**WEEK-2**

**PL SQL HANDS ON**

**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:**

BEGIN

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

        IF FLOOR(MONTHS\_BETWEEN(SYSDATE, customerloop.DOB) / 12) > 60 THEN

            UPDATE Loans

            SET InterestRate = InterestRate - 1

            WHERE CustomerID = customerloop.CustomerID;

        END IF;

    END LOOP;

END;

**OUTPUT:**  
A screenshot of a computer

AI-generated content may be incorrect.

**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:**

BEGIN

  FOR customer\_record IN (

    SELECT CustomerID, Balance

    FROM Customers

  ) LOOP

    IF customer\_record.Balance > 10000 THEN

      UPDATE Customers

      SET IsVIP = 'TRUE'

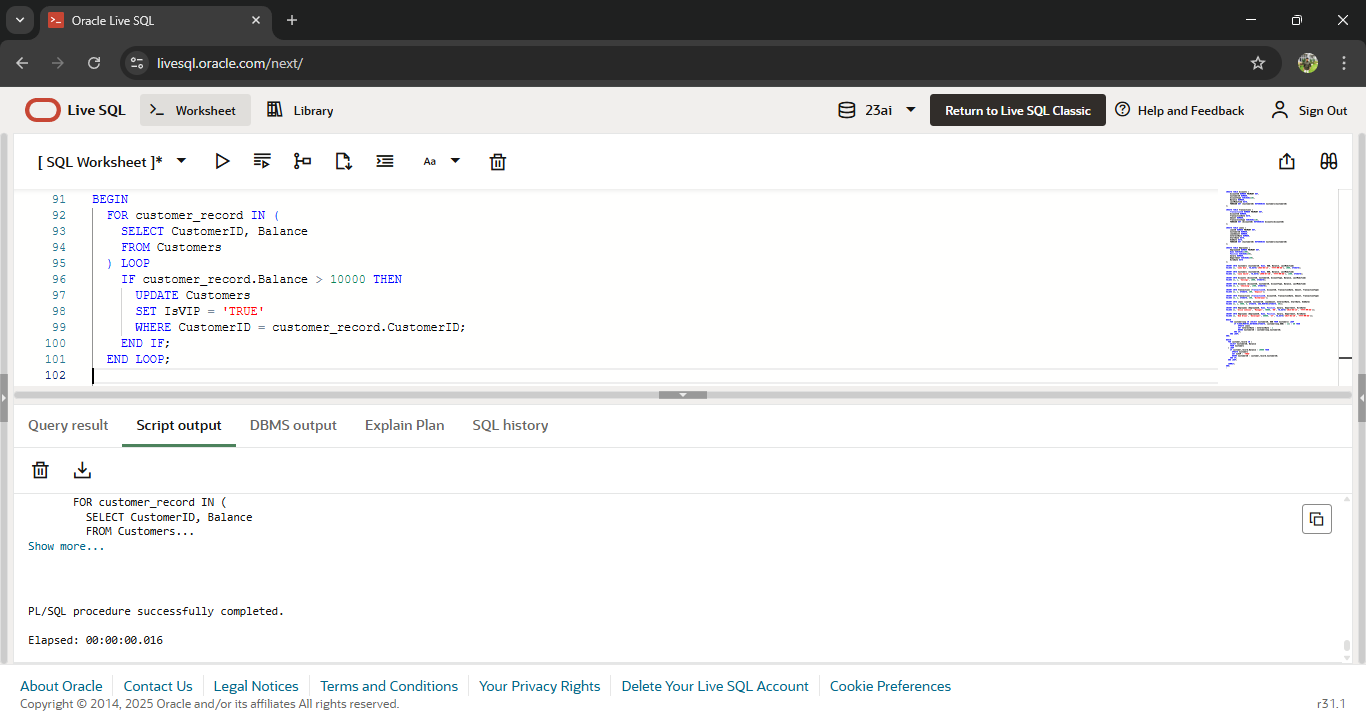
      WHERE CustomerID = customer\_record.CustomerID;

    END IF;

  END LOOP;

  COMMIT;

END;

**OUTPUT:  
**

**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:**BEGIN

  FOR loan\_due\_record IN (

    SELECT c.Name AS customer\_name, l.EndDate AS due\_date

    FROM Loans l

    JOIN Customers c ON l.CustomerID = c.CustomerID

    WHERE l.EndDate <= SYSDATE + 30

  ) LOOP

    DBMS\_OUTPUT.PUT\_LINE(

      'Reminder: Loan for ' || loan\_due\_record.customer\_name ||

      ' is due on ' || TO\_CHAR(loan\_due\_record.due\_date, 'YYYY-MM-DD')

