**Exercise 1: Control Structures**

1. **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.

BEGIN

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

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

UPDATE Loans

SET InterestRate = InterestRate \* 0.99

WHERE CustomerID = customer.CustomerID;

END IF;

END LOOP;

COMMIT;

END;

/

1. **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.

BEGIN

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

IF customer.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = customer.CustomerID;

END IF;

END LOOP;

COMMIT;

END;

/

1. **Question:** Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, CustomerID, EndDate

FROM Loans

WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30;

customer\_name VARCHAR2(100);

reminder\_message VARCHAR2(200);

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO loan\_cursor.LoanID, loan\_cursor.CustomerID, loan\_cursor.EndDate;

EXIT WHEN loan\_cursor%NOTFOUND;

SELECT Name INTO customer\_name FROM Customers WHERE CustomerID = loan\_cursor.CustomerID;

reminder\_message := 'Reminder: Dear ' || customer\_name ||

', your loan (Loan ID: ' || loan\_cursor.LoanID ||

') is due on ' || TO\_CHAR(loan\_cursor.EndDate, 'YYYY-MM-DD') || '.';

DBMS\_OUTPUT.PUT\_LINE(reminder\_message);

END LOOP;

CLOSE loan\_cursor;

END;

/

**Exercise 2: Error Handling**

1. 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.

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

)

IS

insufficient\_funds EXCEPTION;

v\_from\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id

FOR UPDATE;

IF v\_from\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

UPDATE Accounts

SET Balance = Balance - p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_from\_account\_id;

UPDATE Accounts

SET Balance = Balance + p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_to\_account\_id;

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred during the transfer.');

END SafeTransferFunds;

/

1. **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.

CREATE or REPLACE PROCEDURE UpdateSalary(

EmployeeID IN NUMBER,

Increment IN NUMBER

)

IS

Id\_not\_found EXCEPTION;

Updated\_salary NUMBER;

BEGIN

Select Salary INTO Updated\_salary

From Employee

Where Employee.EmployeeID = EmployeeID

For UPDATE;

Update employee

Set salary = salary\*(100+Increment)/100

Where Employee.EmployeeID = EmployeeID;

Commit;

EXCEPTION

When NO\_DATA\_FOUND then

Raise Id\_not\_found;

When Id\_not\_found then

DBMS\_OUTPUT.put\_line('Error: Employee ID ' || p\_employee\_id || ' not found.');

When OTHERS then

ROLLBACK;

DBMS\_OUTPUT.put\_line('Error: Some error occured');

END UpdateSalary;

/

1. **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.

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

)

IS

customer\_exists 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 customer\_exists;

WHEN customer\_exists THEN

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: An unexpected error occurred while adding a new customer.');

END AddNewCustomer;

/

**Exercise 3: Stored Procedures**

1. **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.**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

UPDATE Accounts

SET Balance = Balance \* 1.01

WHERE AccountType = 'Savings';

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest applied to all savings accounts.');

END ProcessMonthlyInterest;

1. **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.**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department IN VARCHAR2,

p\_bonus\_percentage IN NUMBER

) AS

BEGIN

UPDATE Employees

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

WHERE Department = p\_department;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Bonus applied to all employees in department: ' || p\_department);

END UpdateEmployeeBonus;

1. **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.**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_from\_account;

IF v\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in account ' || p\_from\_account);

END IF;

BEGIN

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount || ' from account ' || p\_from\_account || ' to account ' || p\_to\_account || ' completed successfully.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

END TransferFunds;

**Exercise 4: Functions**

1. **Question: Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.**

CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE)

RETURN NUMBER

IS

v\_age NUMBER;

BEGIN

SELECT FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12) INTO v\_age FROM dual;

RETURN v\_age;

EXCEPTION

WHEN OTHERS THEN

RETURN NULL;

END;

1. **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.**

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\_months NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

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

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

IF v\_monthly\_interest\_rate > 0 THEN

v\_monthly\_installment := (p\_loan\_amount \* v\_monthly\_interest\_rate) /

(1 - POWER(1 + v\_monthly\_interest\_rate, -v\_number\_of\_months));

