**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

    CURSOR cust\_cursor IS

        SELECT c.CustomerID, c.DOB, l.LoanID, l.InterestRate

        FROM Customers c

        JOIN Loans l ON c.CustomerID = l.CustomerID;

    v\_age NUMBER;

BEGIN

    FOR rec IN cust\_cursor LOOP

        -- Calculate age

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

        IF v\_age > 60 THEN

            UPDATE Loans

            SET InterestRate = InterestRate - 1

            WHERE LoanID = rec.LoanID;

            DBMS\_OUTPUT.PUT\_LINE('Applied 1% discount for CustomerID ' || 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:**

ALTER TABLE Customers ADD IsVIP VARCHAR2(5) DEFAULT 'FALSE';

BEGIN

    FOR rec IN (

        SELECT CustomerID, Balance FROM Customers

    ) LOOP

        IF rec.Balance > 10000 THEN

            UPDATE Customers

            SET IsVIP = 'TRUE'

            WHERE CustomerID = rec.CustomerID;

            DBMS\_OUTPUT.PUT\_LINE('Customer ' || rec.CustomerID || ' promoted to VIP.');

        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 due\_loans\_cursor IS

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

        FROM Loans l

        JOIN Customers c ON l.CustomerID = c.CustomerID

        WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30;

BEGIN

    FOR rec IN due\_loans\_cursor LOOP

        DBMS\_OUTPUT.PUT\_LINE(

            'Reminder: Loan ' || rec.LoanID || ' for customer "' || rec.Name ||

            '" (CustomerID: ' || rec.CustomerID || ') is due on ' || TO\_CHAR(rec.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.

**Code:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

    p\_from\_account\_id IN NUMBER,

    p\_to\_account\_id   IN NUMBER,

    p\_amount          IN NUMBER

)

IS

    v\_from\_balance NUMBER;

BEGIN

    -- Get balance of source account

    SELECT Balance INTO v\_from\_balance

    FROM Accounts

    WHERE AccountID = p\_from\_account\_id

    FOR UPDATE;

    -- Check if sufficient balance

    IF v\_from\_balance < p\_amount THEN

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

    END IF;

    -- Deduct from source account

    UPDATE Accounts

    SET Balance = Balance - p\_amount

    WHERE AccountID = p\_from\_account\_id;

    -- Add to target account

    UPDATE Accounts

    SET Balance = Balance + p\_amount

    WHERE AccountID = p\_to\_account\_id;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Transferred $' || p\_amount || ' from Account ' || p\_from\_account\_id || ' to Account ' || p\_to\_account\_id);

EXCEPTION

    WHEN OTHERS THEN

        ROLLBACK;

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

END;

/

--example

BEGIN

    SafeTransferFunds(1, 2, 500);

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.

**Code:**

CREATE OR REPLACE PROCEDURE UpdateSalary(

    p\_employee\_id IN NUMBER,

    p\_percent     IN NUMBER

)

IS

    v\_salary NUMBER;

BEGIN

    -- Attempt to get the current salary

    SELECT Salary INTO v\_salary

    FROM Employees

    WHERE EmployeeID = p\_employee\_id

    FOR UPDATE;

    -- Update salary

    UPDATE Employees

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

    WHERE EmployeeID = p\_employee\_id;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Salary updated successfully for EmployeeID ' || p\_employee\_id);

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: Employee with ID ' || p\_employee\_id || ' does not exist.');

    WHEN OTHERS THEN

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

        ROLLBACK;

END;

/

--example1

BEGIN

    UpdateSalary(1, 10); -- Increases salary of EmployeeID 1 by 10%

END;

/

--example2

BEGIN

    UpdateSalary(99, 10); -- Nonexistent EmployeeID

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.

**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

    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 ' || p\_name || ' added successfully.');

EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

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

    WHEN OTHERS THEN

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

        ROLLBACK;

END;

/

--example1

BEGIN

    AddNewCustomer(4, 'Romit Acharya', TO\_DATE('1950-01-01', 'YYYY-MM-DD'), 12000);

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.

