**Exercise 1: Control Structures**

**Scenario 1: Applying a Discount to Loan Interest Rates for Customers Above 60**

**Question:**  
The bank wants to apply a discount to loan interest rates for customers above 60 years old. 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.

**PL/SQL Block:**

DECLARE

v\_age NUMBER;

v\_discount NUMBER := 1;

v\_new\_rate NUMBER;

BEGIN

FOR rec IN (SELECT c.CustomerID, l.LoanID, l.InterestRate, c.DOB

FROM Customers c

JOIN Loans l ON c.CustomerID = l.CustomerID) LOOP

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

IF v\_age > 60 THEN

v\_new\_rate := rec.InterestRate - v\_discount;

UPDATE Loans

SET InterestRate = v\_new\_rate

WHERE LoanID = rec.LoanID;

END IF;

END LOOP;

COMMIT;

END;

/

**Scenario 2: Promoting Customers to VIP Status Based on Balance**

**Question:**  
A customer can be promoted to VIP status based on their balance. 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.

**PL/SQL Block:**

BEGIN

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

IF rec.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'YES'

WHERE CustomerID = rec.CustomerID;

END IF;

END LOOP;

COMMIT;

END;

/

**Scenario 3: Sending Reminders for Loans Due in the Next 30 Days**

**Question:**  
The bank wants to send reminders to customers whose loans are due within the next 30 days. Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

**PL/SQL Block:**

DECLARE

v\_due\_date DATE;

BEGIN

FOR rec IN (SELECT l.LoanID, l.CustomerID, c.Name, l.EndDate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30) LOOP

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

', your loan (Loan ID: ' || rec.LoanID ||

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

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.

**PL/SQL Block:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id

FOR UPDATE;

IF v\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in the account.');

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 OTHERS THEN

ROLLBACK;

INSERT INTO ErrorLog (ErrorMessage, ErrorDate)

VALUES (SQLERRM, SYSDATE);

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.

**PL/SQL Block:**

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) AS

BEGIN

UPDATE Employees

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

LastModified = SYSDATE

WHERE EmployeeID = p\_employee\_id;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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.

**PL/SQL Block:**

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

ROLLBACK;

INSERT INTO ErrorLog (ErrorMessage, ErrorDate)

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

WHEN OTHERS THEN

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.

**PL/SQL Block:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

UPDATE Accounts

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

LastModified = SYSDATE

WHERE AccountType = 'Savings';

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.

**PL/SQL Block:**

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),

LastModified = SYSDATE

WHERE Department = p\_department;

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.

**PL/SQL Block:**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id

FOR UPDATE;

IF v\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in the account.');

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 OTHERS THEN

ROLLBACK;

INSERT INTO ErrorLog (ErrorMessage, ErrorDate)

VALUES (SQLERRM, SYSDATE);

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.

**PL/SQL Block:**

CREATE OR REPLACE FUNCTION CalculateAge (

p\_dob DATE

) RETURN NUMBER AS

v\_age NUMBER;

BEGIN

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.

**PL/SQL Block:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount NUMBER,

p\_interest\_rate NUMBER,

p\_duration\_years NUMBER

) RETURN NUMBER AS

v\_monthly\_installment NUMBER;

v\_monthly\_rate NUMBER;

v\_total\_months NUMBER;

BEGIN

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

v\_total\_months := p\_duration\_years \* 12;

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

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

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.

**PL/SQL Block:**

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_account\_id NUMBER,

p\_amount NUMBER

) RETURN BOOLEAN AS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_account\_id;

IF v\_balance >= p\_amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

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.

**PL/SQL Block:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

: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.

**PL/SQL Block:**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, Action, ActionDate)

VALUES (:NEW.TransactionID, 'INSERT', 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.

**PL/SQL Block:**

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(-20002, 'Insufficient funds for withdrawal.');

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

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

END IF;

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.

