MONICA B (SUPERSET ID - 5008627)

Exercise 1: Control Structures

Scenario 1:-

```
DECLARE
   CURSOR c_customers IS
      SELECT customer_id, loan_interest_rate
      FROM customers
      WHERE age > 60;
   v_customer_id        customers.customer_id%TYPE;
   v_loan_interest_rate customers.loan_interest_rate%TYPE;
BEGIN
   FOR customer_record IN c_customers LOOP
      -- Apply a 1% discount to the current loan interest rate
      v_customer_id := customer_record.customer_id;
      v_loan_interest_rate := customer_record.loan_interest_rate - 1;
      UPDATE customers
      SET loan_interest_rate = v_loan_interest_rate
      WHERE customer_id = v_customer_id;
      DBMS_OUTPUT.PUT_LINE('Discount applied to customer ID: ' || v_customer_id);
   END LOOP;
   COMMIT;
END;
```

Scenario 2:-

```
DECLARE
   CURSOR c_customers IS
      SELECT customer_id, balance
      FROM customers
      WHERE balance > 10000;
   v_customer_id customers.customer_id%TYPE;
BEGIN
   FOR customer_record IN c_customers LOOP
      v_customer_id := customer_record.customer_id;
      -- Set IsVIP flag to TRUE for customers with balance > $10,000
      UPDATE customers
      SET IsVIP = TRUE
      WHERE customer_id = v_customer_id;
      DBMS_OUTPUT.PUT_LINE('Customer ID ' || v_customer_id || ' promoted to VIP status.');
   END LOOP;
   COMMIT;
END;
```

Scenario 3:-

```
DECLARE
   CURSOR c_due_loans IS
      SELECT customer_id, due_date
      FROM loans
```

```
    WHERE due_date BETWEEN SYSDATE AND SYSDATE + 30;
  v_customer_id loans.customer_id%TYPE;
  v_due_date    loans.due_date%TYPE;
BEGIN
  FOR loan_record IN c_due_loans LOOP
    v_customer_id := loan_record.customer_id;
    v_due_date := loan_record.due_date;
    -- Print reminder message for each customer
    DBMS_OUTPUT.PUT_LINE('Reminder: Loan for Customer ID ' || v_customer_id ||
                         ' is due on ' || TO_CHAR(v_due_date, 'DD-MON-YYYY') || '.');
  END LOOP;
END;
```

TIME COMPLEXITY :- O(n)

Exercise 2: Error Handling

Scenario 1:-

```
CREATE PROCEDURE SafeTransferFunds(
  p_from_account_id IN NUMBER,
  p_to_account_id   IN NUMBER,
  p_amount          IN NUMBER
)
IS
  insufficient_funds EXCEPTION;
```

```
BEGIN
    -- Check if there are sufficient funds in the from account
    DECLARE
        v_balance NUMBER;
    BEGIN
        SELECT balance
        INTO v_balance
        FROM accounts
        WHERE account_id = p_from_account_id
        FOR UPDATE;

        IF v_balance < p_amount THEN
            RAISE insufficient_funds;
        END IF;
    END;

    -- Deduct the amount from the from_account
    UPDATE accounts
    SET balance = balance - p_amount
    WHERE account_id = p_from_account_id;

    -- Add the amount to the to_account
    UPDATE accounts
```

```
    SET balance = balance + p_amount
    WHERE account_id = p_to_account_id;

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Transfer successful.');

EXCEPTION
    WHEN insufficient_funds THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Transfer failed: Insufficient funds.');
        -- Log the error
        INSERT INTO transfer_errors (account_id, error_message)
        VALUES (p_from_account_id, 'Insufficient funds during transfer');

    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Transfer failed: ' || SQLERRM);
        -- Log the error
        INSERT INTO transfer_errors (account_id, error_message)
        VALUES (p_from_account_id, SQLERRM);
END;
```

Scenario 2:-

```
CREATE PROCEDURE UpdateSalary(
    p_employee_id IN NUMBER,
    p_percentage  IN NUMBER
)
IS
    employee_not_found EXCEPTION;
BEGIN
    -- Update the salary of the employee
    UPDATE employees
    SET salary = salary * (1 + p_percentage / 100)
    WHERE employee_id = p_employee_id;

    IF SQL%ROWCOUNT = 0 THEN
        RAISE employee_not_found;
    END IF;

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Salary updated successfully.');

EXCEPTION
    WHEN employee_not_found THEN
```

```
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Update failed: Employee not found.');
        -- Log the error
        INSERT INTO salary_update_errors (employee_id, error_message)
        VALUES (p_employee_id, 'Employee not found during salary update');

    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Update failed: ' || SQLERRM);
        -- Log the error
        INSERT INTO salary_update_errors (employee_id, error_message)
        VALUES (p_employee_id, SQLERRM);
END;
```

