**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.**

**Ans:**

DECLARE

CURSOR customer\_cursor IS

SELECT CustomerID, DOB

FROM Customers;

v\_age NUMBER;

BEGIN

FOR customer\_rec IN customer\_cursor LOOP

-- Calculate age

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

IF v\_age > 60 THEN

-- Update loan interest rate with a 1% discount

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = customer\_rec.CustomerID;

END IF;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Loan interest rates updated for customers over 60 years old.');

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.**

**Ans:**

ALTER TABLE Customers ADD (IsVIP CHAR(1) DEFAULT 'N');

DECLARE

CURSOR customer\_cursor IS

SELECT CustomerID, Balance

FROM Customers;

BEGIN

FOR customer\_rec IN customer\_cursor LOOP

IF customer\_rec.Balance > 10000 THEN

-- Update IsVIP flag

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = customer\_rec.CustomerID;

ELSE

UPDATE Customers

SET IsVIP = 'N'

WHERE CustomerID = customer\_rec.CustomerID;

END IF;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('VIP status updated based on balance.');

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.**

**Ans:**

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, CustomerID, EndDate

FROM Loans

WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30;

BEGIN

FOR loan\_rec IN loan\_cursor LOOP

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

' for Customer ID ' || loan\_rec.CustomerID ||

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

'. Please make sure to process it accordingly.');

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Loan reminders sent for loans due in the next 30 days.');

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.**

**Ans:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

p\_from\_account\_id IN Accounts.AccountID%TYPE,

p\_to\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) IS

v\_balance Accounts.Balance%TYPE;

BEGIN

-- Check balance of the from account

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 source account.');

END IF;

-- Deduct amount from the from account

UPDATE Accounts

SET Balance = Balance - p\_amount, LastModified = SYSDATE

WHERE AccountID = p\_from\_account\_id;

-- Add amount to the to account

UPDATE Accounts

SET Balance = Balance + p\_amount, LastModified = SYSDATE

WHERE AccountID = p\_to\_account\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer completed successfully.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END SafeTransferFunds;

/

BEGIN

SafeTransferFunds(1, 2, 100);

END;

/

**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.**

**Ans:**

CREATE OR REPLACE PROCEDURE UpdateSalary(

p\_employee\_id IN Employees.EmployeeID%TYPE,

p\_percentage IN NUMBER

) IS

BEGIN

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END UpdateSalary;

/

BEGIN

UpdateSalary(1, 10);

END;

/

**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.**

**Ans:**

CREATE OR REPLACE PROCEDURE AddNewCustomer(

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE

) IS

BEGIN

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

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

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Customer added successfully.');

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END AddNewCustomer;

/

BEGIN

AddNewCustomer(3, 'New Customer', TO\_DATE('1980-01-01', 'YYYY-MM-DD'), 2000);

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.

**Ans:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance \* 1.01,

LastModified = SYSDATE

WHERE AccountType = 'Savings';

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest processed for all savings accounts.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END ProcessMonthlyInterest;

/

BEGIN

ProcessMonthlyInterest;

END;

/

**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.

**Ans:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_department IN Employees.Department%TYPE,

p\_bonus\_percentage IN NUMBER

) IS

BEGIN

-- Update salaries and apply bonus percentage

UPDATE Employees

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

WHERE Department = p\_department;

-- Check if the update affected any rows

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, 'No employees found in department ' || p\_department);

END IF;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Employee bonuses updated for department ' || p\_department || '.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END UpdateEmployeeBonus;

/

BEGIN

UpdateEmployeeBonus('IT', 10);

END;

/

**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.

**Ans:**

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_from\_account\_id IN Accounts.AccountID%TYPE,

p\_to\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) IS

v\_from\_balance Accounts.Balance%TYPE;

BEGIN

-- Check balance of the from account

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id

FOR UPDATE;

IF v\_from\_balance < p\_amount THEN

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

END IF;

-- Deduct amount from the from account

UPDATE Accounts

SET Balance = Balance - p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_from\_account\_id;

-- Add amount to the to account

UPDATE Accounts

SET Balance = Balance + p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_to\_account\_id;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END TransferFunds;

/

BEGIN

TransferFunds(1, 2, 100); -- Transfer 100 from account 1 to account 2

END;

/

**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.

**Ans:**

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;

END CalculateAge;

/

DECLARE

v\_age NUMBER;

BEGIN

v\_age := CalculateAge(TO\_DATE('1985-05-15', 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('Age: ' || v\_age);

END;

/

**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.

**Ans:**

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\_total\_payments NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

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

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

IF v\_monthly\_interest\_rate = 0 THEN

v\_monthly\_installment := p\_loan\_amount / v\_total\_payments;

ELSE

v\_monthly\_installment := p\_loan\_amount \* (v\_monthly\_interest\_rate \* POWER(1 + v\_monthly\_interest\_rate, v\_total\_payments)) /

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

END IF;

RETURN v\_monthly\_installment;

END CalculateMonthlyInstallment;

/

DECLARE

v\_installment NUMBER;

BEGIN

v\_installment := CalculateMonthlyInstallment(5000, 5, 5);

DBMS\_OUTPUT.PUT\_LINE('Monthly Installment: ' || v\_installment);

END;

/

**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.

**Ans:**

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) RETURN BOOLEAN IS

v\_balance Accounts.Balance%TYPE;

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;

END HasSufficientBalance;

/

DECLARE

v\_has\_sufficient\_balance BOOLEAN;

BEGIN

v\_has\_sufficient\_balance := HasSufficientBalance(1, 100);

IF v\_has\_sufficient\_balance THEN

DBMS\_OUTPUT.PUT\_LINE('Sufficient balance.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Insufficient balance.');

END IF;

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.

