account has sufficient funds*

*DECLARE*

*v\_balance NUMBER;*

*BEGIN*

*SELECT balance INTO v\_balance*

*FROM Accounts*

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

*IF v\_balance < p\_amount THEN*

*RAISE insufficient\_funds;*

*END IF;*

*EXCEPTION*

*WHEN NO\_DATA\_FOUND THEN*

*insufficient\_funds\_message := 'From account does not exist';*

*RAISE insufficient\_funds;*

*END;*

*-- Perform the transfer*

*UPDATE Accounts*

*SET balance = balance - p\_amount*

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

*UPDATE Accounts*

*SET balance = balance + p\_amount*

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

*COMMIT;*

*EXCEPTION*

*WHEN insufficient\_funds THEN*

*DBMS\_OUTPUT.PUT\_LINE(insufficient\_funds\_message);*

*ROLLBACK;*

*WHEN OTHERS THEN*

*DBMS\_OUTPUT.PUT\_LINE('An error occurred during the transfer: ' || SQLERRM);*

*ROLLBACK;*

*END SafeTransferFunds;*

*/*

**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 IN NUMBER,*

*p\_percentage IN NUMBER*

*) IS*

*BEGIN*

*-- Update the salary*

*UPDATE Employees*

*SET salary = salary + (salary \* (p\_percentage / 100))*

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

*-- Check if the update was successful*

*IF SQL%ROWCOUNT = 0 THEN*

*RAISE\_APPLICATION\_ERROR(-20001, 'Employee ID does not exist');*

*END IF;*

*COMMIT;*

*EXCEPTION*

*WHEN OTHERS THEN*

*DBMS\_OUTPUT.PUT\_LINE('An error occurred while updating salary: ' || SQLERRM);*

*ROLLBACK;*

*END UpdateSalary;*

*/*

**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 IN NUMBER,*

*p\_name IN VARCHAR2,*

*p\_age IN NUMBER,*

*p\_balance IN NUMBER*

*) IS*

*BEGIN*

*-- Insert a new customer*

*INSERT INTO Customers (customer\_id, name, age, balance)*

*VALUES (p\_customer\_id, p\_name, p\_age, p\_balance);*

*COMMIT;*

*EXCEPTION*

*WHEN DUP\_VAL\_ON\_INDEX THEN*

*DBMS\_OUTPUT.PUT\_LINE('Error: Customer with the same ID already exists');*

*ROLLBACK;*

*WHEN OTHERS THEN*

*DBMS\_OUTPUT.PUT\_LINE('An error occurred while adding a new customer: ' || SQLERRM);*