    );

  END LOOP;

END;

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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

BEGIN

    DECLARE

        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 source account.');

        END IF;

        UPDATE Accounts

        SET Balance = Balance - p\_amount

        WHERE AccountID = p\_from\_account\_id;

        UPDATE Accounts

        SET Balance = Balance + p\_amount

        WHERE AccountID = p\_to\_account\_id;

        COMMIT;

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

    EXCEPTION

        WHEN OTHERS THEN

            ROLLBACK;

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

    END;

END;

/

EXEC SafeTransferFunds(1, 2, 100);

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Scenario 2:** Manage errors when updating employee salaries.

**Question:** Write a stored procedure **UpdateSalary** that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.

**CODE:**

CREATE OR REPLACE PROCEDURE UpdateSalary (

    p\_employee\_id IN NUMBER,

    p\_percentage IN NUMBER

)

IS

BEGIN

    UPDATE Employees

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

    WHERE EmployeeID = p\_employee\_id;

    IF SQL%ROWCOUNT = 0 THEN

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

    ELSE

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

        COMMIT;

    END IF;

EXCEPTION

    WHEN OTHERS THEN

        ROLLBACK;

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

END;

/

EXEC UpdateSalary(2, 10);

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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 added successfully: ' || p\_name);

EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

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

    WHEN OTHERS THEN

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

END;

/

EXEC AddNewCustomer(1, 'Ravi Kumar', TO\_DATE('1988-12-01','YYYY-MM-DD'), 5000);

**OUTPUT:  
A screenshot of a computer

AI-generated content may be incorrect.**

**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

    ROLLBACK;

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

END;

/

**OUTPUT:  
A screenshot of a computer

AI-generated content may be incorrect.**

**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\_percent IN NUMBER)

ISBEGIN

  UPDATE Employees

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

  WHERE Department = p\_department;

  IF SQL%ROWCOUNT > 0 THEN

    DBMS\_OUTPUT.PUT\_LINE('Bonus applied to employees in department: ' || p\_department);

    COMMIT;

  ELSE

    DBMS\_OUTPUT.PUT\_LINE('No employees found in department: ' || p\_department);

  END IF;

EXCEPTION

  WHEN OTHERS THEN

    ROLLBACK;

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

END;/

EXEC UpdateEmployeeBonus('IT', 10);

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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 IN NUMBER,

  p\_to\_account IN NUMBER,

  p\_amount IN NUMBER

)

IS

  v\_balance NUMBER;

BEGIN

  SELECT Balance INTO v\_balance

  FROM Accounts

  WHERE AccountID = p\_from\_account

  FOR UPDATE;

  IF v\_balance < p\_amount THEN

    DBMS\_OUTPUT.PUT\_LINE('Insufficient balance in account: ' || p\_from\_account);

    ROLLBACK;

    RETURN;

  END IF;

  UPDATE Accounts

  SET Balance = Balance - p\_amount,

      LastModified = SYSDATE

  WHERE AccountID = p\_from\_account;

  UPDATE Accounts

  SET Balance = Balance + p\_amount,

      LastModified = SYSDATE

  WHERE AccountID = p\_to\_account;

  COMMIT;

  DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount || ' completed from Account ' ||

                        p\_from\_account || ' to Account ' || p\_to\_account);

EXCEPTION

  WHEN OTHERS THEN

    ROLLBACK;

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

END;/

EXEC TransferFunds(1, 2, 500);

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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