**Code:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest

IS

BEGIN

    UPDATE Accounts

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

        LastModified = SYSDATE

    WHERE AccountType = 'Savings';

    COMMIT;

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

EXCEPTION

    WHEN OTHERS THEN

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

        ROLLBACK;

END;

/

--example

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.

**Code:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

    p\_department IN VARCHAR2,

    p\_bonus\_pct  IN NUMBER  -- Pass 10 for 10%

)

IS

    v\_count NUMBER;

BEGIN

    UPDATE Employees

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

    WHERE Department = p\_department;

    v\_count := SQL%ROWCOUNT;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Bonus applied to ' || v\_count || ' employee(s) in department "' || p\_department || '".');

EXCEPTION

    WHEN OTHERS THEN

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

        ROLLBACK;

END;

/

--example

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.

**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 balance of source 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 source account.');

    END IF;

    -- Deduct from source account

    UPDATE Accounts

    SET Balance = Balance - p\_amount

    WHERE AccountID = p\_from\_account\_id;

    -- Credit to destination account

    UPDATE Accounts

    SET Balance = Balance + p\_amount

    WHERE AccountID = p\_to\_account\_id;

    COMMIT;

    DBMS\_OUTPUT.PUT\_LINE('Transferred ' || p\_amount || '$ from Account ' || p\_from\_account\_id || ' to Account ' || p\_to\_account\_id);

EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

        DBMS\_OUTPUT.PUT\_LINE('Error: One or both account IDs do not exist.');

        ROLLBACK;

    WHEN OTHERS THEN

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

        ROLLBACK;

END;

/

--example

BEGIN

    TransferFunds(2, 1, 300);

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.

**Code:**

CREATE OR REPLACE FUNCTION CalculateAge(

    p\_dob IN DATE

) RETURN NUMBER

IS

    v\_age NUMBER;

BEGIN

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

    RETURN v\_age;

END;

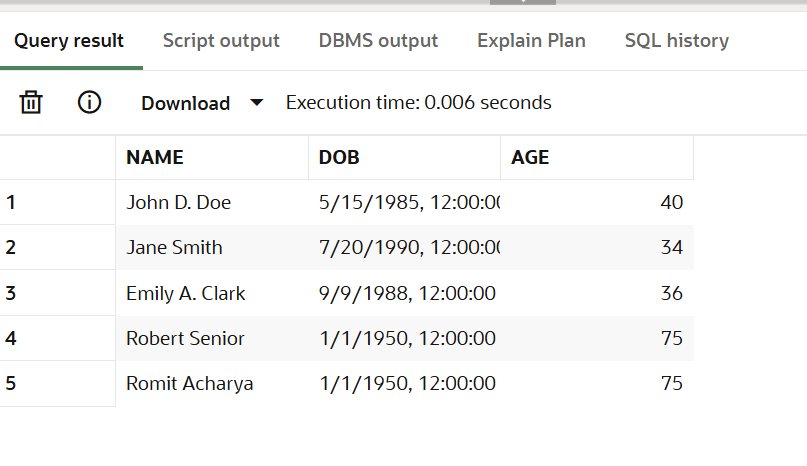
/

--example

SELECT Name, DOB, CalculateAge(DOB) AS Age

FROM Customers;

**Output:**

****

**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\_rate   IN NUMBER,

    p\_years         IN NUMBER

) RETURN NUMBER

IS

    v\_monthly\_rate   NUMBER;

    v\_months         NUMBER;

    v\_installment    NUMBER;

BEGIN

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

    v\_months := p\_years \* 12;

    -- EMI formula: P \* r \* (1+r)^n / ((1+r)^n - 1)

    v\_installment := p\_loan\_amount \* v\_monthly\_rate \* POWER(1 + v\_monthly\_rate, v\_months) / (POWER(1 + v\_monthly\_rate, v\_months) - 1);

    RETURN ROUND(v\_installment, 2);

