**2. PL/SQL PROGRAMMING**

**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.

**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.

**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.

**ANS:**

**Scenario 1: Apply Discount for Customers Above 60**

**Requirement:**  
For all customers aged above 60, reduce their **loan interest rate** by **1%**.

**Code:**

DECLARE

v\_age NUMBER;

BEGIN

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

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

IF v\_age > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = cust.CustomerID;

END IF;

END LOOP;

COMMIT;

END;

**Scenario 2: Promote to VIP Status Based on Balance**

**Requirement:**  
If a customer has a **balance > $10,000**, set their IsVIP flag to TRUE.  
(You may need to **alter the table** to add IsVIP column first.)

**Code:**

-- One-time: Add IsVIP column if not exists

ALTER TABLE Customers ADD IsVIP VARCHAR2(5);

-- Main block

BEGIN

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

IF cust.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = cust.CustomerID;

END IF;

END LOOP;

COMMIT;

END;

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

**Requirement:**  
Print reminder messages for loans where EndDate is within 30 days from today.

**Code:**

BEGIN

FOR loan IN (

SELECT LoanID, CustomerID, EndDate

FROM Loans

WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || loan.LoanID ||

' for Customer ' || loan.CustomerID ||

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

**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.

**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.

**Scenario 1: Safe Fund Transfer Between Accounts**

**Requirement:**  
Create a procedure SafeTransferFunds(from\_acc IN NUMBER, to\_acc IN NUMBER, amount IN NUMBER)

* Check for sufficient balance
* If any error (like insufficient funds), **log error** and **rollback**

**Code:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

from\_acc IN NUMBER,

to\_acc IN NUMBER,

amount IN NUMBER

) IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = from\_acc FOR UPDATE;

IF v\_balance < amount THEN

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

END IF;

UPDATE Accounts SET Balance = Balance - amount WHERE AccountID = from\_acc;

UPDATE Accounts SET Balance = Balance + amount WHERE AccountID = to\_acc;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

**Scenario 2: Update Employee Salary with Error Handling**

**Requirement:**  
Create UpdateSalary(emp\_id IN NUMBER, percent IN NUMBER)

* Increase salary
* If employee ID not found, **handle exception** and log it

**Code:**

CREATE OR REPLACE PROCEDURE UpdateSalary(

emp\_id IN NUMBER,

percent IN NUMBER

) IS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* percent / 100)

WHERE EmployeeID = emp\_id;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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

END;

**Scenario 3: Add New Customer with Duplicate ID Handling**

**Requirement:**  
Create AddNewCustomer(...) to insert new record

* If **CustomerID already exists**, catch the exception and **log error**

**Code:**

CREATE OR REPLACE PROCEDURE AddNewCustomer(

p\_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\_id, p\_name, p\_dob, p\_balance, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END;

**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.

**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.

**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.

**Scenario 1: Process Monthly Interest for Savings Accounts**

**Requirement:**  
Create ProcessMonthlyInterest

* Apply **1% interest** to all accounts where AccountType = 'Savings'

**Code:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'Savings';

COMMIT;

END;

**Scenario 2: Update Employee Bonus by Department**

**Requirement:**  
Create UpdateEmployeeBonus(dept IN VARCHAR2, bonus\_percent IN NUMBER)

* Add bonus to **all employees in that department**

**Code:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

dept IN VARCHAR2,

bonus\_percent IN NUMBER

) IS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* bonus\_percent / 100)

WHERE Department = dept;

COMMIT;

END;

**Scenario 3: Transfer Funds Between Customer Accounts**

**Requirement:**  
Create TransferFunds(from\_acc IN NUMBER, to\_acc IN NUMBER, amount IN NUMBER)

* Ensure **sufficient balance** before transferring

**Code:**

CREATE OR REPLACE PROCEDURE TransferFunds(

from\_acc IN NUMBER,

to\_acc IN NUMBER,

amount IN NUMBER

) IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = from\_acc FOR UPDATE;

IF v\_balance < amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance in source account.');

END IF;

UPDATE Accounts SET Balance = Balance - amount WHERE AccountID = from\_acc;

