**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.

Code:

DECLARE

CURSOR c\_customers IS

SELECT c.CustomerID, l.LoanID, l.InterestRate

FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID

WHERE TRUNC(MONTHS\_BETWEEN(SYSDATE, c.DOB) / 12) > 60;

BEGIN

FOR rec IN c\_customers LOOP

UPDATE Loans

SET InterestRate = rec.InterestRate - 1

WHERE LoanID = rec.LoanID;

END LOOP;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error occurred: ' || SQLERRM);

ROLLBACK;

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.

**Code:**

ALTER TABLE Customers ADD IsVIP NUMBER(1);

BEGIN

UPDATE Customers

SET IsVIP = CASE WHEN balance > 10000 THEN 1 ELSE 0 END;

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.

**Code:**

SET SERVEROUTPUT ON;

DECLARE

CURSOR c\_loans\_due IS

SELECT c.name, l.loanid, l.enddate

FROM customers c

JOIN loans l ON c.customerid = l.customerid

WHERE l.enddate BETWEEN SYSDATE AND SYSDATE + 30;

BEGIN

FOR rec IN c\_loans\_due LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder for ' || rec.name || ': Loan ' || rec.loanid || ' is due on ' || TO\_CHAR(rec.enddate, 'YYYY-MM-DD'));

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.

**Code:**CREATE OR REPLACE PROCEDURE SafeTransferFunds (

from\_account\_id IN NUMBER,

to\_account\_id IN NUMBER,

amount IN NUMBER

)

IS

insufficient\_funds EXCEPTION;

funds\_balance NUMBER;

error\_msg VARCHAR2(4000);

BEGIN

-- Check balance of the from\_account with locking

SELECT balance INTO funds\_balance

FROM accounts

WHERE accountid = from\_account\_id

FOR UPDATE;

-- Raise exception if insufficient funds

IF funds\_balance < amount THEN

RAISE insufficient\_funds;

END IF;

-- Performing transfer

UPDATE accounts

SET balance = balance - amount

WHERE accountid = from\_account\_id;

UPDATE accounts

SET balance = balance + amount

WHERE accountid = to\_account\_id;

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

error\_msg := 'Insufficient funds for transfer';

BEGIN

INSERT INTO error\_logs (message, log\_time)

VALUES (error\_msg, SYSDATE);

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error logging insufficient funds: ' || SQLERRM);

END;

WHEN OTHERS THEN

error\_msg := SQLERRM;

BEGIN

INSERT INTO error\_logs (message, log\_time)

VALUES (error\_msg, SYSDATE);

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error logging general error: ' || SQLERRM);

END;

ROLLBACK;

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.

**Code:**

CREATE OR REPLACE PROCEDURE UpdateSalary (

employee\_id IN NUMBER,

percentage IN NUMBER

)

IS

employee\_not\_found EXCEPTION;

v\_count NUMBER;

error\_msg VARCHAR2(4000);

error\_code NUMBER;

BEGIN

-- Check if employee exists

SELECT COUNT(\*) INTO v\_count

FROM Employees

WHERE employeeid = employee\_id;

IF v\_count = 0 THEN

RAISE employee\_not\_found;

ELSE

-- Update the employee's salary

UPDATE Employees

SET salary = salary \* (1 + percentage / 100)

WHERE employeeid = employee\_id;

END IF;

COMMIT;

EXCEPTION

WHEN employee\_not\_found THEN

INSERT INTO error\_logs (log\_time, message)

VALUES (SYSDATE, 'Employee ID not found');

COMMIT;

WHEN OTHERS THEN

error\_msg := SQLERRM;

error\_code := SQLCODE;

INSERT INTO error\_logs (log\_time, message, error\_code)

VALUES (SYSDATE, error\_msg, error\_code);

COMMIT;

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.