*ROLLBACK;*

*END AddNewCustomer;*

*/*

**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*

*-- Update the balance of all savings accounts by applying a 1% interest rate*

*UPDATE Accounts*

*SET balance = balance + (balance \* 0.01)*

*WHERE account\_type = 'SAVINGS';*

*COMMIT;*

*END ProcessMonthlyInterest;*

*/*

**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\_id IN NUMBER,*

*p\_bonus\_percentage IN NUMBER*

*) IS*

*BEGIN*

*UPDATE Employees*

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

*WHERE department\_id = p\_department\_id;*

*COMMIT;*

*END UpdateEmployeeBonus;*

*/*

**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 IN NUMBER,*

*p\_to\_account\_id IN NUMBER,*

*p\_amount IN NUMBER*

*) IS*

*insufficient\_funds EXCEPTION;*

*BEGIN*

*-- Check if the from account has sufficient funds*

*DECLARE*

*v\_balance NUMBER;*

*BEGIN*

*SELECT balance INTO v\_balance*

*FROM Accounts*

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

*FOR UPDATE; -- Lock the row to prevent concurrent updates*

*IF v\_balance < p\_amount THEN*

*RAISE insufficient\_funds;*

*END IF;*

*EXCEPTION*

*WHEN NO\_DATA\_FOUND THEN*

*DBMS\_OUTPUT.PUT\_LINE('From account does not exist');*

*RAISE insufficient\_funds;*

*END;*

*-- Perform the transfer*

*UPDATE Accounts*

*SET balance = balance - p\_amount*

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

*UPDATE Accounts*

*SET balance = balance + p\_amount*

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

*COMMIT;*

*EXCEPTION*

*WHEN insufficient\_funds THEN*

*DBMS\_OUTPUT.PUT\_LINE('Insufficient funds in the from account');*

*ROLLBACK;*

*WHEN OTHERS THEN*

*DBMS\_OUTPUT.PUT\_LINE('An error occurred during the transfer: ' || SQLERRM);*

*ROLLBACK;*

*END TransferFunds;*

*/*

**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\_date\_of\_birth IN DATE*

*) RETURN NUMBER IS*

*v\_age NUMBER;*

*BEGIN*

*-- Calculate age in years*

*v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_date\_of\_birth) / 12);*

*RETURN v\_age;*

*END CalculateAge;*

*/*

**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 IN NUMBER,*

*p\_annual\_interest\_rate IN NUMBER,*

*p\_loan\_duration\_years IN NUMBER*

*) RETURN NUMBER IS*

*v\_monthly\_interest\_rate NUMBER;*

*v\_number\_of\_payments NUMBER;*

*v\_monthly\_installment NUMBER;*

*BEGIN*

*-- Convert annual interest rate to monthly interest rate*

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

*-- Calculate the number of payments*

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

*-- Calculate monthly installment using the formula for an annuity*

*v\_monthly\_installment := p\_loan\_amount \* v\_monthly\_interest\_rate /*

*(1 - POWER(1 + v\_monthly\_interest\_rate, -v\_number\_of\_payments));*

*RETURN v\_monthly\_installment;*

*END CalculateMonthlyInstallment;*

*/*

**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 IN NUMBER,*

*p\_amount IN NUMBER*

*) RETURN BOOLEAN IS*

*v\_balance NUMBER;*

*BEGIN*

*-- Retrieve the account balance*

*SELECT balance INTO v\_balance*

*FROM Accounts*

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

*-- Check if the balance is sufficient*

*IF v\_balance >= p\_amount THEN*

*RETURN TRUE;*

*ELSE*

*RETURN FALSE;*

*END IF;*

*EXCEPTION*

*WHEN NO\_DATA\_FOUND THEN*

*RETURN FALSE;*

*WHEN OTHERS THEN*

*RETURN FALSE;*

*END HasSufficientBalance;*

*/*

**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 ON Customers*

*FOR EACH ROW*

*BEGIN*

*-- Update the LastModified column to the current date*

*:NEW.LastModified := SYSDATE;*

*END UpdateCustomerLastModified;*

*/*

**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 OR REPLACE TRIGGER LogTransaction*

*AFTER INSERT ON Transactions*

*FOR EACH ROW*

*BEGIN*

*-- Insert a record into the AuditLog table*

*INSERT INTO AuditLog (transaction\_id, account\_id, transaction\_type, amount, transaction\_date)*

*VALUES (:NEW.transaction\_id, :NEW.account\_id, :NEW.transaction\_type, :NEW.amount, SYSDATE);*

*END LogTransaction;*

*/*

**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*

*-- Retrieve the current balance of the account*

*SELECT balance INTO v\_balance*

*FROM Accounts*

*WHERE account\_id = :NEW.account\_id*

*FOR UPDATE; -- Lock the row to prevent concurrent updates*

*-- Check if the transaction is a withdrawal and ensure it does not exceed the balance*

*IF :NEW.transaction\_type = 'WITHDRAWAL' THEN*

*IF :NEW.amount > v\_balance THEN*

*RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds for the withdrawal');*

*END IF;*

*END IF;*

*-- Check if the transaction is a deposit and ensure the amount is positive*

*IF :NEW.transaction\_type = 'DEPOSIT' THEN*

*IF :NEW.amount <= 0 THEN*

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

*END IF;*

*END IF;*

*END CheckTransactionRules;*

*/*

**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 cur\_monthly\_statements IS*

*SELECT c.customer\_id, c.name, t.transaction\_id, t.amount, t.transaction\_date, t.transaction\_type*