UPDATE Accounts SET Balance = Balance + amount WHERE AccountID = to\_acc;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

**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.

**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.

**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.

**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.

**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.

**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.

**Scenario 1: Calculate Age from DOB**

**Requirement:**  
Create CalculateAge(dob IN DATE) RETURN NUMBER

* Return age in **years** using MONTHS\_BETWEEN

**Code:**

CREATE OR REPLACE FUNCTION CalculateAge(

dob IN DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, dob) / 12);

RETURN v\_age;

END;

**Scenario 2: Calculate Monthly Loan Installment**

**Requirement:**  
Create CalculateMonthlyInstallment(loan\_amt IN NUMBER, rate IN NUMBER, duration\_years IN NUMBER) RETURN NUMBER

* Use **EMI formula**:

EMI=P⋅r⋅(1+r)n(1+r)n−1EMI = \frac{P \cdot r \cdot (1 + r)^n}{(1 + r)^n - 1}

where P = loan\_amt, r = monthly interest rate, n = total months

**Code:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

loan\_amt IN NUMBER,

rate IN NUMBER, -- Annual interest rate in percent

duration\_years IN NUMBER

) RETURN NUMBER IS

r NUMBER := rate / 12 / 100; -- Monthly rate

n NUMBER := duration\_years \* 12; -- Total months

emi NUMBER;

BEGIN

emi := (loan\_amt \* r \* POWER(1 + r, n)) / (POWER(1 + r, n) - 1);

RETURN ROUND(emi, 2);

END;

**Scenario 3: Check Sufficient Balance Before Transaction**

**Requirement:**  
Create HasSufficientBalance(acc\_id IN NUMBER, amt IN NUMBER) RETURN BOOLEAN

* Return TRUE if account has enough balance, else FALSE

**Code:**

CREATE OR REPLACE FUNCTION HasSufficientBalance(

acc\_id IN NUMBER,

amt IN NUMBER

) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

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

RETURN v\_balance >= amt;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

WHEN OTHERS THEN

RETURN FALSE;

END;

**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.

**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.

**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.

**Scenario 1: Update LastModified on Customer Update**

**Requirement:**  
Create a trigger UpdateCustomerLastModified

* Fire **before update** on Customers
* Set LastModified = SYSDATE

**Code:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

**Scenario 2: Log All Transactions into an AuditLog**

**Requirement:**

* First, **create** the AuditLog table
* Then, create trigger LogTransaction
* Fire **after insert** on Transactions

**Step 1: Create AuditLog Table**

CREATE TABLE AuditLog (

LogID NUMBER GENERATED BY DEFAULT AS IDENTITY,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10),

LoggedAt DATE DEFAULT SYSDATE

);

**Step 2: Trigger Code**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

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

VALUES (:NEW.AccountID, :NEW.TransactionDate, :NEW.Amount, :NEW.TransactionType);

END;

**Scenario 3: Enforce Transaction Rules**

**Requirement:**  
Trigger CheckTransactionRules

* Fire **before insert** on Transactions
* Rule 1: Withdrawals can’t exceed balance
* Rule 2: Deposits must be positive

**Code:**

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(-20001, 'Withdrawal exceeds account 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 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.

**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.

**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.

**Scenario 1: Generate Monthly Statements for Customers**

**Requirement:**  
Create a PL/SQL block using a cursor GenerateMonthlyStatements

* Fetch all **current month’s transactions**
* Print statement per customer

**Code:**

DECLARE

CURSOR txn\_cursor IS

SELECT t.AccountID, c.CustomerID, 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 TRUNC(t.TransactionDate, 'MM') = TRUNC(SYSDATE, 'MM');

rec txn\_cursor%ROWTYPE;

BEGIN

OPEN txn\_cursor;

LOOP

FETCH txn\_cursor INTO rec;

EXIT WHEN txn\_cursor%NOTFOUND;

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

' | Account: ' || rec.AccountID ||

' | ' || rec.TransactionType ||

' | Amount: $' || rec.Amount ||

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

END LOOP;

CLOSE txn\_cursor;

END;