**Code:**  
CREATE OR REPLACE PROCEDURE AddNewCustomer (

customer\_id IN NUMBER,

customer\_name IN VARCHAR2,

dob IN DATE,

balance IN NUMBER

)

IS

customer\_exists EXCEPTION;

v\_count NUMBER;

error\_msg VARCHAR2(4000);

error\_code NUMBER;

BEGIN

-- Check if customer already exists

SELECT COUNT(\*) INTO v\_count

FROM Customers

WHERE customerid = customer\_id;

IF v\_count > 0 THEN

RAISE customer\_exists;

ELSE

INSERT INTO Customers (customerid, name, dob, balance, lastmodified, isvip)

VALUES (customer\_id, customer\_name, dob, balance, SYSDATE, 0);

END IF;

COMMIT;

EXCEPTION

WHEN customer\_exists THEN

error\_msg := 'Customer ID already exists';

error\_code := SQLCODE;

INSERT INTO error\_logs (log\_time, message, error\_code)

VALUES (SYSDATE, error\_msg, error\_code);

WHEN OTHERS THEN

error\_msg := SQLERRM;

error\_code := SQLCODE;

INSERT INTO error\_logs (log\_time, message, error\_code)

VALUES (SYSDATE, error\_msg, error\_code);

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.

**Code:**  
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest

AS

update\_interest\_rate CONSTANT NUMBER := 0.01;

BEGIN

UPDATE Accounts

SET Balance = Balance \* (1 + update\_interest\_rate),

LastModified = SYSDATE

WHERE AccountType = 'Savings'

AND LastModified < TRUNC(SYSDATE);

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.

**Code:**  
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(dept VARCHAR2, bonus\_percent NUMBER)

AS

error\_msg VARCHAR2(4000);

BEGIN

IF bonus\_percent <= 0 THEN

INSERT INTO error\_logs (log\_time, message, error\_code)

VALUES (SYSTIMESTAMP, 'Bonus percentage must be greater than zero', -20001);

RETURN;

END IF;

BEGIN

UPDATE Employees

SET Salary = Salary \* (1 + bonus\_percent / 100)

WHERE Department = dept

AND Salary > 0;

IF SQL%ROWCOUNT = 0 THEN

INSERT INTO error\_logs (log\_time, message, error\_code)

VALUES (SYSTIMESTAMP, 'No employees found in department ' || dept, -20002);

END IF;

EXCEPTION

WHEN OTHERS THEN

error\_msg := 'An unexpected error occurred: ' || SQLERRM;

INSERT INTO error\_logs (log\_time, message, error\_code)

VALUES (SYSTIMESTAMP, error\_msg, -20003);

END;

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.

**Code:**  
CREATE OR REPLACE PROCEDURE TransferFunds(src\_acc\_id NUMBER, target\_acc\_id NUMBER, p\_amount NUMBER)

AS

src\_balance NUMBER;

BEGIN

-- Begin a transaction

BEGIN

-- Check source account balance

SELECT Balance INTO src\_balance

FROM Accounts

WHERE AccountID = src\_acc\_id;

IF src\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds');

END IF;

-- Update account balance

UPDATE Accounts

SET Balance = Balance - p\_amount,

LastModified = SYSDATE

WHERE AccountID = src\_acc\_id;

UPDATE Accounts

SET Balance = Balance + p\_amount,

LastModified = SYSDATE

WHERE AccountID = target\_acc\_id;

-- Insert transaction records

INSERT INTO Transactions (TRANSACTIONID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (trans\_id\_seq.NEXTVAL, src\_acc\_id, SYSDATE, -p\_amount, 'TransOut');

INSERT INTO Transactions (TRANSACTIONID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (trans\_id\_seq.NEXTVAL, target\_acc\_id, SYSDATE, p\_amount, 'TransIn');

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

RAISE;

END;

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.