*FROM Customers c*

*JOIN Transactions t ON c.customer\_id = t.customer\_id*

*WHERE TRUNC(t.transaction\_date, 'MM') = TRUNC(SYSDATE, 'MM');*

*v\_customer\_id Customers.customer\_id%TYPE;*

*v\_name Customers.name%TYPE;*

*v\_transaction\_id Transactions.transaction\_id%TYPE;*

*v\_amount Transactions.amount%TYPE;*

*v\_transaction\_date Transactions.transaction\_date%TYPE;*

*v\_transaction\_type Transactions.transaction\_type%TYPE;*

*BEGIN*

*OPEN cur\_monthly\_statements;*

*LOOP*

*FETCH cur\_monthly\_statements INTO v\_customer\_id, v\_name, v\_transaction\_id, v\_amount, v\_transaction\_date, v\_transaction\_type;*

*EXIT WHEN cur\_monthly\_statements%NOTFOUND;*

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

*DBMS\_OUTPUT.PUT\_LINE('Transaction ID: ' || v\_transaction\_id || ', Amount: ' || v\_amount || ', Date: ' || TO\_CHAR(v\_transaction\_date, 'DD-MON-YYYY') || ', Type: ' || v\_transaction\_type);*

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

*END LOOP;*

*CLOSE cur\_monthly\_statements;*

*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 cur\_apply\_annual\_fee IS*

*SELECT account\_id, balance*

*FROM Accounts;*

*v\_account\_id Accounts.account\_id%TYPE;*

*v\_balance Accounts.balance%TYPE;*

*v\_annual\_fee CONSTANT NUMBER := 50; -- Set the annual fee amount*

*BEGIN*

*OPEN cur\_apply\_annual\_fee;*

*LOOP*

*FETCH cur\_apply\_annual\_fee INTO v\_account\_id, v\_balance;*

*EXIT WHEN cur\_apply\_annual\_fee%NOTFOUND;*

*-- Deduct the annual fee*

*UPDATE Accounts*

*SET balance = balance - v\_annual\_fee*

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

*END LOOP;*

*CLOSE cur\_apply\_annual\_fee;*

*COMMIT;*

*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 cur\_update\_loan\_interest\_rates IS*

*SELECT loan\_id, interest\_rate*

*FROM Loans;*

*v\_loan\_id Loans.loan\_id%TYPE;*

*v\_interest\_rate Loans.interest\_rate%TYPE;*

*v\_new\_interest\_rate Loans.interest\_rate%TYPE;*

*BEGIN*

*OPEN cur\_update\_loan\_interest\_rates;*

*LOOP*

*FETCH cur\_update\_loan\_interest\_rates INTO v\_loan\_id, v\_interest\_rate;*

*EXIT WHEN cur\_update\_loan\_interest\_rates%NOTFOUND;*

*-- Apply the new policy to calculate the new interest rate*

*v\_new\_interest\_rate := v\_interest\_rate \* 1.05; -- For example, increase by 5%*

*-- Update the loan interest rate*

*UPDATE Loans*

*SET interest\_rate = v\_new\_interest\_rate*

*WHERE loan\_id = v\_loan\_id;*

*END LOOP;*

*CLOSE cur\_update\_loan\_interest\_rates;*

*COMMIT;*

*END;*

*/*

**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 to add a new customer*

*PROCEDURE AddCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_date\_of\_birth IN DATE);*

*-- Procedure to update customer details*

*PROCEDURE UpdateCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_date\_of\_birth IN DATE);*

*-- Function to get customer balance*

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

*END CustomerManagement;*

*/*

*CREATE OR REPLACE PACKAGE BODY CustomerManagement AS*

*PROCEDURE AddCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_date\_of\_birth IN DATE) IS*