Scenario 3:-

```
CREATE PROCEDURE AddNewCustomer(
    p_customer_id IN NUMBER,
    p_name        IN VARCHAR2,
p_email       IN VARCHAR2
)
IS
    duplicate_customer EXCEPTION;
BEGIN
```

```
    -- Attempt to insert a new customer
    INSERT INTO customers (customer_id, name, email)
    VALUES (p_customer_id, p_name, p_email);

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Customer added successfully.');

EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Insert failed: Duplicate customer ID.');
        -- Log the error
        INSERT INTO customer_add_errors (customer_id, error_message)
        VALUES (p_customer_id, 'Duplicate customer ID during insertion');

    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Insert failed: ' || SQLERRM);
        -- Log the error
        INSERT INTO customer_add_errors (customer_id, error_message)
        VALUES (p_customer_id, SQLERRM);
END;
```

TIME COMPLEXITY :- O(1)

Exercise 3: Stored Procedures

Scenario 1:

```
CREATE PROCEDURE ProcessMonthlyInterest
IS
   v_interest_rate CONSTANT NUMBER := 0.01; -- 1% interest rate
BEGIN
   UPDATE savings_accounts
   SET balance = balance * (1 + v_interest_rate);
   COMMIT;
   DBMS_OUTPUT.PUT_LINE('Monthly interest has been processed for all savings accounts.');
END;
```

TIME COMPLEXITY :- O(n)

Scenario 2:

```
CREATE PROCEDURE UpdateEmployeeBonus(
   p_department_id IN NUMBER,
   p_bonus_percentage IN NUMBER
)
IS
BEGIN
   UPDATE employees
   SET salary = salary * (1 + p_bonus_percentage / 100)
   WHERE department_id = p_department_id;
```

```
    COMMIT;

    DBMS_OUTPUT.PUT_LINE('Employee bonuses have been updated for department ID ' ||
p_department_id);

END;
```

TIME COMPLEXITY :- O(n)

Scenario 3:

```
CREATE PROCEDURE TransferFunds(

    p_from_account_id IN NUMBER,

    p_to_account_id IN NUMBER,

    p_amount IN NUMBER

)

IS

    insufficient_funds EXCEPTION;

    v_balance NUMBER;

BEGIN

    SELECT balance INTO v_balance

    FROM accounts

    WHERE account_id = p_from_account_id

    FOR UPDATE;

    IF v_balance < p_amount THEN

        RAISE insufficient_funds;

    END IF;

    UPDATE accounts

    SET balance = balance - p_amount
```

```
    WHERE account_id = p_from_account_id;
    UPDATE accounts
    SET balance = balance + p_amount
    WHERE account_id = p_to_account_id;
    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Funds have been successfully transferred.');
EXCEPTION
    WHEN insufficient_funds THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Transfer failed: Insufficient funds.');
        -- Log the error (optional)
        INSERT INTO transfer_errors (account_id, error_message)
        VALUES (p_from_account_id, 'Insufficient funds during transfer');
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Transfer failed: ' || SQLERRM);
        INSERT INTO transfer_errors (account_id, error_message)
        VALUES (p_from_account_id, SQLERRM);
END;
```

TIME COMPLEXITY :- O(1)

Exercise 4: Functions

Scenario 1:

```
CREATE FUNCTION CalculateAge(p_date_of_birth DATE)
RETURN NUMBER
IS
   v_age NUMBER;
BEGIN
   v_age := FLOOR(MONTHS_BETWEEN(SYSDATE, p_date_of_birth) / 12);
   RETURN v_age;
END;
```

TIME COMPLEXITY :- O(1)

