**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\_age NUMBER;

v\_discount\_rate NUMBER := 0.01; -- 1% discount

v\_customer\_id NUMBER;

CURSOR c\_customers IS

SELECT CustomerID, DOB FROM Customers;

BEGIN

FOR customer\_rec IN c\_customers LOOP

-- Calculate age

v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, customer\_rec.DOB) / 12);

-- Check if the customer is above 60 years old

IF v\_age > 60 THEN

-- Apply discount to their loan interest rate

UPDATE Loans

SET InterestRate = InterestRate - (InterestRate \* v\_discount\_rate)

WHERE CustomerID = customer\_rec.CustomerID;

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:

DECLARE

v\_balance NUMBER;

CURSOR c\_customers IS

SELECT CustomerID, Balance FROM Customers;

BEGIN

FOR customer\_rec IN c\_customers LOOP

-- Check if the customer's balance is over $10,000

IF customer\_rec.Balance > 10000 THEN

-- Promote to VIP status

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = customer\_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

CURSOR c\_loans IS

SELECT l.LoanID, l.DueDate, c.Name, c.CustomerID

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.DueDate BETWEEN SYSDATE AND SYSDATE + 30;

BEGIN

FOR loan\_rec IN c\_loans LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Dear ' || loan\_rec.Name ||

', your loan with ID ' || loan\_rec.LoanID ||

' is due on ' || TO\_CHAR(loan\_rec.DueDate, '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

) IS

v\_balance NUMBER;

BEGIN

-- Check if the 'from' account has sufficient balance

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

IF v\_balance < p\_amount THEN

-- If insufficient funds, raise an error

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

ELSE

-- Deduct the amount from the 'from' account

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account\_id;

-- Add the amount to the 'to' account

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account\_id;

-- Commit the transaction

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

-- Rollback in case of any error

ROLLBACK;

-- Log the error message

INSERT INTO ErrorLog (ErrorMessage, ErrorDate)

VALUES (SQLERRM, SYSDATE);

-- Reraise the exception if necessary

RAISE;

END SafeTransferFunds;

/

**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

) IS

v\_current\_salary NUMBER;

BEGIN

-- Fetch the current salary for the given employee

SELECT Salary INTO v\_current\_salary FROM Employees WHERE EmployeeID = p\_employee\_id FOR UPDATE;

-- Update the salary with the given percentage increase

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

-- Commit the transaction

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

-- Handle case where employee ID does not exist

INSERT INTO ErrorLog (ErrorMessage, ErrorDate)

VALUES ('Employee ID ' || p\_employee\_id || ' not found.', SYSDATE);

WHEN OTHERS THEN

-- Rollback and log any other errors

ROLLBACK;

INSERT INTO ErrorLog (ErrorMessage, ErrorDate)

VALUES (SQLERRM, SYSDATE);

RAISE;

END UpdateSalary;

/

**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

) IS

BEGIN

-- Attempt to insert the new customer

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

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

-- Commit the transaction

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

-- Handle case where a customer with the same ID already exists

INSERT INTO ErrorLog (ErrorMessage, ErrorDate)

VALUES ('Customer ID ' || p\_customer\_id || ' already exists.', SYSDATE);

WHEN OTHERS THEN

-- Rollback and log any other errors

ROLLBACK;

INSERT INTO ErrorLog (ErrorMessage, ErrorDate)

VALUES (SQLERRM, SYSDATE);

RAISE;

END AddNewCustomer;

/

**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

v\_interest\_rate CONSTANT NUMBER := 0.01; -- 1% interest rate

BEGIN

-- Update the balance of all savings accounts by adding 1% interest

UPDATE Accounts

SET Balance = Balance + (Balance \* v\_interest\_rate)

WHERE AccountType = 'Savings';

-- Commit the changes

COMMIT;

END ProcessMonthlyInterest;

/

**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\_id IN NUMBER,

p\_bonus\_percentage IN NUMBER

) IS

BEGIN

-- Update the salary of employees in the specified department

UPDATE Employees

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

WHERE DepartmentID = p\_department\_id;

-- Commit the changes

COMMIT;

END UpdateEmployeeBonus;

/

**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\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) IS

v\_balance NUMBER;

BEGIN

-- Check the balance of the source account

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

-- Ensure the source account has sufficient balance

IF v\_balance < p\_amount THEN

-- Raise an error if insufficient funds

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

ELSE

-- Deduct the amount from the source account

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account\_id;

