## Exercise 1: Control Structures

### Scenario 1: Apply Discount to Loan Interest Rates for Customers Above 60 Years Old

DECLARE

CURSOR cur\_customers IS

SELECT l.LoanID, l.InterestRate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM c.DOB) > 60;

BEGIN

FOR customer\_rec IN cur\_customers LOOP

UPDATE Loans

SET InterestRate = customer\_rec.InterestRate - 1

WHERE LoanID = customer\_rec.LoanID;

END LOOP;

COMMIT;

END;

/

### Scenario 2: Promote Customers to VIP Status Based on Balance

DECLARE

CURSOR cur\_customers IS

SELECT CustomerID

FROM Customers

WHERE Balance > 10000;

BEGIN

FOR customer\_rec IN cur\_customers LOOP

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = customer\_rec.CustomerID;

END LOOP;

COMMIT;

END;

/

Scenario 3: Send Reminders to Customers Whose Loans are Due within the Next 30 Days:

DECLARE

CURSOR cur\_loans IS

SELECT l.LoanID, c.Name, l.EndDate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30;

BEGIN

FOR loan\_rec IN cur\_loans LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || loan\_rec.LoanID || ' for customer ' || loan\_rec.Name || ' is due on ' || TO\_CHAR(loan\_rec.EndDate, 'YYYY-MM-DD'));

END LOOP;

END;

/

## Exercise 2: Error Handling

### Scenario 1: Safe Transfer Funds

### CREATE OR REPLACE PROCEDURE SafeTransferFunds (

### p\_source\_account\_id IN Accounts.AccountID%TYPE,

### p\_target\_account\_id IN Accounts.AccountID%TYPE,

### p\_amount IN NUMBER

### ) IS

### insufficient\_funds EXCEPTION;

### BEGIN

### UPDATE Accounts

### SET Balance = Balance - p\_amount

### WHERE AccountID = p\_source\_account\_id;

### IF SQL%NOTFOUND OR SQL%ROWCOUNT = 0 THEN

### RAISE insufficient\_funds;

### END IF;

### UPDATE Accounts

### SET Balance = Balance + p\_amount

### WHERE AccountID = p\_target\_account\_id;

### IF SQL%NOTFOUND OR SQL%ROWCOUNT = 0 THEN

### RAISE insufficient\_funds;

### END IF;

### COMMIT;

### EXCEPTION

### WHEN insufficient\_funds THEN

### ROLLBACK;

### DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds for transfer.');

### WHEN OTHERS THEN

### ROLLBACK;

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

### END;

### /

### Scenario 2: Update Salary

### CREATE OR REPLACE PROCEDURE UpdateSalary (

### p\_employee\_id IN Employees.EmployeeID%TYPE,

### p\_percentage IN NUMBER

### ) IS

### employee\_not\_found EXCEPTION;

### BEGIN

### UPDATE Employees

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

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

### IF SQL%NOTFOUND OR SQL%ROWCOUNT = 0 THEN

### RAISE employee\_not\_found;

### END IF;

### COMMIT;

### EXCEPTION

### WHEN employee\_not\_found THEN

### DBMS\_OUTPUT.PUT\_LINE('Error: Employee with ID ' || p\_employee\_id || ' does not exist.');

### WHEN OTHERS THEN

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

### END;

### /

### Scenario 3: Add New Customer

### CREATE OR REPLACE PROCEDURE AddNewCustomer (

### p\_customer\_id IN Customers.CustomerID%TYPE,

### p\_name IN Customers.Name%TYPE,

### p\_dob IN Customers.DOB%TYPE,

### p\_balance IN Customers.Balance%TYPE

### ) IS

### duplicate\_customer EXCEPTION;

### 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

### RAISE duplicate\_customer;

### WHEN duplicate\_customer THEN

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

### WHEN OTHERS THEN

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

### END;

### /

## Exercise 3: Stored Procedures

### Scenario 1: Process Monthly Interest

### CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

### BEGIN

### UPDATE Accounts

### SET Balance = Balance \* 1.01

### WHERE AccountType = 'Savings';

### COMMIT;

### END;

### /

### Scenario 2: Update Employee Bonus

### CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

### p\_department IN Employees.Department%TYPE,

### p\_bonus\_percentage IN NUMBER

### ) IS

### BEGIN

### UPDATE Employees

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

### WHERE Department = p\_department;

### 

### COMMIT;

### END;

### /

### Scenario 3: Transfer Funds Between Accounts

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_source\_account\_id IN Accounts.AccountID%TYPE,

p\_target\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) IS

insufficient\_balance EXCEPTION;

BEGIN

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_source\_account\_id;

IF SQL%NOTFOUND OR SQL%ROWCOUNT = 0 THEN

RAISE insufficient\_balance;

END IF;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_target\_account\_id;

IF SQL%NOTFOUND OR SQL%ROWCOUNT = 0 THEN

RAISE insufficient\_balance;

END IF;

COMMIT;

EXCEPTION

WHEN insufficient\_balance THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient balance for transfer.');

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

## Exercise 4: Functions

### Scenario 1: Calculate Age CREATE OR REPLACE FUNCTION CalculateAge (

### p\_dob IN DATE

### ) RETURN NUMBER IS

### l\_age NUMBER;

### BEGIN

### l\_age := TRUNC((SYSDATE - p\_dob) / 365.25);

### RETURN l\_age;

### END;

### /

### Scenario 2: Calculate Monthly Installment

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount IN NUMBER,

p\_interest\_rate IN NUMBER,

p\_loan\_duration\_years IN NUMBER

) RETURN NUMBER IS

l\_monthly\_installment NUMBER;