Scenario 2:

```
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_number_of_payments NUMBER;
   v_monthly_installment NUMBER;
BEGIN
   -- Convert the annual interest rate to a monthly interest rate
```

```
    v_monthly_interest_rate := p_annual_interest_rate / 12 / 100;
    v_number_of_payments := p_loan_duration_years * 12;
    v_monthly_installment := p_loan_amount * v_monthly_interest_rate /
                        (1 - POWER(1 + v_monthly_interest_rate, -v_number_of_payments));

    RETURN v_monthly_installment;
END;
```

TIME COMPLEXITY :- O(1)

Scenario 3:

```
CREATE 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_number_of_payments NUMBER;
    v_monthly_installment NUMBER;
BEGIN
    -- Convert the annual interest rate to a monthly interest rate
    v_monthly_interest_rate := p_annual_interest_rate / 12 / 100;
    v_number_of_payments := p_loan_duration_years * 12;
```

```
    v_monthly_installment := p_loan_amount * v_monthly_interest_rate /
                             (1 - POWER(1 + v_monthly_interest_rate, -v_number_of_payments));

    RETURN v_monthly_installment;
END;
```

TIME COMPLEXITY :- O(1)

Exercise 5: Triggers

Scenario 1:

```
CREATE TRIGGER UpdateCustomerLastModified
BEFORE UPDATE ON Customers
FOR EACH ROW
BEGIN
    :NEW.LastModified := SYSDATE;
END;
```

Scenario 2:

```
CREATE TRIGGER LogTransaction
AFTER INSERT ON Transactions
FOR EACH ROW
BEGIN
```

```
    INSERT INTO AuditLog (transaction_id, account_id, transaction_date, amount,
transaction_type)

    VALUES (:NEW.transaction_id, :NEW.account_id, :NEW.transaction_date,
:NEW.amount, :NEW.transaction_type);

END;
```

Scenario 3:

```
CREATE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

BEGIN

    IF :NEW.transaction_type = 'WITHDRAWAL' THEN

        DECLARE

            v_balance NUMBER;

        BEGIN

            SELECT balance INTO v_balance

            FROM accounts

            WHERE account_id = :NEW.account_id;

            IF :NEW.amount > v_balance THEN

                RAISE_APPLICATION_ERROR(-20001, 'Insufficient funds for withdrawal.');

            END IF;

        END;

    END IF;
```

```
    IF :NEW.transaction_type = 'DEPOSIT' AND :NEW.amount <= 0 THEN
        RAISE_APPLICATION_ERROR(-20002, 'Deposit amount must be positive.');
    END IF;
END;
```

Exercise 6: Cursors

Scenario 1:

```
DECLARE
    CURSOR cur_MonthlyStatements IS
        SELECT customer_id, SUM(amount) AS total_amount
        FROM Transactions
        WHERE transaction_date BETWEEN TRUNC(SYSDATE, 'MM') AND
LAST_DAY(SYSDATE)
        GROUP BY customer_id;

    v_customer_id  Transactions.customer_id%TYPE;
    v_total_amount Transactions.amount%TYPE;
BEGIN
    OPEN cur_MonthlyStatements;
    LOOP
        FETCH cur_MonthlyStatements INTO v_customer_id, v_total_amount;
```

```
        EXIT WHEN cur_MonthlyStatements%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE('Customer ID: ' || v_customer_id ||
                             ' | Total Transactions for Current Month: ' || v_total_amount);
    END LOOP;

    CLOSE cur_MonthlyStatements;
END;
```

Scenario 2:

```
DECLARE
    CURSOR cur_ApplyAnnualFee IS
        SELECT account_id, balance
        FROM Accounts;
    v_account_id  Accounts.account_id%TYPE;
    v_balance     Accounts.balance%TYPE;
    c_annual_fee CONSTANT NUMBER := 50;  -- Annual fee amount
BEGIN
    OPEN cur_ApplyAnnualFee;
    LOOP
        FETCH cur_ApplyAnnualFee INTO v_account_id, v_balance;
        EXIT WHEN cur_ApplyAnnualFee%NOTFOUND;
        UPDATE Accounts
```

```
        SET balance = balance - c_annual_fee
        WHERE account_id = v_account_id;

        DBMS_OUTPUT.PUT_LINE('Annual fee applied to Account ID: ' || v_account_id);
    END LOOP;
    CLOSE cur_ApplyAnnualFee;
END;
```