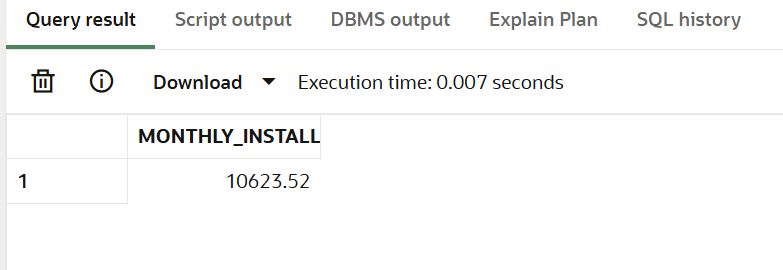
END;

/

--example

SELECT CalculateMonthlyInstallment(500000, 10, 5) AS Monthly\_Installment FROM dual;

**Output:**

****

**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

    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;

/

--usage in pl/sql

DECLARE

    v\_result BOOLEAN;

BEGIN

    v\_result := HasSufficientBalance(1, 500);

    IF v\_result THEN

        DBMS\_OUTPUT.PUT\_LINE('Account has sufficient balance.');

    ELSE

        DBMS\_OUTPUT.PUT\_LINE('Insufficient balance or account not found.');

    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.

**Code:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

    :NEW.LastModified := SYSDATE;

END;

/

--test trigger

UPDATE Customers

SET Name = 'John D. Doe'

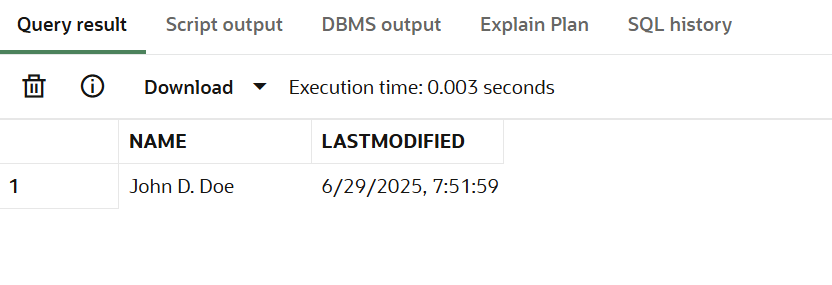
WHERE CustomerID = 1;

SELECT Name, LastModified

FROM Customers

WHERE CustomerID = 1;

**Output:**

****

**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 TABLE AuditLog (

    LogID           NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

    TransactionID   NUMBER,

    ActionType      VARCHAR2(20),

    LogDate         DATE,

    AccountID       NUMBER,

    Amount          NUMBER

);

--create trigger

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

    INSERT INTO AuditLog (

        TransactionID, ActionType, LogDate, AccountID, Amount

    )

    VALUES (

        :NEW.TransactionID, 'INSERT', SYSDATE, :NEW.AccountID, :NEW.Amount

    );

END;

/

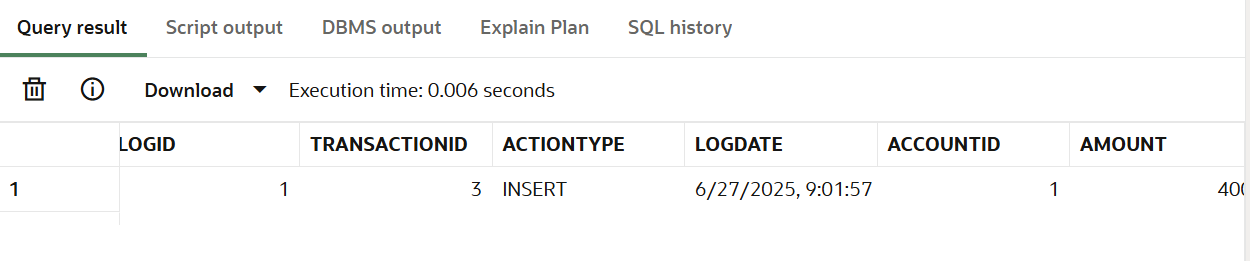
--test trigger

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

VALUES (3, 1, SYSDATE, 400, 'Deposit');

SELECT \* FROM AuditLog WHERE TransactionID = 3;

**Output:**

****

**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

    IF :NEW.TransactionType = 'Deposit' THEN

        IF :NEW.Amount <= 0 THEN

            RAISE\_APPLICATION\_ERROR(-20001, 'Deposit amount must be positive.');