**Code :**  
CREATE OR REPLACE FUNCTION CalculateAge (

cust\_id IN NUMBER

) RETURN NUMBER

IS

age NUMBER;

dob DATE;

BEGIN

SELECT DOB INTO dob FROM Customers WHERE CustomerID = cust\_id;

age := FLOOR(MONTHS\_BETWEEN(SYSDATE, dob) / 12);

RETURN age;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

WHEN OTHERS THEN

RETURN NULL;

END;

/

**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.

**Code:**  
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

loan\_amount IN NUMBER,

annual\_interest\_rate IN NUMBER,

loan\_duration\_yrs IN NUMBER

) RETURN NUMBER

IS

monthly\_interest\_rate NUMBER;

no\_of\_payments NUMBER;

monthly\_installment NUMBER;

BEGIN

IF loan\_amount <= 0 OR annual\_interest\_rate < 0 OR loan\_duration\_yrs <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Invalid input parameters');

END IF;

monthly\_interest\_rate := annual\_interest\_rate / 12 / 100;

no\_of\_payments := loan\_duration\_yrs \* 12;

IF monthly\_interest\_rate = 0 THEN

monthly\_installment := loan\_amount / no\_of\_payments;

ELSE

monthly\_installment := loan\_amount \* monthly\_interest\_rate /

(1 - POWER(1 + monthly\_interest\_rate, -no\_of\_payments));

END IF;

RETURN monthly\_installment;

EXCEPTION

WHEN OTHERS THEN

RAISE;

END;

/

**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.

**Code:**

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id NUMBER,

p\_amount NUMBER

) RETURN NUMBER

IS

v\_balance NUMBER;

BEGIN

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

IF v\_balance >= p\_amount THEN

RETURN 1; -- TRUE

ELSE

RETURN 0; -- FALSE

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0; -- FALSE

WHEN OTHERS THEN

RETURN 0; -- FALSE

END;

/

**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.

**Code:**  
CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

: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.

**Code:**  
CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, AccountID, TransactionDate, Amount, TransactionType, Action, LogTime)

VALUES (:NEW.TransactionID, :NEW.AccountID, :NEW.TransactionDate, :NEW.Amount, :NEW.TransactionType, 'INSERT', CURRENT\_TIMESTAMP);

END;

/

**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.

**Code:**  
CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

current\_balance NUMBER;

BEGIN

-- Check if the transaction is a withdrawal

IF :NEW.TransactionType = 'WITHDRAWAL' THEN

-- Retrieve the current balance of the account

SELECT Balance INTO current\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID

FOR UPDATE;

-- Ensure the withdrawal amount does not exceed the current balance

IF :NEW.Amount > current\_balance THEN

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

END IF;

ELSIF :NEW.TransactionType = 'DEPOSIT' THEN

-- Ensure the deposit amount is positive

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.

**Code:**SET SERVEROUTPUT ON;

DECLARE

CURSOR c\_transactions IS

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

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);

v\_customerID Customers.CustomerID%TYPE;

v\_name Customers.Name%TYPE;

v\_transactionDate Transactions.TransactionDate%TYPE;

v\_amount Transactions.Amount%TYPE;

v\_transactionType Transactions.TransactionType%TYPE;

BEGIN

OPEN c\_transactions;

LOOP

FETCH c\_transactions INTO v\_customerID, v\_name, v\_transactionDate, v\_amount, v\_transactionType;

EXIT WHEN c\_transactions%NOTFOUND;

-- Print the statement (for demonstration purposes, using DBMS\_OUTPUT)

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

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

DBMS\_OUTPUT.PUT\_LINE('Transaction Date: ' || TO\_CHAR(v\_transactionDate, 'YYYY-MM-DD'));

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

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

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

END LOOP;

CLOSE c\_transactions;

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.

**Code:**  
SET SERVEROUTPUT ON;

DECLARE

CURSOR accounts\_cursor IS

SELECT AccountID, Balance

FROM Accounts;

v\_acc\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

v\_annual\_fee NUMBER := 10; -- Example of annual fee

BEGIN

OPEN accounts\_cursor;

LOOP

FETCH accounts\_cursor INTO v\_acc\_id, v\_balance;

EXIT WHEN accounts\_cursor%NOTFOUND;

UPDATE Accounts

SET Balance = Balance - v\_annual\_fee

WHERE AccountID = v\_acc\_id;

END LOOP;

CLOSE accounts\_cursor;

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.