ELSE

v\_monthly\_installment := p\_loan\_amount / v\_number\_of\_months;

END IF;

RETURN v\_monthly\_installment;

EXCEPTION

WHEN OTHERS THEN

RETURN NULL;

END;

1. **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.**

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 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**

1. **Write a trigger UpdateCustomerLastModified that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

1. **Write a trigger LogTransaction that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.**

CREATE TABLE AuditLog (

AuditID NUMBER PRIMARY KEY,

TransactionID NUMBER,

ChangeDate DATE,

ChangeType VARCHAR2(50)

);

CREATE SEQUENCE AuditLogSeq

START WITH 1

INCREMENT BY 1

NOCACHE

NOCYCLE;

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (AuditID, TransactionID, ChangeDate, ChangeType)

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

END;

1. **Write a trigger CheckTransactionRules that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

IF :NEW.TransactionType = 'Withdrawal' THEN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

IF v\_balance < :NEW.Amount THEN

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

END IF;

END IF;

IF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

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

END IF;

END IF;

END;

**Exercise 6: Cursors**

1. **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.**

DECLARE

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

LOOP

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

EXIT WHEN cur\_transactions%NOTFOUND;

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

DBMS\_OUTPUT.PUT\_LINE('Transaction Date: ' || v\_transactionDate);

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

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

END LOOP;

CLOSE cur\_transactions;

END;

1. **Write a PL/SQL block using an explicit cursor ApplyAnnualFee that deducts an annual maintenance fee from the balance of all accounts.**

DECLARE

CURSOR cur\_accounts IS

SELECT AccountID, Balance

FROM Accounts;

v\_accountID Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

v\_annualFee CONSTANT NUMBER := 100;

BEGIN

OPEN cur\_accounts;

LOOP

FETCH cur\_accounts INTO v\_accountID, v\_balance;

EXIT WHEN cur\_accounts%NOTFOUND;

UPDATE Accounts

SET Balance = Balance - v\_annualFee

WHERE AccountID = v\_accountID;

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || v\_accountID || ' New Balance: ' || (v\_balance - v\_annualFee));

END LOOP;

CLOSE cur\_accounts;

END;

1. **Write a PL/SQL block using an explicit cursor UpdateLoanInterestRates that fetches all loans and updates their interest rates based on the new policy.**

DECLARE

CURSOR cur\_loans IS

SELECT LoanID, InterestRate

FROM Loans;

v\_loanID Loans.LoanID%TYPE;

v\_interestRate Loans.InterestRate%TYPE;

v\_newInterestRate CONSTANT NUMBER := 5;

BEGIN

OPEN cur\_loans;

LOOP

FETCH cur\_loans INTO v\_loanID, v\_interestRate;

EXIT WHEN cur\_loans%NOTFOUND;

UPDATE Loans

SET InterestRate = v\_newInterestRate

WHERE LoanID = v\_loanID;

DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || v\_loanID || ' New Interest Rate: ' || v\_newInterestRate);

END LOOP;

CLOSE cur\_loans;

END;

**Exercise 7: Packages**

1. **Create a package CustomerManagement with procedures for adding a new customer, updating customer details, and a function to get customer balance.**

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);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

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;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

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;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END GetCustomerBalance;

END CustomerManagement;

1. **Write a package EmployeeManagement with procedures to hire new employees, update employee details, and a function to calculate annual salary.**

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);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

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;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

END UpdateEmployee;

FUNCTION CalculateAnnualSalary(p\_EmployeeID NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary

FROM Employees

WHERE EmployeeID = p\_EmployeeID;

RETURN v\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END CalculateAnnualSalary;

END EmployeeManagement;

1. **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.**

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);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

END OpenAccount;

PROCEDURE CloseAccount(p\_AccountID NUMBER) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_AccountID;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

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;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

END GetTotalBalance;

END AccountOperations;