*BEGIN*

*INSERT INTO Customers (customer\_id, name, date\_of\_birth, last\_modified)*

*VALUES (p\_customer\_id, p\_name, p\_date\_of\_birth, SYSDATE);*

*END AddCustomer;*

*PROCEDURE UpdateCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_date\_of\_birth IN DATE) IS*

*BEGIN*

*UPDATE Customers*

*SET name = p\_name, date\_of\_birth = p\_date\_of\_birth, last\_modified = SYSDATE*

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

*END UpdateCustomer;*

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

*v\_balance NUMBER;*

*BEGIN*

*SELECT SUM(balance) INTO v\_balance*

*FROM Accounts*

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

*RETURN v\_balance;*

*EXCEPTION*

*WHEN NO\_DATA\_FOUND THEN*

*RETURN 0;*

*END GetCustomerBalance;*

*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 to hire a new employee*

*PROCEDURE HireEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_department\_id IN NUMBER, p\_salary IN NUMBER);*

*-- Procedure to update employee details*

*PROCEDURE UpdateEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_department\_id IN NUMBER, p\_salary IN NUMBER);*

*-- Function to calculate annual salary*

*FUNCTION CalculateAnnualSalary(p\_employee\_id IN NUMBER) RETURN NUMBER;*

*END EmployeeManagement;*

*/*

*CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS*

*PROCEDURE HireEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_department\_id IN NUMBER, p\_salary IN NUMBER) IS*

*BEGIN*

*INSERT INTO Employees (employee\_id, name, department\_id, salary, hire\_date)*

*VALUES (p\_employee\_id, p\_name, p\_department\_id, p\_salary, SYSDATE);*

*END HireEmployee;*

*PROCEDURE UpdateEmployee(p\_employee\_id IN NUMBER, p\_name IN VARCHAR2, p\_department\_id IN NUMBER, p\_salary IN NUMBER) IS*

*BEGIN*

*UPDATE Employees*

*SET name = p\_name, department\_id = p\_department\_id, salary = p\_salary*

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

*END UpdateEmployee;*

*FUNCTION CalculateAnnualSalary(p\_employee\_id IN NUMBER) RETURN NUMBER IS*

*v\_salary NUMBER;*

*BEGIN*

*SELECT salary \* 12 INTO v\_salary*

*FROM Employees*

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

*RETURN v\_salary;*

*EXCEPTION*

*WHEN NO\_DATA\_FOUND THEN*

*RETURN 0;*

*END CalculateAnnualSalary;*

*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 to open a new account*

*PROCEDURE OpenAccount(p\_account\_id IN NUMBER, p\_customer\_id IN NUMBER, p\_account\_type IN VARCHAR2, p\_initial\_balance IN NUMBER);*

*-- Procedure to close an account*

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

*-- Function to get the total balance of a customer across all accounts*

*FUNCTION GetTotalBalance(p\_customer\_id IN NUMBER) RETURN NUMBER;*

*END AccountOperations;*

*/*

*CREATE OR REPLACE PACKAGE BODY AccountOperations AS*

*PROCEDURE OpenAccount(p\_account\_id IN NUMBER, p\_customer\_id IN NUMBER, p\_account\_type IN VARCHAR2, p\_initial\_balance IN NUMBER) IS*

*BEGIN*

*INSERT INTO Accounts (account\_id, customer\_id, account\_type, balance, date\_opened)*

*VALUES (p\_account\_id, p\_customer\_id, p\_account\_type, p\_initial\_balance, SYSDATE);*

*END OpenAccount;*

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

*BEGIN*

*DELETE FROM Accounts*

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

*END CloseAccount;*

*FUNCTION GetTotalBalance(p\_customer\_id IN NUMBER) RETURN NUMBER IS*

*v\_total\_balance NUMBER;*

*BEGIN*

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

*FROM Accounts*

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

*RETURN v\_total\_balance;*

*EXCEPTION*

*WHEN NO\_DATA\_FOUND THEN*

*RETURN 0;*

*END GetTotalBalance;*

*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'));*