**PL/SQL Block:**

DECLARE

CURSOR c\_transactions IS

SELECT t.TransactionID, t.AccountID, t.TransactionDate, t.Amount, t.TransactionType, a.CustomerID

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

WHERE EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE);

v\_customer\_id Customers.CustomerID%TYPE;

v\_transaction\_details VARCHAR2(4000);

BEGIN

FOR rec IN c\_transactions LOOP

v\_customer\_id := rec.CustomerID;

v\_transaction\_details := 'Transaction ID: ' || rec.TransactionID ||

', Date: ' || TO\_CHAR(rec.TransactionDate, 'YYYY-MM-DD') ||

', Amount: ' || rec.Amount ||

', Type: ' || rec.TransactionType;

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

DBMS\_OUTPUT.PUT\_LINE(v\_transaction\_details);

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

END LOOP;

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.

**PL/SQL Block:**

DECLARE

CURSOR c\_accounts IS

SELECT AccountID, Balance

FROM Accounts;

v\_annual\_fee NUMBER := 50; -- Example fee amount

BEGIN

FOR rec IN c\_accounts LOOP

UPDATE Accounts

SET Balance = Balance - v\_annual\_fee, LastModified = SYSDATE

WHERE AccountID = rec.AccountID;

END LOOP;

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.

**PL/SQL Block:**

DECLARE

CURSOR c\_loans IS

SELECT LoanID, InterestRate

FROM Loans;

v\_new\_interest\_rate NUMBER := 6; -- Example new interest rate

BEGIN

FOR rec IN c\_loans LOOP

UPDATE Loans

SET InterestRate = v\_new\_interest\_rate

WHERE LoanID = rec.LoanID;

END LOOP;

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.

**PL/SQL Block:**

-- Package Specification

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddNewCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

);

PROCEDURE UpdateCustomerDetails (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

);

FUNCTION GetCustomerBalance (

p\_customer\_id IN NUMBER

) RETURN NUMBER;

END CustomerManagement;

/

-- Package Body

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer (

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);

COMMIT;

END AddNewCustomer;

PROCEDURE UpdateCustomerDetails (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) IS

BEGIN

UPDATE Customers

SET Name = p\_name, DOB = p\_dob, Balance = p\_balance, LastModified = SYSDATE

WHERE CustomerID = p\_customer\_id;

COMMIT;

END UpdateCustomerDetails;

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.

**PL/SQL Block:**

-- Package Specification

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireNewEmployee (

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_department IN VARCHAR2,

p\_hire\_date IN DATE

);

PROCEDURE UpdateEmployeeDetails (

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_department IN VARCHAR2

);

FUNCTION CalculateAnnualSalary (

p\_employee\_id IN NUMBER

) RETURN NUMBER;

END EmployeeManagement;

/

-- Package Body

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireNewEmployee (

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_department IN VARCHAR2,

p\_hire\_date IN DATE

) IS

BEGIN

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

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

COMMIT;

END HireNewEmployee;

PROCEDURE UpdateEmployeeDetails (

p\_employee\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_position IN VARCHAR2,

p\_salary IN NUMBER,

p\_department IN VARCHAR2

) IS

BEGIN

UPDATE Employees

SET Name = p\_name, Position = p\_position, Salary = p\_salary, Department = p\_department

WHERE EmployeeID = p\_employee\_id;

COMMIT;

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary (

p\_employee\_id IN NUMBER

) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_salary

FROM Employees

WHERE EmployeeID = p\_employee\_id;

RETURN v\_salary \* 12;

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.

**PL/SQL Block:**

-- Package Specification

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenNewAccount (

p\_account\_id IN NUMBER,

p\_customer\_id IN NUMBER,

p\_account\_type IN VARCHAR2,

p\_balance IN NUMBER

);

PROCEDURE CloseAccount (

p\_account\_id IN NUMBER

);

FUNCTION GetTotalBalance (

p\_customer\_id IN NUMBER

) RETURN NUMBER;

END AccountOperations;

/

-- Package Body

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenNewAccount (

p\_account\_id IN NUMBER,

p\_customer\_id IN NUMBER,

p\_account\_type IN VARCHAR2,

p\_balance IN NUMBER

) IS

BEGIN

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

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

COMMIT;

END OpenNewAccount;

PROCEDURE CloseAccount (

p\_account\_id IN NUMBER

) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_account\_id;

COMMIT;

END CloseAccount;

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;

/