Exercise 1: Control Structures

Scenario 1:

BEGIN

FOR r IN (

SELECT l.LoanID, l.InterestRate, c.DOB

FROM Loans l JOIN Customers c ON l.CustomerID = c.CustomerID

) LOOP

IF MONTHS\_BETWEEN(SYSDATE, r.DOB) / 12 > 60 THEN

UPDATE Loans SET InterestRate = InterestRate - 1 WHERE LoanID = r.LoanID;

END IF;

END LOOP;

END;

/

**Output:** Loan interest rates updated for eligible customers.

Scenario 2:

ALTER TABLE Customers ADD IsVIP VARCHAR2(5);

BEGIN

FOR r IN (SELECT CustomerID FROM Customers WHERE Balance > 10000) LOOP

UPDATE Customers SET IsVIP = 'TRUE' WHERE CustomerID = r.CustomerID;

END LOOP;

END;

/

**Output:** IsVIP set to TRUE for qualifying customers.

Scenario 3:

BEGIN

FOR r IN (

SELECT l.LoanID, c.Name

FROM Loans l JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate <= SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || r.LoanID || ' for ' || r.Name || ' is due within 30 days.');

END LOOP;

END;

/

**Output:** Reminder messages printed for due loans.

Exercise 2: Error Handling

Scenario 1:

CREATE OR REPLACE PROCEDURE SafeTransferFunds(p\_FromAcc NUMBER, p\_ToAcc NUMBER, p\_Amount NUMBER) IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance FROM Accounts WHERE AccountID = p\_FromAcc FOR UPDATE;

IF v\_Balance < p\_Amount THEN

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

END IF;

UPDATE Accounts SET Balance = Balance - p\_Amount WHERE AccountID = p\_FromAcc;

UPDATE Accounts SET Balance = Balance + p\_Amount WHERE AccountID = p\_ToAcc;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

VALUES (Transactions\_seq.NEXTVAL, p\_FromAcc, SYSDATE, p\_Amount, 'Failed');

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

END;

/

**Output:** Success or failure message; transaction rollback on error.

Scenario 2:

CREATE OR REPLACE PROCEDURE UpdateSalary(p\_EmpID NUMBER, p\_Percent NUMBER) IS

BEGIN

UPDATE Employees SET Salary = Salary + (Salary \* p\_Percent / 100) WHERE EmployeeID = p\_EmpID;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END;

/

**Output:** Error logged if employee not found.

Scenario 3:

CREATE OR REPLACE PROCEDURE AddNewCustomer(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);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

ROLLBACK;

END;

/

**Output:** Error logged if duplicate ID.

Exercise 3: Stored Procedures

Scenario 1:

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'Savings';

COMMIT;

END;

/

**Output:** Updated balances for all savings accounts.

Scenario 2:

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(p\_Dept VARCHAR2, p\_Bonus NUMBER) IS

BEGIN

UPDATE Employees

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

WHERE Department = p\_Dept;

COMMIT;

END;

/

Scenario 3:

CREATE OR REPLACE PROCEDURE TransferFunds(p\_From NUMBER, p\_To NUMBER, p\_Amount NUMBER) IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance FROM Accounts WHERE AccountID = p\_From FOR UPDATE;

IF v\_Balance < p\_Amount THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Insufficient balance');

END IF;

UPDATE Accounts SET Balance = Balance - p\_Amount WHERE AccountID = p\_From;

UPDATE Accounts SET Balance = Balance + p\_Amount WHERE AccountID = p\_To;

COMMIT;

END;

/

**Output:** Successful transfer or error on insufficient balance.

Exercise 4: Functions

Scenario 1:

CREATE OR REPLACE FUNCTION CalculateAge(p\_DOB DATE) RETURN NUMBER IS

BEGIN

RETURN TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_DOB)/12);

END;

/

**Output:** Returns age in years.

Scenario 2:

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(p\_Amount NUMBER, p\_Rate NUMBER, p\_Years NUMBER) RETURN NUMBER IS

v\_Months NUMBER := p\_Years \* 12;

v\_Rate NUMBER := p\_Rate / (12 \* 100);

BEGIN

RETURN (p\_Amount \* v\_Rate) / (1 - POWER(1 + v\_Rate, -v\_Months));

END;

/

**Output:** Returns EMI value.

Scenario 3:

CREATE OR REPLACE FUNCTION HasSufficientBalance(p\_AccID NUMBER, p\_Amount NUMBER) RETURN BOOLEAN IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance FROM Accounts WHERE AccountID = p\_AccID;

RETURN v\_Balance >= p\_Amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END;

/

**Output:** Returns TRUE or FALSE.

Exercise 5: Triggers

Scenario 1:

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

**Output:** LastModified auto-updated.

Scenario 2:

CREATE TABLE AuditLog (

LogID NUMBER GENERATED ALWAYS AS IDENTITY,

AccountID NUMBER,

Action VARCHAR2(20),

Timestamp DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog(AccountID, Action, Timestamp)

VALUES (:NEW.AccountID, 'Transaction', SYSDATE);

END;

/

**Output:** Audit log entries created.

Scenario 3:

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance FROM Accounts WHERE AccountID = :NEW.AccountID;

IF :NEW.TransactionType = 'Withdrawal' AND :NEW.Amount > v\_Balance THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Insufficient funds');

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

RAISE\_APPLICATION\_ERROR(-20005, 'Invalid deposit amount');

END IF;

END;

/

**Output:** Errors raised for invalid transactions.

Exercise 6: Cursors

Scenario 1:

DECLARE

CURSOR cur IS

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

FROM Customers c JOIN Accounts a ON c.CustomerID = a.CustomerID

JOIN Transactions t ON a.AccountID = t.AccountID

WHERE TO\_CHAR(t.TransactionDate, 'MM-YYYY') = TO\_CHAR(SYSDATE, 'MM-YYYY');

BEGIN

FOR r IN cur LOOP

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || r.Name || ', Amount: ' || r.Amount || ', Date: ' || r.TransactionDate);

END LOOP;

END;

/

**Output:** Monthly statement lines printed.

Scenario 2:

DECLARE

CURSOR cur IS SELECT AccountID FROM Accounts;

BEGIN

FOR r IN cur LOOP

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = r.AccountID;

END LOOP;

COMMIT;

END;

/

**Output:** $100 fee deducted from each account.

Scenario 3:

DECLARE

CURSOR cur IS SELECT LoanID, InterestRate FROM Loans;

BEGIN

FOR r IN cur LOOP

UPDATE Loans SET InterestRate = r.InterestRate + 0.5 WHERE LoanID = r.LoanID;

END LOOP;

COMMIT;

END;

/

**Output:** Interest rate updated per policy.

Exercise 7: Packages

Scenario 1:

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

PROCEDURE UpdateCustomer(p\_ID NUMBER, p\_Name VARCHAR2);

FUNCTION GetBalance(p\_ID NUMBER) RETURN NUMBER;

END CustomerManagement;

/

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

END;

PROCEDURE UpdateCustomer(p\_ID NUMBER, p\_Name VARCHAR2) IS

BEGIN

UPDATE Customers SET Name = p\_Name WHERE CustomerID = p\_ID;

END;

FUNCTION GetBalance(p\_ID NUMBER) RETURN NUMBER IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance FROM Customers WHERE CustomerID = p\_ID;

RETURN v\_Balance;

END;

END CustomerManagement;

/

**Output:** Reusable customer procedures and function.

Scenario 2:

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_ID NUMBER, p\_Name VARCHAR2, p\_Pos VARCHAR2, p\_Salary NUMBER, p\_Dept VARCHAR2);

PROCEDURE UpdateEmployee(p\_ID NUMBER, p\_Salary NUMBER);

FUNCTION AnnualSalary(p\_ID NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_ID NUMBER, p\_Name VARCHAR2, p\_Pos VARCHAR2, p\_Salary NUMBER, p\_Dept VARCHAR2) IS

BEGIN

INSERT INTO Employees VALUES(p\_ID, p\_Name, p\_Pos, p\_Salary, p\_Dept, SYSDATE);

END;

PROCEDURE UpdateEmployee(p\_ID NUMBER, p\_Salary NUMBER) IS

BEGIN

UPDATE Employees SET Salary = p\_Salary WHERE EmployeeID = p\_ID;

END;

FUNCTION AnnualSalary(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;

END;

END EmployeeManagement;

/

**Output:** Complete employee management module.

Scenario 3:

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_ID NUMBER, p\_CustID NUMBER, p\_Type VARCHAR2, p\_Balance NUMBER);

PROCEDURE CloseAccount(p\_ID NUMBER);

FUNCTION TotalCustomerBalance(p\_CustID NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_ID NUMBER, p\_CustID NUMBER, p\_Type VARCHAR2, p\_Balance NUMBER) IS

BEGIN

INSERT INTO Accounts VALUES(p\_ID, p\_CustID, p\_Type, p\_Balance, SYSDATE);

END;

PROCEDURE CloseAccount(p\_ID NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_ID;

END;

FUNCTION TotalCustomerBalance(p\_CustID NUMBER) RETURN NUMBER IS

v\_Total NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_Total FROM Accounts WHERE CustomerID = p\_CustID;

RETURN v\_Total;

END;

END AccountOperations;

/

**Output:** Modular account operations.