        END IF;

    ELSIF :NEW.TransactionType = 'Withdrawal' THEN

        SELECT Balance INTO v\_balance

        FROM Accounts

        WHERE AccountID = :NEW.AccountID;

        IF :NEW.Amount > v\_balance THEN

            RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient balance for withdrawal.');

        END IF;

    END IF;

END;

/

--test case

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

VALUES (6, 1, SYSDATE, 5000000, 'Withdrawal');

**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 txn\_cursor IS

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

        ORDER BY c.CustomerID, t.TransactionDate;

BEGIN

    DBMS\_OUTPUT.PUT\_LINE('--- Monthly Transaction Statements ---');

    FOR rec IN txn\_cursor LOOP

        DBMS\_OUTPUT.PUT\_LINE(

            'Customer: ' || rec.Name ||

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

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

            ' | Type: ' || rec.TransactionType ||

            ' | Amount: $' || rec.Amount

        );

    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.

**Code:**

DECLARE

    CURSOR acc\_cursor IS

        SELECT AccountID, Balance

        FROM Accounts;

    v\_fee CONSTANT NUMBER := 100;

BEGIN

    FOR rec IN acc\_cursor LOOP

        IF rec.Balance >= v\_fee THEN

            UPDATE Accounts

            SET Balance = Balance - v\_fee,

                LastModified = SYSDATE

            WHERE AccountID = rec.AccountID;

            DBMS\_OUTPUT.PUT\_LINE('Annual fee of $' || v\_fee || ' deducted from Account ' || rec.AccountID);

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Account ' || rec.AccountID || ' has insufficient funds for fee deduction.');

        END IF;

    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.

**Code:**

DECLARE

    CURSOR loan\_cursor IS

        SELECT LoanID, LoanAmount, InterestRate

        FROM Loans;

    v\_new\_rate NUMBER;

BEGIN

    FOR rec IN loan\_cursor LOOP

        IF rec.LoanAmount > 10000 THEN

            v\_new\_rate := rec.InterestRate - 0.5;

        ELSE

            v\_new\_rate := rec.InterestRate + 0.25;

        END IF;

        UPDATE Loans

        SET InterestRate = v\_new\_rate

        WHERE LoanID = rec.LoanID;

        DBMS\_OUTPUT.PUT\_LINE('Loan ' || rec.LoanID || ' interest updated to ' || v\_new\_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.

**Code:**

--package specification

CREATE OR REPLACE PACKAGE CustomerManagement AS

    PROCEDURE AddCustomer(

        p\_customer\_id IN NUMBER,

        p\_name        IN VARCHAR2,

        p\_dob         IN DATE,

        p\_balance     IN NUMBER

    );

    PROCEDURE UpdateCustomer(

        p\_customer\_id IN NUMBER,

        p\_name        IN VARCHAR2,

        p\_balance     IN NUMBER

    );

    FUNCTION GetBalance(p\_customer\_id IN NUMBER) RETURN NUMBER;

END CustomerManagement;

/

--package body

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

    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);

        DBMS\_OUTPUT.PUT\_LINE('Customer added: ' || p\_name);

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

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

    END;

    PROCEDURE UpdateCustomer(

        p\_customer\_id IN NUMBER,

        p\_name        IN VARCHAR2,

        p\_balance     IN NUMBER

    ) IS

    BEGIN

        UPDATE Customers

        SET Name = p\_name,

            Balance = p\_balance,

            LastModified = SYSDATE

        WHERE CustomerID = p\_customer\_id;

        IF SQL%ROWCOUNT = 0 THEN

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

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Customer updated: ' || p\_name);

        END IF;

    END;

    FUNCTION GetBalance(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;

    EXCEPTION

        WHEN NO\_DATA\_FOUND THEN

            RETURN NULL;

    END;

END CustomerManagement;

/

--add customer

BEGIN

    CustomerManagement.AddCustomer(7, 'Emily Clark', TO\_DATE('1988-09-09', 'YYYY-MM-DD'), 9000);

END;

/

--update customer

BEGIN

    CustomerManagement.UpdateCustomer(7, 'Emily A. Clark', 9500);

