**WEEK 2 Hands On**

**PL SQL**

**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

v\_customer\_id Customers.CustomerID%TYPE;

v\_dob Customers.DOB%TYPE;

v\_age NUMBER;

BEGIN

FOR rec IN (SELECT CustomerID, DOB FROM Customers) LOOP

v\_customer\_id := rec.CustomerID;

v\_dob := rec.DOB;

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, v\_dob) / 12);

IF v\_age > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = v\_customer\_id;

END IF;

END LOOP;

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.

**CODE**

ALTER TABLE Customers ADD IsVIP VARCHAR2(5);

BEGIN

FOR rec IN (SELECT CustomerID, Balance FROM Customers) LOOP

IF rec.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = rec.CustomerID;

ELSE

UPDATE Customers

SET IsVIP = 'FALSE'

WHERE CustomerID = rec.CustomerID;

END IF;

END LOOP;

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**

DECLARE

v\_name Customers.Name%TYPE;

v\_due\_date Loans.EndDate%TYPE;

BEGIN

FOR rec IN (

SELECT c.Name, l.EndDate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

v\_name := rec.Name;

v\_due\_date := rec.EndDate;

DBMS\_OUTPUT.PUT\_LINE('Reminder: Dear ' || v\_name ||

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

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(

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

v\_balance Accounts.Balance%TYPE;

BEGIN

-- Check balance of from\_account

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id;

IF v\_balance < p\_amount THEN

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

END IF;

-- Debit from source account

UPDATE Accounts

SET Balance = Balance - p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_from\_account\_id;

-- Credit to destination account

UPDATE Accounts

SET Balance = Balance + p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_to\_account\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error in SafeTransferFunds: ' || 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.

**CODE**

CREATE OR REPLACE PROCEDURE UpdateSalary(

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) AS

v\_salary Employees.Salary%TYPE;

BEGIN

-- Try updating the salary

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

-- Check if any row was updated

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error in UpdateSalary: ' || 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.

**CODE**

CREATE OR REPLACE PROCEDURE AddNewCustomer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) AS

BEGIN

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Unexpected error in AddNewCustomer: ' || 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.

**CODE**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01),

LastModified = SYSDATE

WHERE AccountType = 'Savings';

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.

**CODE**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_department IN VARCHAR2,

p\_bonus\_percentage IN NUMBER

) AS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* 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.

**CODE**

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

v\_balance Accounts.Balance%TYPE;

v\_customer\_from Accounts.CustomerID%TYPE;

v\_customer\_to Accounts.CustomerID%TYPE;

BEGIN

-- Validate same customer owns both accounts

SELECT CustomerID INTO v\_customer\_from FROM Accounts WHERE AccountID = p\_from\_account;

SELECT CustomerID INTO v\_customer\_to FROM Accounts WHERE AccountID = p\_to\_account;

IF v\_customer\_from != v\_customer\_to THEN

RAISE\_APPLICATION\_ERROR(-20010, 'Both accounts must belong to the same customer.');

END IF;

-- Check for sufficient funds

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

IF v\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20011, 'Insufficient balance in source account.');

END IF;

-- Perform transfer

UPDATE Accounts

SET Balance = Balance - p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_from\_account;

UPDATE Accounts

SET Balance = Balance + p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_to\_account;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

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(

p\_dob IN DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

RETURN v\_age;

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(

p\_loan\_amount IN NUMBER,

p\_interest\_rate IN NUMBER, -- Annual interest rate

p\_duration\_years IN NUMBER

) RETURN NUMBER IS

r NUMBER; -- monthly interest rate

n NUMBER; -- total months

emi NUMBER;

BEGIN

r := p\_interest\_rate / 12 / 100;

n := p\_duration\_years \* 12;

IF r = 0 THEN

emi := p\_loan\_amount / n; -- No interest case

ELSE

emi := (p\_loan\_amount \* r \* POWER(1 + r, n)) / (POWER(1 + r, n) - 1);

END IF;

RETURN ROUND(emi, 2);

END;

/

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

**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 IN NUMBER,

p\_amount IN NUMBER

) RETURN BOOLEAN IS

v\_balance Accounts.Balance%TYPE;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_account\_id;

