Solutions for Week 2 Exercises : PL/SQL

**Exercise 1: Control Structures**

**Scenario 1: Apply a discount to loan interest rates for customers above 60 years old**

UPDATE customers

SET loan\_interest\_rate = loan\_interest\_rate - 1

WHERE age > 60;

**Scenario 2: Promote a customer to VIP status based on their balance**

UPDATE customers

SET IsVIP = TRUE

WHERE balance > 10000;

**Scenario 3: Send reminders to customers whose loans are due within the next 30 days**

SELECT customer\_id, loan\_due\_date

FROM loans

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

To print the reminder message for each customer (assuming using a simple script or command line tool):

SELECT 'Reminder: Customer ID ' || customer\_id || ' has a loan due on ' || TO\_CHAR(loan\_due\_date, 'YYYY-MM-DD')

FROM loans

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

**Exercise 2: Error Handling**

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

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

insufficient\_funds EXCEPTION;

BEGIN

-- Check if the from account has enough funds

DECLARE

v\_from\_balance NUMBER;

BEGIN

SELECT balance

INTO v\_from\_balance

FROM accounts

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

IF v\_from\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

ROLLBACK;

RETURN;

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;

DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully.');

EXCEPTION

WHEN insufficient\_funds THEN

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

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END SafeTransferFunds;

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

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) AS

BEGIN

UPDATE employees

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

WHERE employee\_id = p\_employee\_id;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Employee ID does not exist.');

ROLLBACK;

ELSE

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully.');

END IF;

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END UpdateSalary;

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

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_age IN NUMBER,

p\_balance IN NUMBER

) AS

BEGIN

INSERT INTO customers (customer\_id, name, age, balance)

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

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Customer added successfully.');

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END AddNewCustomer;

**Exercise 3: Stored Procedures**

**Scenario 1: Process monthly interest for all savings accounts**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

UPDATE accounts

SET balance = balance \* 1.01

WHERE account\_type = 'SAVINGS';

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest processed for all savings accounts.');

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END ProcessMonthlyInterest;

**Scenario 2: Implement a bonus scheme for employees based on their performance**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department\_id IN NUMBER,

p\_bonus\_percentage IN NUMBER

) AS

BEGIN

UPDATE employees

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

WHERE department\_id = p\_department\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Employee bonuses updated for department ' || p\_department\_id);

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END UpdateEmployeeBonus;

**Scenario 3: Transfer funds between customer accounts**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

insufficient\_funds EXCEPTION;

BEGIN

-- Check if the from account has enough funds

DECLARE

v\_from\_balance NUMBER;

BEGIN

SELECT balance

INTO v\_from\_balance

FROM accounts

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

FOR UPDATE;

IF v\_from\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

ROLLBACK;

RETURN;

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;

DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully.');

EXCEPTION

WHEN insufficient\_funds THEN

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

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END TransferFunds;

**Exercise 4: Functions**

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

CREATE OR REPLACE FUNCTION CalculateAge (

p\_date\_of\_birth DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

SELECT FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_date\_of\_birth) / 12)

INTO v\_age

FROM dual;

RETURN v\_age;

EXCEPTION

WHEN OTHERS THEN

RETURN NULL;

END CalculateAge;

**Scenario 2: Compute the monthly installment for a loan**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount NUMBER,

p\_annual\_interest\_rate NUMBER,

p\_loan\_duration\_years NUMBER

) RETURN NUMBER IS

v\_monthly\_interest\_rate NUMBER;

v\_number\_of\_payments NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

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

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

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;

EXCEPTION

WHEN OTHERS THEN

RETURN NULL;

END CalculateMonthlyInstallment;

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

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 account\_id = p\_account\_id;

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**

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**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON transactions

FOR EACH ROW

BEGIN

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

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

END LogTransaction;

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

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

-- Check if the transaction is a withdrawal

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

-- Get the current balance of the account

SELECT balance

INTO v\_balance

FROM accounts

WHERE account\_id = :NEW.account\_id

FOR UPDATE;

