Exercise 1: Control Structures

Scenario 1: Apply a 1% Discount to Loan Interest Rates for Customers Above 60 Years Old

PL/SQL BLOCK:

```
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_rec IN c_customers LOOP
      v_customer_id := customer_rec.customer_id;
      v_loan_interest_rate := customer_rec.loan_interest_rate;

      UPDATE customers
      SET loan_interest_rate = v_loan_interest_rate - (v_loan_interest_rate * 0.01)
      WHERE customer_id = v_customer_id;

      DBMS_OUTPUT.PUT_LINE('Applied 1% discount to customer ' || v_customer_id);
   END LOOP;
   COMMIT;
END;
```

Scenario 2 : Set IsVIP Flag to TRUE for Customers with Balance Over $10,000

```
DECLARE
   CURSOR c_customers IS
      SELECT customer_id, balance
      FROM customers
      WHERE balance > 10000;

   v_customer_id customers.customer_id%TYPE;
   v_balance customers.balance%TYPE;
BEGIN
   FOR customer_rec IN c_customers LOOP
      v_customer_id := customer_rec.customer_id;
      v_balance := customer_rec.balance;
```

```
        UPDATE customers
        SET is_vip = TRUE
        WHERE customer_id = v_customer_id;

        DBMS_OUTPUT.PUT_LINE('Promoted customer ' || v_customer_id || ' to VIP status');
    END LOOP;
    COMMIT;
END;
```

Scenario 3 : Send Reminders to Customers Whose Loans Are Due Within the Next 30 Days

```
DECLARE
    CURSOR c_loans IS
        SELECT loan_id, customer_id, due_date
        FROM loans
        WHERE due_date BETWEEN SYSDATE AND SYSDATE + 30;

    v_loan_id loans.loan_id%TYPE;
    v_customer_id loans.customer_id%TYPE;
    v_due_date loans.due_date%TYPE;
BEGIN
    FOR loan_rec IN c_loans LOOP
        v_loan_id := loan_rec.loan_id;
        v_customer_id := loan_rec.customer_id;
        v_due_date := loan_rec.due_date;

        DBMS_OUTPUT.PUT_LINE('Reminder: Loan ' || v_loan_id || ' for customer ' ||
v_customer_id || ' is due on ' || TO_CHAR(v_due_date, 'DD-MON-YYYY'));
    END LOOP;
END;
```

Exercise 2: Error Handling

1)SafeTransferFunds Stored Procedure:

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds(
    p_from_account_id IN NUMBER,
    p_to_account_id IN NUMBER,
    p_amount IN NUMBER
) IS
    insufficient_funds EXCEPTION;
    v_from_balance NUMBER;
BEGIN
    -- Check balance of the from account
    SELECT balance INTO v_from_balance FROM accounts WHERE account_id =
p_from_account_id FOR UPDATE;

    -- Raise exception if insufficient funds
    IF v_from_balance < p_amount THEN
        RAISE insufficient_funds;
    END IF;

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

    -- Add amount to the to account
    UPDATE accounts
    SET balance = balance + p_amount
    WHERE account_id = p_to_account_id;

    -- Commit transaction
    COMMIT;

    DBMS_OUTPUT.PUT_LINE('Funds transferred successfully.');

EXCEPTION
    WHEN insufficient_funds THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in the from account.');
        -- Log error message
        INSERT INTO error_log (error_message) VALUES ('Insufficient funds during transfer.');
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Error: An unexpected error occurred.');
        -- Log error message
        INSERT INTO error_log (error_message) VALUES (SQLERRM);
END SafeTransferFunds;
```

Scenario 2:UpdateSalary Stored Procedure

```
CREATE OR REPLACE PROCEDURE UpdateSalary(
    p_employee_id IN NUMBER,
    p_percentage IN NUMBER
) IS
    employee_not_found EXCEPTION;
    v_count NUMBER;
BEGIN
    -- Check if the employee exists
    SELECT COUNT(*) INTO v_count FROM employees WHERE employee_id =
p_employee_id;

    -- Raise exception if employee does not exist
    IF v_count = 0 THEN
       RAISE employee_not_found;
    END IF;