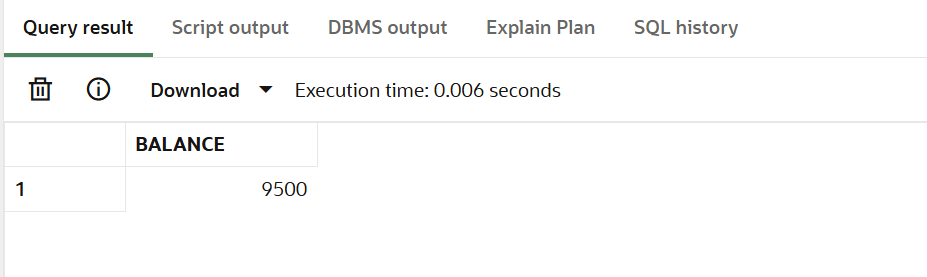
END;

/

--balance

SELECT CustomerManagement.GetBalance(7) AS Balance FROM dual;

**Output:**

****

**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:**

--package specification

CREATE OR REPLACE PACKAGE EmployeeManagement AS

    PROCEDURE HireEmployee(

        p\_employee\_id IN NUMBER,

        p\_name        IN VARCHAR2,

        p\_position    IN VARCHAR2,

        p\_salary      IN NUMBER,

        p\_department  IN VARCHAR2,

        p\_hiredate    IN DATE

    );

    PROCEDURE UpdateEmployee(

        p\_employee\_id IN NUMBER,

        p\_salary      IN NUMBER,

        p\_position    IN VARCHAR2

    );

    FUNCTION GetAnnualSalary(p\_employee\_id IN NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

--package body

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

    PROCEDURE HireEmployee(

        p\_employee\_id IN NUMBER,

        p\_name        IN VARCHAR2,

        p\_position    IN VARCHAR2,

        p\_salary      IN NUMBER,

        p\_department  IN VARCHAR2,

        p\_hiredate    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\_hiredate);

        DBMS\_OUTPUT.PUT\_LINE('Hired new employee: ' || p\_name);

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

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

    END;

    PROCEDURE UpdateEmployee(

        p\_employee\_id IN NUMBER,

        p\_salary      IN NUMBER,

        p\_position    IN VARCHAR2

    ) IS

    BEGIN

        UPDATE Employees

        SET Salary = p\_salary,

            Position = p\_position

        WHERE EmployeeID = p\_employee\_id;

        IF SQL%ROWCOUNT = 0 THEN

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

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Employee updated.');

        END IF;

    END;

    FUNCTION GetAnnualSalary(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;

    EXCEPTION

        WHEN NO\_DATA\_FOUND THEN

            RETURN NULL;

    END;

END EmployeeManagement;

/

--hire employee

BEGIN

    EmployeeManagement.HireEmployee(3, 'Sam Wilson', 'Analyst', 55000, 'Finance', TO\_DATE('2020-01-15', 'YYYY-MM-DD'));

END;

/

--update employee

BEGIN

    EmployeeManagement.UpdateEmployee(3, 58000, 'Senior Analyst');

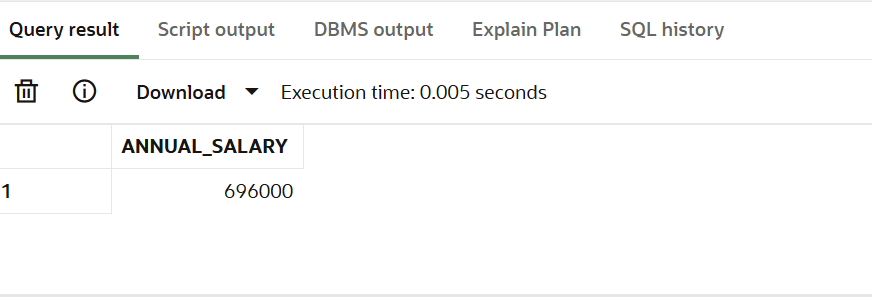
END;

/

--salary

SELECT EmployeeManagement.GetAnnualSalary(3) AS Annual\_Salary FROM dual;