-- Ensure the withdrawal does not exceed the balance

IF :NEW.amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance for withdrawal.');

END IF;

END IF;

-- Ensure deposits are positive

IF :NEW.transaction\_type = 'DEPOSIT' AND :NEW.amount <= 0 THEN

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

END IF;

END CheckTransactionRules;

**Exercise 6: Cursors**

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

DECLARE

CURSOR transactions\_cursor IS

SELECT customer\_id, transaction\_date, amount, transaction\_type

FROM transactions

WHERE EXTRACT(MONTH FROM transaction\_date) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM transaction\_date) = EXTRACT(YEAR FROM SYSDATE);

v\_customer\_id transactions.customer\_id%TYPE;

v\_transaction\_date transactions.transaction\_date%TYPE;

v\_amount transactions.amount%TYPE;

v\_transaction\_type transactions.transaction\_type%TYPE;

BEGIN

OPEN transactions\_cursor;

LOOP

FETCH transactions\_cursor INTO v\_customer\_id, v\_transaction\_date, v\_amount, v\_transaction\_type;

EXIT WHEN transactions\_cursor%NOTFOUND;

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

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

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

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

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

END LOOP;

CLOSE transactions\_cursor;

END;

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

DECLARE

CURSOR accounts\_cursor 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; -- Annual fee amount

BEGIN

OPEN accounts\_cursor;

LOOP

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

EXIT WHEN accounts\_cursor%NOTFOUND;

-- Deduct the annual fee

UPDATE accounts

SET balance = balance - v\_annual\_fee

WHERE account\_id = v\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Annual fee applied to account ID: ' || v\_account\_id);

END LOOP;

CLOSE accounts\_cursor;

COMMIT;

END;

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

DECLARE

CURSOR loans\_cursor 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 CONSTANT NUMBER := 5; -- New interest rate

BEGIN

OPEN loans\_cursor;

LOOP

FETCH loans\_cursor INTO v\_loan\_id, v\_interest\_rate;

EXIT WHEN loans\_cursor%NOTFOUND;

-- Update the interest rate

UPDATE loans

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

WHERE loan\_id = v\_loan\_id;

DBMS\_OUTPUT.PUT\_LINE('Interest rate updated for loan ID: ' || v\_loan\_id);

END LOOP;

CLOSE loans\_cursor;

COMMIT;

END;

**Exercise 7: Packages**

**Scenario 1: CustomerManagement Package**

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

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

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) IS

BEGIN

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

VALUES (p\_customer\_id, p\_name, p\_dob, p\_balance, SYSDATE);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Customer with this ID already exists.');

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails(p\_customer\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER) IS

BEGIN

UPDATE Customers

SET Name = p\_name, DOB = p\_dob, Balance = p\_balance, LastModified = SYSDATE

WHERE CustomerID = p\_customer\_id;

IF SQL%NOTFOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Customer not found.');

END IF;

END UpdateCustomerDetails;

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;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END GetCustomerBalance;

END CustomerManagement;

/

**Scenario 2: EmployeeManagement Package**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(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\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2);

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

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(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);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Employee with this ID already exists.');

END HireEmployee;

PROCEDURE UpdateEmployeeDetails(p\_employee\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2) IS

BEGIN

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

IF SQL%NOTFOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Employee not found.');

END IF;

END UpdateEmployeeDetails;

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

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary

FROM Employees

WHERE EmployeeID = p\_employee\_id;

RETURN v\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END CalculateAnnualSalary;

END EmployeeManagement;

/

**Scenario 3: AccountOperations Package**

CREATE OR REPLACE PACKAGE AccountOperations AS

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

PROCEDURE CloseAccount(p\_account\_id NUMBER);

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

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

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

BEGIN

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

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Account with this ID already exists.');

END OpenAccount;

PROCEDURE CloseAccount(p\_account\_id NUMBER) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_account\_id;

IF SQL%NOTFOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Account not found.');

END IF;

END CloseAccount;

FUNCTION GetTotalBalance(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;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END GetTotalBalance;

END AccountOperations;

/