    -- Update the salary
    UPDATE employees
    SET salary = salary * (1 + p_percentage / 100)
    WHERE employee_id = p_employee_id;

    -- Commit transaction
    COMMIT;

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

EXCEPTION
    WHEN employee_not_found THEN
       DBMS_OUTPUT.PUT_LINE('Error: Employee not found.');
       -- Log error message
       INSERT INTO error_log (error_message) VALUES ('Employee not found during salary
update.');
    WHEN OTHERS THEN
       DBMS_OUTPUT.PUT_LINE('Error: An unexpected error occurred.');
       -- Log error message
       INSERT INTO error_log (error_message) VALUES (SQLERRM);
END UpdateSalary;
```

Scenario 3 : AddNewCustomer Stored Procedure

```
CREATE OR REPLACE PROCEDURE AddNewCustomer(
   p_customer_id IN NUMBER,
   p_name IN VARCHAR2,
   p_age IN NUMBER,
   p_balance IN NUMBER
) IS
   customer_exists EXCEPTION;
```

```
    v_count NUMBER;
BEGIN
    -- Check if the customer ID already exists
    SELECT COUNT(*) INTO v_count FROM customers WHERE customer_id = p_customer_id;

    -- Raise exception if customer already exists
    IF v_count > 0 THEN
        RAISE customer_exists;
    END IF;

    -- Insert new customer
    INSERT INTO customers (customer_id, name, age, balance)
    VALUES (p_customer_id, p_name, p_age, p_balance);

    -- Commit transaction
    COMMIT;

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

EXCEPTION
    WHEN customer_exists THEN
        DBMS_OUTPUT.PUT_LINE('Error: Customer ID already exists.');
        -- Log error message
        INSERT INTO error_log (error_message) VALUES ('Customer ID already exists during
customer addition.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: An unexpected error occurred.');
        -- Log error message
        INSERT INTO error_log (error_message) VALUES (SQLERRM);
END AddNewCustomer;
```

Exercise 3: Stored Procedures:

Scenario 1: ProcessMonthlyInterest Stored Procedure

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS
BEGIN
    UPDATE savings_accounts
    SET balance = balance * 1.01;

    COMMIT;

    DBMS_OUTPUT.PUT_LINE('Monthly interest processed for all savings accounts.');
EXCEPTION
```

```
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: An unexpected error occurred during interest
processing.');
        -- Log error message
        INSERT INTO error_log (error_message) VALUES (SQLERRM);
END ProcessMonthlyInterest;
```

Scenario 2 : UpdateEmployeeBonus Stored Procedure

```
CREATE OR REPLACE 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('Bonus updated for employees in department ' ||
p_department_id);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: An unexpected error occurred during bonus update.');
        -- Log error message
        INSERT INTO error_log (error_message) VALUES (SQLERRM);
END UpdateEmployeeBonus;
```

Scenario 3 : TransferFunds Stored Procedure

```
CREATE OR REPLACE PROCEDURE TransferFunds(
    p_from_account_id IN NUMBER,
    p_to_account_id IN NUMBER,
    p_amount IN NUMBER
) IS
    insufficient_funds EXCEPTION;
    v_from_balance NUMBER;
BEGIN
    -- Check balance of the from account
    SELECT balance INTO v_from_balance FROM accounts WHERE account_id =
p_from_account_id FOR UPDATE;

    -- Raise exception if insufficient funds
    IF v_from_balance < p_amount THEN
        RAISE insufficient_funds;
    END IF;

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

    -- Add amount to the to account
    UPDATE accounts
    SET balance = balance + p_amount
    WHERE account_id = p_to_account_id;

    -- Commit transaction
    COMMIT;

    DBMS_OUTPUT.PUT_LINE('Funds transferred successfully from account ' ||
p_from_account_id || ' to account ' || p_to_account_id);

EXCEPTION
    WHEN insufficient_funds THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in the from account.');
        -- Log error message
        INSERT INTO error_log (error_message) VALUES ('Insufficient funds during transfer.');
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Error: An unexpected error occurred during fund transfer.');
        -- Log error message
        INSERT INTO error_log (error_message) VALUES (SQLERRM);
END TransferFunds;
```