**Scenario 2: Apply Annual Maintenance Fee to All Accounts**

**Requirement:**  
Use cursor ApplyAnnualFee

* Deduct a fixed annual fee (e.g. ₹100) from all accounts

**Code:**

DECLARE

CURSOR acc\_cursor IS

SELECT AccountID, Balance FROM Accounts;

v\_acc\_id Accounts.AccountID%TYPE;

BEGIN

FOR rec IN acc\_cursor LOOP

UPDATE Accounts

SET Balance = Balance - 100

WHERE AccountID = rec.AccountID;

END LOOP;

COMMIT;

END;

**Scenario 3: Update Loan Interest Rates Based on New Policy**

**Requirement:**  
Use cursor UpdateLoanInterestRates

* Update interest based on some rule (e.g., +0.5% if rate < 6)

**Code:**

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate FROM Loans;

BEGIN

FOR rec IN loan\_cursor LOOP

IF rec.InterestRate < 6 THEN

UPDATE Loans

SET InterestRate = rec.InterestRate + 0.5

WHERE LoanID = rec.LoanID;

END IF;

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.

**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.

**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.

Each package has:

1. A **Package Specification** (CREATE OR REPLACE PACKAGE)
2. A **Package Body** (CREATE OR REPLACE PACKAGE BODY)

**Scenario 1: Customer Management Package**

**Goal:**

Package CustomerManagement with:

* AddCustomer
* UpdateCustomerDetails
* GetCustomerBalance (function)

**Package Specification**

CREATE OR REPLACE PACKAGE CustomerManagement IS

PROCEDURE AddCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER);

PROCEDURE UpdateCustomerDetails(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE);

FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

**Package Body**

CREATE OR REPLACE PACKAGE BODY CustomerManagement IS

PROCEDURE AddCustomer(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);

END;

PROCEDURE UpdateCustomerDetails(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE) IS

BEGIN

UPDATE Customers

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

WHERE CustomerID = p\_id;

END;

FUNCTION GetCustomerBalance(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;

**Scenario 2: Employee Management Package**

**Goal:**

Package EmployeeManagement with:

* HireEmployee
* UpdateEmployee
* CalculateAnnualSalary (function)

**Package Specification**

CREATE OR REPLACE PACKAGE EmployeeManagement IS

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_pos VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2, p\_hire DATE);

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_salary NUMBER, p\_dept VARCHAR2);

FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

**Package Body**

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_pos VARCHAR2, p\_salary NUMBER, p\_dept VARCHAR2, p\_hire DATE) IS

BEGIN

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

VALUES (p\_id, p\_name, p\_pos, p\_salary, p\_dept, p\_hire);

END;

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_salary NUMBER, p\_dept VARCHAR2) IS

BEGIN

UPDATE Employees

SET Salary = p\_salary, Department = p\_dept

WHERE EmployeeID = p\_id;

END;

FUNCTION CalculateAnnualSalary(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;

**Scenario 3: Account Operations Package**

**Goal:**

Package AccountOperations with:

* OpenAccount
* CloseAccount
* GetTotalBalance (function)

**Package Specification**

CREATE OR REPLACE PACKAGE AccountOperations IS

PROCEDURE OpenAccount(p\_acc\_id NUMBER, p\_cust\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

PROCEDURE CloseAccount(p\_acc\_id NUMBER);

FUNCTION GetTotalBalance(p\_cust\_id NUMBER) RETURN NUMBER;

END AccountOperations;

**Package Body**

CREATE OR REPLACE PACKAGE BODY AccountOperations IS

PROCEDURE OpenAccount(p\_acc\_id NUMBER, p\_cust\_id NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

BEGIN

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

VALUES (p\_acc\_id, p\_cust\_id, p\_type, p\_balance, SYSDATE);

END;

PROCEDURE CloseAccount(p\_acc\_id NUMBER) IS

BEGIN

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

END;

FUNCTION GetTotalBalance(p\_cust\_id NUMBER) RETURN NUMBER IS

v\_total NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_total

FROM Accounts

WHERE CustomerID = p\_cust\_id;

RETURN NVL(v\_total, 0);

END;

END AccountOperations;