**Code:**  
SET SERVEROUTPUT ON;

DECLARE

CURSOR c\_loans IS

SELECT LoanID, InterestRate

FROM Loans

FOR UPDATE; -- clause for locking

v\_loanID Loans.LoanID%TYPE;

currentInterestRate Loans.InterestRate%TYPE;

newInterestRate Loans.InterestRate%TYPE;

-- Example function to calculate new interest rate

FUNCTION calculate\_new\_interest\_rate(p\_current\_rate NUMBER) RETURN NUMBER IS

BEGIN

RETURN p\_current\_rate + 0.01; -- Example increase interest rate by 1%

END;

BEGIN

OPEN c\_loans;

LOOP

FETCH c\_loans INTO v\_loanID, currentInterestRate;

EXIT WHEN c\_loans%NOTFOUND;

newInterestRate := calculate\_new\_interest\_rate(currentInterestRate);

UPDATE Loans

SET InterestRate = newInterestRate

WHERE CURRENT OF c\_loans; -- Use WHERE CURRENT OF for efficiency

-- Optional logging

DBMS\_OUTPUT.PUT\_LINE('Updated LoanID: ' || v\_loanID || ' to new interest rate: ' || newInterestRate);

END LOOP;

CLOSE c\_loans;

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.

**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.

**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.

**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'));*

**Code:**

**Scenario 1:**

--1. Create the Package Specification:

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddNewCustomer(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER);

PROCEDURE UpdateCustomerDetails(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_Balance NUMBER);

FUNCTION GetCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER;

END CustomerManagement;

/

--2. Create the Package Body:

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_DOB DATE, p\_Balance NUMBER) IS

BEGIN

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

VALUES (p\_CustomerID, p\_Name, p\_DOB, p\_Balance, SYSDATE);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID ' || p\_CustomerID || ' already exists.');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails(p\_CustomerID NUMBER, p\_Name VARCHAR2, p\_Balance NUMBER) IS

BEGIN

UPDATE Customers

SET Name = p\_Name, Balance = p\_Balance, LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID ' || p\_CustomerID || ' not found.');

END IF;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance

FROM Customers

WHERE CustomerID = p\_CustomerID;

RETURN v\_Balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Customer ID ' || p\_CustomerID || ' not found.');

RETURN NULL;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);

RETURN NULL;

END GetCustomerBalance;

END CustomerManagement;

/

**Scenario 2:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2, p\_hiredate DATE);

PROCEDURE UpdateEmployee(employee\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2);

FUNCTION CalculateAnnualSalary(employee\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2, p\_hiredate DATE) IS

BEGIN

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

VALUES (Employees\_Seq.NEXTVAL, p\_name, p\_position, p\_salary, p\_dept, p\_hiredate);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Employee ID already exists.');

WHEN OTHERS THEN

RAISE; -- Re-raise the exception for detailed handling

END;

PROCEDURE UpdateEmployee(employee\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2) IS

BEGIN

UPDATE Employees

SET Name = p\_name, Position = p\_position, Salary = p\_salary, Department = p\_dept

WHERE EmployeeID = employee\_id;

END;

FUNCTION CalculateAnnualSalary(employee\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary

FROM Employees

WHERE EmployeeID = employee\_id;

RETURN v\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN -1; -- Or handle the exception appropriately

END;

END EmployeeManagement;

/

**Scenario 3:**

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(cust\_id NUMBER, p\_acc\_type VARCHAR2) IS

BEGIN

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

VALUES (Accounts\_Seq.NEXTVAL, cust\_id, p\_acc\_type, 0, SYSDATE);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Account ID already exists.');

WHEN OTHERS THEN

RAISE;

END;

PROCEDURE CloseAccount(p\_acc\_id NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_acc\_id;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Account not found.');

WHEN OTHERS THEN

RAISE;

END;

FUNCTION GetCustomerTotalBalance(cust\_id NUMBER) RETURN NUMBER IS

v\_total\_balance NUMBER := 0;

BEGIN

SELECT SUM(Balance) INTO v\_total\_balance

FROM Accounts

WHERE CustomerID = cust\_id;

RETURN v\_total\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0; -- Or handle as needed

WHEN OTHERS THEN

RAISE;

END;

END AccountOperations;