**Ans:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

-- Update a customer record to trigger the UpdateCustomerLastModified trigger

UPDATE Customers

SET Name = 'John Doe Updated'

WHERE CustomerID = 1;

-- Verify that the LastModified date is updated

SELECT CustomerID, LastModified FROM Customers WHERE CustomerID = 1;

**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.

**Ans:**

CREATE TABLE AuditLog (

AuditID NUMBER PRIMARY KEY,

TransactionID NUMBER,

ActionTime DATE,

ActionType VARCHAR2(50),

CONSTRAINT fk\_transaction FOREIGN KEY (TransactionID) REFERENCES Transactions(TransactionID)

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (AuditID, TransactionID, ActionTime, ActionType)

VALUES (AuditLog\_seq.NEXTVAL, :NEW.TransactionID, SYSDATE, 'INSERT');

END;

/

CREATE SEQUENCE AuditLog\_seq START WITH 1 INCREMENT BY 1;

-- Insert a new transaction to trigger the LogTransaction trigger

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

VALUES (8, 1, SYSDATE, 100, 'Deposit');

-- Verify that a record is added to the AuditLog

SELECT \* FROM AuditLog WHERE TransactionID = 8;

**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.

**Ans:**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

-- Fetch the current balance for the account

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

-- Check business rules

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

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

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

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

END IF;

END;

/

-- Attempt to insert a transaction with insufficient funds to trigger the CheckTransactionRules trigger

BEGIN

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

VALUES (9, 1, SYSDATE, 2000, 'Withdrawal'); -- Assuming AccountID 1 has less than 2000 balance

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE(SQLERRM);

END;

/

-- Attempt to insert a transaction with a negative amount to trigger the CheckTransactionRules trigger

BEGIN

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

VALUES (10, 1, SYSDATE, -100, 'Deposit');

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE(SQLERRM);

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.

**Ans:**

DECLARE

CURSOR customer\_cursor IS

SELECT DISTINCT c.CustomerID, c.Name

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\_customer customer\_cursor%ROWTYPE;

BEGIN

OPEN customer\_cursor;

LOOP

FETCH customer\_cursor INTO v\_customer;

EXIT WHEN customer\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Statement for Customer: ' || v\_customer.Name);

FOR rec IN (SELECT t.Amount, t.TransactionType, t.TransactionDate

FROM Transactions t

JOIN Accounts a ON t.AccountID = a.AccountID

WHERE a.CustomerID = v\_customer.CustomerID

AND t.TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE)) LOOP

DBMS\_OUTPUT.PUT\_LINE('Date: ' || rec.TransactionDate || ' | Type: ' || rec.TransactionType || ' | Amount: ' || rec.Amount);

END LOOP;

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

END LOOP;

CLOSE customer\_cursor;

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.

**Ans:**

DECLARE

CURSOR account\_cursor IS

SELECT AccountID, Balance

FROM Accounts;

annual\_fee CONSTANT NUMBER := 100; -- Example annual fee

BEGIN

FOR rec IN account\_cursor LOOP

UPDATE Accounts

SET Balance = Balance - annual\_fee

WHERE AccountID = rec.AccountID;

DBMS\_OUTPUT.PUT\_LINE('Applied annual fee to AccountID: ' || rec.AccountID || ' | New Balance: ' || (rec.Balance - annual\_fee));

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.

**Ans:**

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate

FROM Loans;

new\_interest\_rate CONSTANT NUMBER := 6; -- Example new interest rate policy

BEGIN

FOR rec IN loan\_cursor LOOP

UPDATE Loans

SET InterestRate = new\_interest\_rate

WHERE LoanID = rec.LoanID;

DBMS\_OUTPUT.PUT\_LINE('Updated LoanID: ' || rec.LoanID || ' | New Interest Rate: ' || new\_interest\_rate);

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.