-- Add the amount to the destination account

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account\_id;

-- Commit the transaction

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

-- Rollback the transaction in case of any error

ROLLBACK;

-- Optionally log the error or re-raise it

RAISE;

END TransferFunds;

/

**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

-- Calculate age in years

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

RETURN v\_age;

END CalculateAge;

/

**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\_annual\_interest\_rate IN NUMBER,

p\_loan\_duration\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_interest\_rate NUMBER;

v\_total\_payments NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

-- Convert annual interest rate to a monthly interest rate

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

-- Calculate total number of payments (months)

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

-- Calculate the monthly installment using the formula for an annuity

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

(1 - POWER(1 + v\_monthly\_interest\_rate, -v\_total\_payments));

RETURN v\_monthly\_installment;

END CalculateMonthlyInstallment;

/

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

**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 NUMBER;

BEGIN

-- Fetch the current balance for the account

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

-- Check if the balance is sufficient

IF v\_balance >= p\_amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

-- Handle case where account ID does not exist

RETURN FALSE;

END HasSufficientBalance;

/

**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

-- Update the LastModified column to the current date

:NEW.LastModified := SYSDATE;

END UpdateCustomerLastModified;

/

**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 a record into the AuditLog table

INSERT INTO AuditLog (TransactionID, AccountID, Amount, TransactionDate, AuditDate)

VALUES (:NEW.TransactionID, :NEW.AccountID, :NEW.Amount, :NEW.TransactionDate, SYSDATE);

END LogTransaction;

/

**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 NUMBER;

BEGIN

-- Check if the transaction is a withdrawal (negative amount)

IF :NEW.Amount < 0 THEN

-- Get the current balance for the account

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = :NEW.AccountID FOR UPDATE;

-- Ensure the withdrawal does not exceed the balance

IF v\_balance + :NEW.Amount < 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Withdrawal amount exceeds available balance.');

END IF;

ELSIF :NEW.Amount = 0 THEN

-- Ensure the deposit amount is positive

RAISE\_APPLICATION\_ERROR(-20003, 'Deposit amount must be greater than zero.');

END IF;

END CheckTransactionRules;

/

**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 cur\_monthly\_transactions IS

SELECT CustomerID, TransactionID, Amount, TransactionDate

FROM Transactions

WHERE TRUNC(TransactionDate, 'MM') = TRUNC(SYSDATE, 'MM');

v\_customer\_id Customers.CustomerID%TYPE;

v\_transaction\_id Transactions.TransactionID%TYPE;

v\_amount Transactions.Amount%TYPE;

v\_transaction\_date Transactions.TransactionDate%TYPE;

BEGIN

OPEN cur\_monthly\_transactions;

LOOP

FETCH cur\_monthly\_transactions INTO v\_customer\_id, v\_transaction\_id, v\_amount, v\_transaction\_date;

EXIT WHEN cur\_monthly\_transactions%NOTFOUND;

-- Print or log the statement for the current transaction

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

', Transaction ID: ' || v\_transaction\_id ||

', Amount: ' || v\_amount ||

', Date: ' || v\_transaction\_date);

END LOOP;

CLOSE cur\_monthly\_transactions;

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

DECLARE

CURSOR cur\_accounts IS

SELECT AccountID, Balance

FROM Accounts;

v\_account\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

v\_annual\_fee CONSTANT NUMBER := 50; -- Define the annual fee

BEGIN

OPEN cur\_accounts;

LOOP

FETCH cur\_accounts INTO v\_account\_id, v\_balance;

EXIT WHEN cur\_accounts%NOTFOUND;

-- Deduct the annual fee from the account balance

UPDATE Accounts

SET Balance = Balance - v\_annual\_fee

WHERE AccountID = v\_account\_id;

-- Optionally log the fee deduction

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || v\_account\_id ||

' has been charged an annual fee of $' || v\_annual\_fee);

END LOOP;

CLOSE cur\_accounts;

-- Commit the changes

COMMIT;

END;

/

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

**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 cur\_loans IS

SELECT LoanID, InterestRate

FROM Loans;

v\_loan\_id Loans.LoanID%TYPE;

v\_interest\_rate Loans.InterestRate%TYPE;

v\_new\_interest\_rate NUMBER;

BEGIN

OPEN cur\_loans;

LOOP

FETCH cur\_loans INTO v\_loan\_id, v\_interest\_rate;