Exercise 4: Functions

Scenario 1: CalculateAge Function

```
CREATE OR REPLACE FUNCTION CalculateAge(
    p_dob DATE
) RETURN NUMBER IS
    v_age NUMBER;
BEGIN
    -- Calculate age
    v_age := FLOOR(MONTHS_BETWEEN(SYSDATE, p_dob) / 12);
    RETURN v_age;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: An unexpected error occurred while calculating age.');
        -- Log error message
        INSERT INTO error_log (error_message) VALUES (SQLERRM);
        RETURN NULL;
END CalculateAge;
```

Scenario 2 : CalculateMonthlyInstallment Function

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(
    p_loan_amount NUMBER,
    p_annual_interest_rate NUMBER,
    p_loan_duration_years NUMBER
) RETURN NUMBER IS
    v_monthly_installment NUMBER;
    v_monthly_rate NUMBER;
    v_number_of_months NUMBER;
BEGIN
    -- Calculate the monthly interest rate
    v_monthly_rate := p_annual_interest_rate / 12 / 100;
    -- Calculate the total number of monthly payments
    v_number_of_months := p_loan_duration_years * 12;

    -- Calculate the monthly installment using the formula
    IF v_monthly_rate > 0 THEN
        v_monthly_installment := p_loan_amount * (v_monthly_rate * POWER(1 + v_monthly_rate,
v_number_of_months)) / (POWER(1 + v_monthly_rate, v_number_of_months) - 1);
    ELSE
        v_monthly_installment := p_loan_amount / v_number_of_months;
    END IF;

    RETURN v_monthly_installment;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: An unexpected error occurred while calculating the
monthly installment.');
        -- Log error message
        INSERT INTO error_log (error_message) VALUES (SQLERRM);
        RETURN NULL;
END CalculateMonthlyInstallment;
```

Scenario 3 : HasSufficientBalance Function

```
CREATE OR REPLACE FUNCTION HasSufficientBalance(
   p_account_id NUMBER,
   p_amount NUMBER
) RETURN BOOLEAN IS
   v_balance NUMBER;
BEGIN
   -- Fetch the balance of the account
   SELECT balance INTO v_balance FROM accounts WHERE account_id = p_account_id;

   -- Check if the balance is sufficient
   IF v_balance >= p_amount THEN
      RETURN TRUE;
   ELSE
      RETURN FALSE;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Error: Account not found.');
      -- Log error message
      INSERT INTO error_log (error_message) VALUES ('Account not found during balance
check.');
      RETURN FALSE;
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: An unexpected error occurred while checking
balance.');
      -- Log error message
      INSERT INTO error_log (error_message) VALUES (SQLERRM);
      RETURN FALSE;
END HasSufficientBalance;
```

Exercise 5: Triggers

Scenario 1 : Automatically Update LastModified Column

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

Scenario 2 : Maintain an Audit Log for Transactions

```
CREATE OR REPLACE TRIGGER LogTransaction
AFTER INSERT ON Transactions
FOR EACH ROW
BEGIN
  INSERT INTO AuditLog (transaction_id, account_id, transaction_type, amount,
transaction_date)
  VALUES (:NEW.transaction_id, :NEW.account_id, :NEW.transaction_type, :NEW.amount,
:NEW.transaction_date);
END LogTransaction;
```

Scenario 3 : Enforce Business Rules on Deposits and Withdrawals

```
CREATE OR REPLACE TRIGGER CheckTransactionRules
BEFORE INSERT ON Transactions
FOR EACH ROW
DECLARE
  v_balance NUMBER;
BEGIN
  -- Check if the transaction is a withdrawal and if it exceeds the account balance
  IF :NEW.transaction_type = 'WITHDRAWAL' THEN
    SELECT balance INTO v_balance FROM accounts WHERE account_id =
:NEW.account_id;
    IF v_balance < :NEW.amount THEN
      RAISE_APPLICATION_ERROR(-20001, 'Withdrawal amount exceeds account
balance.');
    END IF;