BEGIN

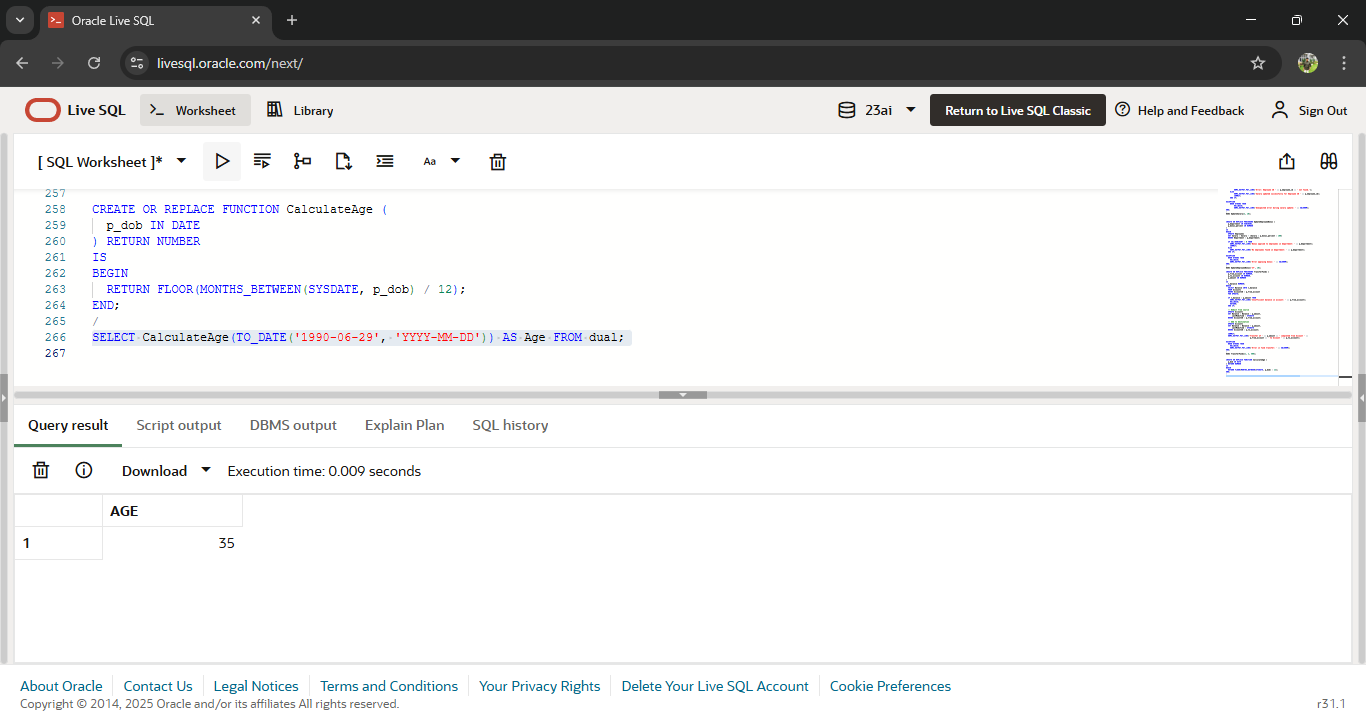
  RETURN FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

END;

/

SELECT CalculateAge(TO\_DATE('1990-06-29', 'YYYY-MM-DD')) AS Age FROM dual;

**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;

  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;

/

SELECT CalculateMonthlyInstallment(100000, 6.5, 5) AS EMI FROM dual;

**OUTPUT:  
A screenshot of a computer

AI-generated content may be incorrect.**

**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;

/

DECLARE

  result BOOLEAN;

BEGIN

  result := HasSufficientBalance(1, 5000);

  IF result THEN

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

  ELSE

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

  END IF;

END;

/

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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;

/**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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 ON NULL AS IDENTITY PRIMARY KEY,

  TransactionID NUMBER,

  AccountID NUMBER,

  ActionType VARCHAR2(20),

  ActionDate DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

  INSERT INTO AuditLog (

    TransactionID, AccountID, ActionType, ActionDate

  ) VALUES (

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

  );

END;

/

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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

  -- Get account balance

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

  -- Rule: Withdrawal must not exceed current balance

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

    RAISE\_APPLICATION\_ERROR(-20001, 'Withdrawal exceeds available balance.');

  END IF;

  -- Rule: Deposit must be positive

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

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

  END IF;

END;

/

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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:**

SET SERVEROUTPUT ON;

DECLARE

  CURSOR transaction\_cursor IS

    SELECT t.TransactionID, t.AccountID, t.TransactionDate, t.Amount, t.TransactionType, c.Name

    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');

  v\_transaction transaction\_cursor%ROWTYPE;

BEGIN

  OPEN transaction\_cursor;

  LOOP

    FETCH transaction\_cursor INTO v\_transaction;

    EXIT WHEN transaction\_cursor%NOTFOUND;

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

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

                         ' | Type: ' || v\_transaction.TransactionType ||

                         ' | Amount: ' || v\_transaction.Amount);

  END LOOP;

  CLOSE transaction\_cursor;

END;