BEGIN

l\_monthly\_installment := (p\_loan\_amount \* (1 + (p\_interest\_rate / 100) \* p\_loan\_duration\_years)) / (p\_loan\_duration\_years \* 12);

RETURN l\_monthly\_installment;

END;

/

### Scenario 3: Has Sufficient Balance

### CREATE OR REPLACE FUNCTION HasSufficientBalance (

### p\_account\_id IN Accounts.AccountID%TYPE,

### p\_amount IN NUMBER

### ) RETURN BOOLEAN IS

### l\_balance NUMBER;

### BEGIN

### SELECT Balance

### INTO l\_balance

### FROM Accounts

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

### 

### RETURN l\_balance >= p\_amount;

### EXCEPTION

### WHEN NO\_DATA\_FOUND THEN

### RETURN FALSE;

### END;

### /

## Exercise 5: Triggers

### Scenario 1: Update Last Modified Date

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

### Scenario 2: Maintain Audit Log

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (AuditID, TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (AuditLog\_seq.NEXTVAL, :NEW.TransactionID, :NEW.AccountID, :NEW.TransactionDate, :NEW.Amount, :NEW.TransactionType);

END;

/

### Scenario 3: Enforce Transaction Rules

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

BEGIN

IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > (SELECT Balance FROM Accounts WHERE AccountID = :NEW.AccountID) THEN

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

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

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

END IF;

END;

/

## Exercise 6: Cursors

### Scenario 1: Generate Monthly Statements

DECLARE

CURSOR cur\_transactions IS

SELECT t.AccountID, c.Name, t.TransactionDate, t.Amount, t.TransactionType

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

JOIN Customers c ON a.CustomerID = c.CustomerID

WHERE t.TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE);

BEGIN

FOR trans\_rec IN cur\_transactions LOOP

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || trans\_rec.Name);

DBMS\_OUTPUT.PUT\_LINE('AccountID: ' || trans\_rec.AccountID);

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

DBMS\_OUTPUT.PUT\_LINE('Amount: ' || trans\_rec.Amount);

DBMS\_OUTPUT.PUT\_LINE('Transaction Type: ' || trans\_rec.TransactionType);

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

END LOOP;

END;

/

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

DECLARE

CURSOR cur\_accounts IS

SELECT AccountID, Balance

FROM Accounts;

annual\_fee CONSTANT NUMBER := 100; -- Define the annual fee

BEGIN

FOR account\_rec IN cur\_accounts LOOP

UPDATE Accounts

SET Balance = Balance - annual\_fee

WHERE AccountID = account\_rec.AccountID;

END LOOP;

COMMIT;

END;

/

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

DECLARE

CURSOR cur\_loans IS

SELECT LoanID, InterestRate

FROM Loans;

new\_interest\_rate CONSTANT NUMBER := 4.5; -- Define the new interest rate

BEGIN

FOR loan\_rec IN cur\_loans LOOP

UPDATE Loans

SET InterestRate = new\_interest\_rate

WHERE LoanID = loan\_rec.LoanID;

END LOOP;

COMMIT;

END;

/

### Exercise 7: Packages

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

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

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

FUNCTION GetCustomerBalance(p\_CustomerID NUMBER) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(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);

END AddCustomer;

PROCEDURE UpdateCustomer(p\_CustomerID 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\_CustomerID;

END UpdateCustomer;

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;

END GetCustomerBalance;

END CustomerManagement;

/

#### Scenario 2: Create a package to manage employee data

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_EmployeeID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2, p\_HireDate DATE);

PROCEDURE UpdateEmployee(p\_EmployeeID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2);

FUNCTION CalculateAnnualSalary(p\_EmployeeID NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_EmployeeID NUMBER, p\_Name VARCHAR2, p\_Position VARCHAR2, p\_Salary NUMBER, p\_Department VARCHAR2, p\_HireDate DATE) IS

BEGIN

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

VALUES (p\_EmployeeID, p\_Name, p\_Position, p\_Salary, p\_Department, p\_HireDate);

END HireEmployee;

PROCEDURE UpdateEmployee(p\_EmployeeID 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\_EmployeeID;

END UpdateEmployee;

FUNCTION CalculateAnnualSalary(p\_EmployeeID NUMBER) RETURN NUMBER IS

v\_Salary NUMBER;

BEGIN

SELECT Salary \* 12 INTO v\_Salary

FROM Employees

WHERE EmployeeID = p\_EmployeeID;

RETURN v\_Salary;

END CalculateAnnualSalary;

END EmployeeManagement;

/

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

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_AccountID NUMBER, p\_CustomerID NUMBER, p\_AccountType VARCHAR2, p\_Balance NUMBER);

PROCEDURE CloseAccount(p\_AccountID NUMBER);

FUNCTION GetTotalBalance(p\_CustomerID NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_AccountID NUMBER, p\_CustomerID NUMBER, p\_AccountType VARCHAR2, p\_Balance NUMBER) IS

BEGIN

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

VALUES (p\_AccountID, p\_CustomerID, p\_AccountType, p\_Balance, SYSDATE);

END OpenAccount;

PROCEDURE CloseAccount(p\_AccountID NUMBER) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_AccountID;

END CloseAccount;

FUNCTION GetTotalBalance(p\_CustomerID NUMBER) RETURN NUMBER IS

v\_TotalBalance NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_TotalBalance

FROM Accounts

WHERE CustomerID = p\_CustomerID;

RETURN v\_TotalBalance;

END GetTotalBalance;

END AccountOperations;

/