EXIT WHEN cur\_loans%NOTFOUND;

-- Define the new interest rate based on some policy (for simplicity, let's assume a 0.5% increase)

v\_new\_interest\_rate := v\_interest\_rate + 0.5;

-- Update the interest rate for the current loan

UPDATE Loans

SET InterestRate = v\_new\_interest\_rate

WHERE LoanID = v\_loan\_id;

-- Optionally log the interest rate update

DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || v\_loan\_id ||

' interest rate updated to ' || v\_new\_interest\_rate || '%');

END LOOP;

CLOSE cur\_loans;

-- Commit the changes

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 CustomerManagement AS

-- Procedure to add a new customer

PROCEDURE AddCustomer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

);

-- Procedure to update customer details

PROCEDURE UpdateCustomer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE

);

-- Function to get customer balance

FUNCTION GetCustomerBalance(

p\_customer\_id IN NUMBER

) RETURN NUMBER;

END CustomerManagement;

/

PACKAGE BODY:  
  
CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

-- Procedure to add a new customer

PROCEDURE AddCustomer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) IS

BEGIN

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

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

END AddCustomer;

-- Procedure to update customer details

PROCEDURE UpdateCustomer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE

) IS

BEGIN

UPDATE Customers

SET Name = p\_name,

DOB = p\_dob,

LastModified = SYSDATE

WHERE CustomerID = p\_customer\_id;

END UpdateCustomer;

-- Function to get customer balance

FUNCTION GetCustomerBalance(

p\_customer\_id IN NUMBER

) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Customers WHERE CustomerID = p\_customer\_id;

RETURN v\_balance;

END GetCustomerBalance;

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 EmployeeManagement AS

-- Procedure to hire a new employee

PROCEDURE HireEmployee(

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_department\_id IN NUMBER,

p\_salary IN NUMBER

);

-- Procedure to update employee details

PROCEDURE UpdateEmployee(

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_department\_id IN NUMBER,

p\_salary IN NUMBER

);

-- Function to calculate annual salary

FUNCTION CalculateAnnualSalary(

p\_employee\_id IN NUMBER

) RETURN NUMBER;

END EmployeeManagement;

/

PACKAGE BODY:

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

-- Procedure to hire a new employee

PROCEDURE HireEmployee(

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_department\_id IN NUMBER,

p\_salary IN NUMBER

) IS

BEGIN

INSERT INTO Employees (EmployeeID, Name, DepartmentID, Salary)

VALUES (p\_employee\_id, p\_name, p\_department\_id, p\_salary);

END HireEmployee;

-- Procedure to update employee details

PROCEDURE UpdateEmployee(

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_department\_id IN NUMBER,

p\_salary IN NUMBER

) IS

BEGIN

UPDATE Employees

SET Name = p\_name,

DepartmentID = p\_department\_id,

Salary = p\_salary

WHERE EmployeeID = p\_employee\_id;

END UpdateEmployee;

-- Function to calculate annual salary

FUNCTION CalculateAnnualSalary(

p\_employee\_id IN NUMBER

) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary \* 12 INTO v\_salary FROM Employees WHERE EmployeeID = p\_employee\_id;

RETURN v\_salary;

END CalculateAnnualSalary;

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 AccountOperations AS

-- Procedure to open a new account

PROCEDURE OpenAccount(

p\_account\_id IN NUMBER,

p\_customer\_id IN NUMBER,

p\_account\_type IN VARCHAR2,

p\_initial\_balance IN NUMBER

);

-- Procedure to close an account

PROCEDURE CloseAccount(

p\_account\_id IN NUMBER

);

-- Function to get total balance across all accounts for a customer

FUNCTION GetTotalBalance(

p\_customer\_id IN NUMBER

) RETURN NUMBER;

END AccountOperations;

/

PACKAGE BODY:

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

-- Procedure to open a new account

PROCEDURE OpenAccount(

p\_account\_id IN NUMBER,

p\_customer\_id IN NUMBER,

p\_account\_type IN VARCHAR2,

p\_initial\_balance IN NUMBER

) IS

BEGIN

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

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

END OpenAccount;

-- Procedure to close an account

PROCEDURE CloseAccount(

p\_account\_id IN NUMBER

) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_account\_id;

END CloseAccount;

-- Function to get total balance across all accounts for a customer

FUNCTION GetTotalBalance(

p\_customer\_id IN 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;

END GetTotalBalance;

END AccountOperations;

/