RETURN v\_balance >= p\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

WHEN OTHERS THEN

RETURN 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;

/

**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, Action, ActionDate)

VALUES (:NEW.TransactionID, :NEW.AccountID, 'INSERT', SYSDATE);

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

v\_balance Accounts.Balance%TYPE;

BEGIN

-- Check positive deposit

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

RAISE\_APPLICATION\_ERROR(-20020, 'Deposit amount must be positive.');

END IF;

-- Check withdrawal doesn't exceed balance

IF :NEW.TransactionType = 'Withdrawal' THEN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

IF :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20021, 'Withdrawal exceeds available balance.');

END IF;

END IF;

END;

/

**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**

DECLARE

CURSOR trans\_cursor IS

SELECT 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 TRUNC(t.TransactionDate, 'MM') = TRUNC(SYSDATE, 'MM')

ORDER BY c.CustomerID, t.TransactionDate;

v\_customer\_id Customers.CustomerID%TYPE;

v\_name Customers.Name%TYPE;

v\_date Transactions.TransactionDate%TYPE;

v\_amount Transactions.Amount%TYPE;

v\_type Transactions.TransactionType%TYPE;

BEGIN

OPEN trans\_cursor;

LOOP

FETCH trans\_cursor INTO v\_customer\_id, v\_name, v\_date, v\_amount, v\_type;

EXIT WHEN trans\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || v\_name ||

' | Date: ' || TO\_CHAR(v\_date, 'DD-Mon-YYYY') ||

' | ' || v\_type || ': $' || v\_amount);

END LOOP;

CLOSE trans\_cursor;

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**

DECLARE

CURSOR account\_cursor IS

SELECT AccountID, Balance FROM Accounts;

v\_account\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

v\_fee CONSTANT NUMBER := 100; -- example annual fee

BEGIN

OPEN account\_cursor;

LOOP

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

EXIT WHEN account\_cursor%NOTFOUND;

UPDATE Accounts

SET Balance = Balance - v\_fee,

LastModified = SYSDATE

WHERE AccountID = v\_account\_id;

END LOOP;

CLOSE account\_cursor;

COMMIT;

END;

/

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

**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**

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, LoanAmount FROM Loans;

v\_loan\_id Loans.LoanID%TYPE;

v\_loan\_amount Loans.LoanAmount%TYPE;

v\_new\_rate NUMBER;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_loan\_id, v\_loan\_amount;

EXIT WHEN loan\_cursor%NOTFOUND;

IF v\_loan\_amount < 5000 THEN

v\_new\_rate := 4;

ELSIF v\_loan\_amount <= 10000 THEN

v\_new\_rate := 5;

ELSE

v\_new\_rate := 6;

END IF;

UPDATE Loans

SET InterestRate = v\_new\_rate

WHERE LoanID = v\_loan\_id;

END LOOP;

CLOSE loan\_cursor;

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.

**CODE**

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

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

BEGIN

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

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END;

PROCEDURE UpdateCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_balance NUMBER) IS

BEGIN

UPDATE Customers

SET Name = p\_name,

Balance = p\_balance,

LastModified = SYSDATE

WHERE CustomerID = p\_id;

END;

FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Customers

WHERE CustomerID = p\_id;

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

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.

**CODE**

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

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

END;

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

BEGIN

UPDATE Employees

SET Salary = p\_salary,

Position = p\_position

WHERE EmployeeID = p\_id;

END;

FUNCTION GetAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary FROM Employees WHERE EmployeeID = p\_id;

RETURN v\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

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.

**CODE**

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

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

BEGIN

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

VALUES(p\_acc\_id, p\_cust\_id, p\_type, p\_balance, SYSDATE);

END;

PROCEDURE CloseAccount(p\_acc\_id NUMBER) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_acc\_id;

END;

FUNCTION GetTotalBalance(p\_cust\_id NUMBER) RETURN NUMBER IS

v\_total NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_total

FROM Accounts

WHERE CustomerID = p\_cust\_id;

RETURN NVL(v\_total, 0);

END;

END AccountOperations;

/