**Ans:**

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddNewCustomer(

p\_CustomerID IN Customers.CustomerID%TYPE,

p\_Name IN Customers.Name%TYPE,

p\_DOB IN Customers.DOB%TYPE,

p\_Balance IN Customers.Balance%TYPE

);

PROCEDURE UpdateCustomerDetails(

p\_CustomerID IN Customers.CustomerID%TYPE,

p\_Name IN Customers.Name%TYPE,

p\_Balance IN Customers.Balance%TYPE

);

FUNCTION GetCustomerBalance(

p\_CustomerID IN Customers.CustomerID%TYPE

) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer(

p\_CustomerID IN Customers.CustomerID%TYPE,

p\_Name IN Customers.Name%TYPE,

p\_DOB IN Customers.DOB%TYPE,

p\_Balance IN Customers.Balance%TYPE

) 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 AddNewCustomer;

PROCEDURE UpdateCustomerDetails(

p\_CustomerID IN Customers.CustomerID%TYPE,

p\_Name IN Customers.Name%TYPE,

p\_Balance IN Customers.Balance%TYPE

) IS

BEGIN

UPDATE Customers

SET Name = p\_Name, Balance = p\_Balance, LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

END UpdateCustomerDetails;

FUNCTION GetCustomerBalance(

p\_CustomerID IN Customers.CustomerID%TYPE

) 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 0;

END GetCustomerBalance;

END CustomerManagement;

/

-- Adding a new customer

BEGIN

CustomerManagement.AddNewCustomer(3, 'Alice Johnson', TO\_DATE('1992-11-25', 'YYYY-MM-DD'), 2000);

END;

/

-- Updating a customer

BEGIN

CustomerManagement.UpdateCustomerDetails(3, 'Alice Johnson', 2500);

END;

/

-- Getting customer balance

DECLARE

v\_Balance NUMBER;

BEGIN

v\_Balance := CustomerManagement.GetCustomerBalance(3);

DBMS\_OUTPUT.PUT\_LINE('Customer Balance: ' || v\_Balance);

END;

/

**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.

**Ans:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireNewEmployee(

p\_EmployeeID IN Employees.EmployeeID%TYPE,

p\_Name IN Employees.Name%TYPE,

p\_Position IN Employees.Position%TYPE,

p\_Salary IN Employees.Salary%TYPE,

p\_Department IN Employees.Department%TYPE,

p\_HireDate IN Employees.HireDate%TYPE

);

PROCEDURE UpdateEmployeeDetails(

p\_EmployeeID IN Employees.EmployeeID%TYPE,

p\_Name IN Employees.Name%TYPE,

p\_Salary IN Employees.Salary%TYPE

);

FUNCTION CalculateAnnualSalary(

p\_EmployeeID IN Employees.EmployeeID%TYPE

) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireNewEmployee(

p\_EmployeeID IN Employees.EmployeeID%TYPE,

p\_Name IN Employees.Name%TYPE,

p\_Position IN Employees.Position%TYPE,

p\_Salary IN Employees.Salary%TYPE,

p\_Department IN Employees.Department%TYPE,

p\_HireDate IN Employees.HireDate%TYPE

) 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 HireNewEmployee;

PROCEDURE UpdateEmployeeDetails(

p\_EmployeeID IN Employees.EmployeeID%TYPE,

p\_Name IN Employees.Name%TYPE,

p\_Salary IN Employees.Salary%TYPE

) IS

BEGIN

UPDATE Employees

SET Name = p\_Name, Salary = p\_Salary

WHERE EmployeeID = p\_EmployeeID;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(

p\_EmployeeID IN Employees.EmployeeID%TYPE

) 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 0;

END CalculateAnnualSalary;

END EmployeeManagement;

/

-- Hiring a new employee

BEGIN

EmployeeManagement.HireNewEmployee(3, 'Charlie Davis', 'Analyst', 55000, 'Finance', SYSDATE);

END;

/

-- Updating an employee

BEGIN

EmployeeManagement.UpdateEmployeeDetails(3, 'Charlie Davis', 60000);

END;

/

-- Calculating annual salary

DECLARE

v\_AnnualSalary NUMBER;

BEGIN

v\_AnnualSalary := EmployeeManagement.CalculateAnnualSalary(3);

DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || v\_AnnualSalary);

END;

/

**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.

**Ans:**

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenNewAccount(

p\_AccountID IN Accounts.AccountID%TYPE,

p\_CustomerID IN Accounts.CustomerID%TYPE,

p\_AccountType IN Accounts.AccountType%TYPE,

p\_Balance IN Accounts.Balance%TYPE

);

PROCEDURE CloseAccount(

p\_AccountID IN Accounts.AccountID%TYPE

);

FUNCTION GetTotalBalance(

p\_CustomerID IN Accounts.CustomerID%TYPE

) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenNewAccount(

p\_AccountID IN Accounts.AccountID%TYPE,

p\_CustomerID IN Accounts.CustomerID%TYPE,

p\_AccountType IN Accounts.AccountType%TYPE,

p\_Balance IN Accounts.Balance%TYPE

) 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 OpenNewAccount;

PROCEDURE CloseAccount(

p\_AccountID IN Accounts.AccountID%TYPE

) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_AccountID;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

END CloseAccount;

FUNCTION GetTotalBalance(

p\_CustomerID IN Accounts.CustomerID%TYPE

) 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;

/

-- Opening a new account

BEGIN

AccountOperations.OpenNewAccount(3, 1, 'Checking', 3000);

END;

/

-- Closing an account

BEGIN

AccountOperations.CloseAccount(3);

END;

/

-- Getting total balance

DECLARE

v\_TotalBalance NUMBER;

BEGIN

v\_TotalBalance := AccountOperations.GetTotalBalance(1);

DBMS\_OUTPUT.PUT\_LINE('Total Balance: ' || v\_TotalBalance);

END;

/