Scenario 3:

```
DECLARE
    CURSOR cur_UpdateLoanInterestRates IS
        SELECT loan_id, interest_rate
        FROM Loans;

    v_loan_id       Loans.loan_id%TYPE;
    v_interest_rate Loans.interest_rate%TYPE;
    c_new_interest_rate CONSTANT NUMBER := 0.07;  -- New interest rate (7%)
BEGIN
    OPEN cur_UpdateLoanInterestRates;
    LOOP
        FETCH cur_UpdateLoanInterestRates INTO v_loan_id, v_interest_rate;
        EXIT WHEN cur_UpdateLoanInterestRates%NOTFOUND;
        UPDATE Loans
```

```
        SET interest_rate = c_new_interest_rate

        WHERE loan_id = v_loan_id;


        DBMS_OUTPUT.PUT_LINE('Updated Loan ID: ' || v_loan_id || ' with new interest
rate: ' || c_new_interest_rate);

    END LOOP;


    CLOSE cur_UpdateLoanInterestRates;

END;
```

Exercise 7: Packages

Scenario 1:

```
CREATE CustomerManagement AS

    PROCEDURE AddNewCustomer(p_CustomerID NUMBER, p_Name VARCHAR2,
p_DOB DATE, p_Balance NUMBER) IS

    BEGIN

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

        VALUES (p_CustomerID, p_Name, p_DOB, p_Balance, SYSDATE);

    END AddNewCustomer;


    PROCEDURE UpdateCustomerDetails(p_CustomerID NUMBER, p_Name
VARCHAR2, p_DOB DATE, p_Balance NUMBER) IS

    BEGIN

        UPDATE Customers
```

```
    SET Name = p_Name, DOB = p_DOB, Balance = p_Balance, LastModified =
SYSDATE

    WHERE CustomerID = p_CustomerID;

  END UpdateCustomerDetails;


  FUNCTION GetCustomerBalance(p_CustomerID NUMBER) RETURN NUMBER IS

    v_Balance NUMBER;

  BEGIN

    SELECT Balance INTO v_Balance FROM Customers WHERE CustomerID =
p_CustomerID;

    RETURN v_Balance;

  END GetCustomerBalance;

END CustomerManagement;
```

Scenario 2:

```
CREATE EmployeeManagement AS


  PROCEDURE HireNewEmployee(p_EmployeeID NUMBER, p_Name VARCHAR2,
p_Position VARCHAR2, p_Salary NUMBER, p_Department VARCHAR2) IS

  BEGIN

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

    VALUES (p_EmployeeID, p_Name, p_Position, p_Salary, p_Department,
SYSDATE);

  END HireNewEmployee;
```

```
    PROCEDURE UpdateEmployeeDetails(p_EmployeeID NUMBER, p_Name
VARCHAR2, p_Position VARCHAR2, p_Salary NUMBER, p_Department VARCHAR2)
IS

    BEGIN

        UPDATE Employees

        SET Name = p_Name, Position = p_Position, Salary = p_Salary, Department =
p_Department

        WHERE EmployeeID = p_EmployeeID;

    END UpdateEmployeeDetails;



    FUNCTION CalculateAnnualSalary(p_EmployeeID NUMBER) RETURN NUMBER IS

        v_AnnualSalary NUMBER;

    BEGIN

        SELECT Salary * 12 INTO v_AnnualSalary FROM Employees WHERE
EmployeeID = p_EmployeeID;

        RETURN v_AnnualSalary;

    END CalculateAnnualSalary;

END EmployeeManagement;
```

Scenario 3:

```
CREATE AccountOperations AS



    PROCEDURE OpenNewAccount(p_AccountID NUMBER, p_CustomerID NUMBER,
p_AccountType VARCHAR2, p_Balance NUMBER) IS
```

```
    BEGIN
        INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance,
LastModified)
        VALUES (p_AccountID, p_CustomerID, p_AccountType, p_Balance, SYSDATE);
    END OpenNewAccount;
    PROCEDURE CloseAccount(p_AccountID NUMBER) IS
    BEGIN
        DELETE FROM Accounts WHERE AccountID = p_AccountID;
    END CloseAccount;
    FUNCTION GetTotalBalance(p_CustomerID NUMBER) RETURN NUMBER IS
        v_TotalBalance NUMBER;
    BEGIN
        SELECT SUM(Balance) INTO v_TotalBalance FROM Accounts WHERE
CustomerID = p_CustomerID;
        RETURN v_TotalBalance;
    END GetTotalBalance;
END AccountOperations;
```