**Output:**

****

**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:**

--package specification

CREATE OR REPLACE PACKAGE AccountOperations AS

    PROCEDURE OpenAccount(

        p\_account\_id  IN NUMBER,

        p\_customer\_id IN NUMBER,

        p\_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 OpenAccount(

        p\_account\_id  IN NUMBER,

        p\_customer\_id IN NUMBER,

        p\_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\_type, p\_balance, SYSDATE);

        DBMS\_OUTPUT.PUT\_LINE('Account opened successfully.');

    EXCEPTION

        WHEN DUP\_VAL\_ON\_INDEX THEN

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

        WHEN OTHERS THEN

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

    END;

    PROCEDURE CloseAccount(

        p\_account\_id IN NUMBER

    ) IS

    BEGIN

        DELETE FROM Accounts

        WHERE AccountID = p\_account\_id;

        IF SQL%ROWCOUNT = 0 THEN

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

        ELSE

            DBMS\_OUTPUT.PUT\_LINE('Account closed successfully.');

        END IF;

    END;

    FUNCTION GetTotalBalance(p\_customer\_id IN NUMBER) RETURN NUMBER IS

        v\_total NUMBER;

    BEGIN

        SELECT SUM(Balance) INTO v\_total

        FROM Accounts

        WHERE CustomerID = p\_customer\_id;

        RETURN NVL(v\_total, 0);

    END;

END AccountOperations;

/

--open account

BEGIN

    AccountOperations.OpenAccount(5, 1, 'Savings', 2000);

END;

/

--close account

BEGIN

    AccountOperations.CloseAccount(5);

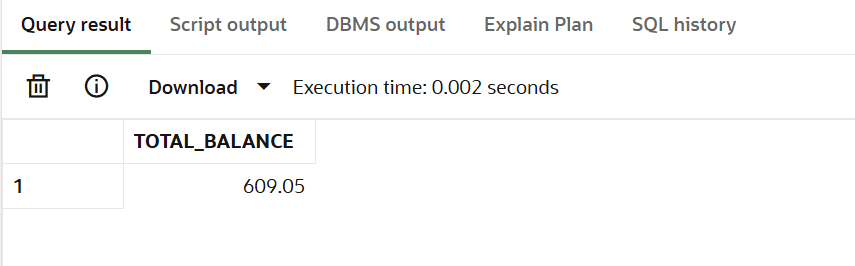
END;

/

--total balance

SELECT AccountOperations.GetTotalBalance(1) AS Total\_Balance FROM dual;

**Output:**

****

**Schema to be Created**

*CREATE TABLE Customers (*

*CustomerID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*DOB DATE,*

*Balance NUMBER,*

*LastModified DATE*

*);*

*CREATE TABLE Accounts (*

*AccountID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*AccountType VARCHAR2(20),*

*Balance NUMBER,*

*LastModified DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Transactions (*

*TransactionID NUMBER PRIMARY KEY,*

*AccountID NUMBER,*

*TransactionDate DATE,*

*Amount NUMBER,*

*TransactionType VARCHAR2(10),*

*FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)*

*);*

*CREATE TABLE Loans (*

*LoanID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*LoanAmount NUMBER,*

*InterestRate NUMBER,*

*StartDate DATE,*

*EndDate DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Employees (*

*EmployeeID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*Position VARCHAR2(50),*

*Salary NUMBER,*

*Department VARCHAR2(50),*

*HireDate DATE*

*);*

**Example Scripts for Sample Data Insertion**

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

*VALUES (1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);*

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

*VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);*

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

*VALUES (1, 1, 'Savings', 1000, SYSDATE);*

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

*VALUES (2, 2, 'Checking', 1500, SYSDATE);*

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

*VALUES (1, 1, SYSDATE, 200, 'Deposit');*

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

*VALUES (2, 2, SYSDATE, 300, 'Withdrawal');*

*INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)*

*VALUES (1, 1, 5000, 5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));*

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

*VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO\_DATE('2015-06-15', 'YYYY-MM-DD'));*

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

*VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO\_DATE('2017-03-20', 'YYYY-MM-DD'));*