**OUTPUT:  
A screenshot of a computer

AI-generated content may be incorrect.**

**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 account\_cursor IS

    SELECT AccountID, Balance FROM Accounts;

  v\_account account\_cursor%ROWTYPE;

  v\_fee CONSTANT NUMBER := 100; -- Annual fee amount

BEGIN

  OPEN account\_cursor;

  LOOP

    FETCH account\_cursor INTO v\_account;

    EXIT WHEN account\_cursor%NOTFOUND;

    UPDATE Accounts

    SET Balance = Balance - v\_fee,

        LastModified = SYSDATE

    WHERE AccountID = v\_account.AccountID;

    DBMS\_OUTPUT.PUT\_LINE('Annual fee deducted from Account ID: ' || v\_account.AccountID);

  END LOOP;

  CLOSE account\_cursor;

  COMMIT;

END;

/

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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\_loan loan\_cursor%ROWTYPE;

  v\_new\_rate NUMBER;

BEGIN

  OPEN loan\_cursor;

  LOOP

    FETCH loan\_cursor INTO v\_loan;

    EXIT WHEN loan\_cursor%NOTFOUND;

    -- Apply new rate policy

    IF v\_loan.LoanAmount >= 10000 THEN

      v\_new\_rate := 6;

    ELSE

      v\_new\_rate := 5;

    END IF;

    UPDATE Loans

    SET InterestRate = v\_new\_rate

    WHERE LoanID = v\_loan.LoanID;

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

                         ' | Old Rate: ' || v\_loan.InterestRate ||

                         ' | New Rate: ' || v\_new\_rate);

  END LOOP;

  CLOSE loan\_cursor;

  COMMIT;

END;

/

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Exercise 7: Packages**

**Scenario 1:** Group all customer-related procedures and functions into a package.

**Question:** Create a package CustomerManagement with procedures for adding a new customer, updating customer details, and a function to get customer balance**.**

**CODE:**CREATE OR REPLACE PACKAGE CustomerManagement AS

  PROCEDURE AddCustomer(

    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

  );

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

END CustomerManagement;

/

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);

    COMMIT;

  EXCEPTION

    WHEN DUP\_VAL\_ON\_INDEX THEN

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

  END;

  PROCEDURE UpdateCustomerDetails(

    p\_customer\_id IN NUMBER,

    p\_name IN VARCHAR2,

    p\_dob IN DATE

  ) IS

  BEGIN

    UPDATE Customers

    SET Name = p\_name,

        DOB = p\_dob,

        LastModified = SYSDATE

    WHERE CustomerID = p\_customer\_id;

    COMMIT;

  END;

  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;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

      RETURN 0;

  END;

END CustomerManagement;

/

**OUTPUT:**  
A screenshot of a computer

AI-generated content may be incorrect.

**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.

**OUTPUT:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

  PROCEDURE HireEmployee(

    p\_emp\_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 UpdateEmployee(

    p\_emp\_id IN NUMBER,

    p\_salary IN NUMBER,

    p\_department IN VARCHAR2

  );

  FUNCTION CalculateAnnualSalary(p\_emp\_id IN NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

  PROCEDURE HireEmployee(

    p\_emp\_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\_emp\_id, p\_name, p\_position, p\_salary, p\_department, p\_hire\_date);

    COMMIT;

  END;

  PROCEDURE UpdateEmployee(

    p\_emp\_id IN NUMBER,

    p\_salary IN NUMBER,

    p\_department IN VARCHAR2

  ) IS

  BEGIN

    UPDATE Employees

    SET Salary = p\_salary,

        Department = p\_department

    WHERE EmployeeID = p\_emp\_id;

    COMMIT;

  END;

  FUNCTION CalculateAnnualSalary(p\_emp\_id IN NUMBER) RETURN NUMBER IS

    v\_salary NUMBER;

  BEGIN

    SELECT Salary INTO v\_salary FROM Employees WHERE EmployeeID = p\_emp\_id;

    RETURN v\_salary \* 12;

  EXCEPTION

    WHEN NO\_DATA\_FOUND THEN

      RETURN 0;

  END;

END EmployeeManagement;

/

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Scenario 3**: Group all account-related operations into a package.

**Question:** Create a package AccountOperations with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.

**CODE:**

CREATE OR REPLACE PACKAGE AccountOperations AS

  PROCEDURE 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;

/

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);

    COMMIT;

  END;

  PROCEDURE CloseAccount(p\_account\_id IN NUMBER) IS

  BEGIN

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

    COMMIT;

  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;

/

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**