  -- Check if the transaction is a deposit and if the amount is positive
  ELSIF :NEW.transaction_type = 'DEPOSIT' THEN
    IF :NEW.amount <= 0 THEN
      RAISE_APPLICATION_ERROR(-20002, 'Deposit amount must be positive.');
    END IF;
  END IF;
END CheckTransactionRules;
```

AuditLog
SQL:

```
CREATE TABLE AuditLog (
  log_id NUMBER GENERATED BY DEFAULT AS IDENTITY,
  transaction_id NUMBER,
  account_id NUMBER,
  transaction_type VARCHAR2(50),
  amount NUMBER,
  transaction_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (log_id)
);
```

Exercise 6: Cursors

Scenario 1 : Generate Monthly Statements for All Customers

```
DECLARE
    CURSOR transactions_cursor IS
        SELECT customer_id, transaction_id, transaction_type, amount, transaction_date
        FROM Transactions
        WHERE EXTRACT(MONTH FROM transaction_date) = EXTRACT(MONTH FROM
SYSDATE)
        AND EXTRACT(YEAR FROM transaction_date) = EXTRACT(YEAR FROM SYSDATE);

    v_customer_id Transactions.customer_id%TYPE;
    v_transaction_id Transactions.transaction_id%TYPE;
    v_transaction_type Transactions.transaction_type%TYPE;
    v_amount Transactions.amount%TYPE;
    v_transaction_date Transactions.transaction_date%TYPE;
BEGIN
    OPEN transactions_cursor;
    LOOP
        FETCH transactions_cursor INTO v_customer_id, v_transaction_id, v_transaction_type,
v_amount, v_transaction_date;
        EXIT WHEN transactions_cursor%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE('Customer ID: ' || v_customer_id);
        DBMS_OUTPUT.PUT_LINE('Transaction ID: ' || v_transaction_id);
        DBMS_OUTPUT.PUT_LINE('Transaction Type: ' || v_transaction_type);
        DBMS_OUTPUT.PUT_LINE('Amount: ' || v_amount);
        DBMS_OUTPUT.PUT_LINE('Transaction Date: ' || v_transaction_date);
        DBMS_OUTPUT.PUT_LINE('-------------------------');
    END LOOP;
    CLOSE transactions_cursor;
END;
```

Scenario 2 : Apply Annual Fee to All Accounts

```
DECLARE
```

```
    CURSOR accounts_cursor IS
        SELECT account_id, balance
        FROM Accounts;

    v_account_id Accounts.account_id%TYPE;
    v_balance Accounts.balance%TYPE;
    v_annual_fee CONSTANT NUMBER := 50; -- Assuming the annual fee is 50 units
BEGIN
    OPEN accounts_cursor;
    LOOP
        FETCH accounts_cursor INTO v_account_id, v_balance;
        EXIT WHEN accounts_cursor%NOTFOUND;

        -- Deduct the annual fee from the account balance
        UPDATE Accounts
        SET balance = balance - v_annual_fee
        WHERE account_id = v_account_id;

        DBMS_OUTPUT.PUT_LINE('Applied annual fee to Account ID: ' || v_account_id || ', New
Balance: ' || (v_balance - v_annual_fee));
    END LOOP;
    CLOSE accounts_cursor;

    COMMIT;
END;
```

Scenario 3 : Update the Interest Rate for All Loans Based on a New Policy

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

    v_loan_id Loans.loan_id%TYPE;
    v_interest_rate Loans.interest_rate%TYPE;
    v_new_interest_rate NUMBER;
BEGIN
    OPEN loans_cursor;
    LOOP
        FETCH loans_cursor INTO v_loan_id, v_interest_rate;
        EXIT WHEN loans_cursor%NOTFOUND;

        -- Assuming a new policy that increases the interest rate by 0.5%
        v_new_interest_rate := v_interest_rate + 0.5;
        -- Update the loan with the new interest rate
        UPDATE Loans
        SET interest_rate = v_new_interest_rate
        WHERE loan_id = v_loan_id;

        DBMS_OUTPUT.PUT_LINE('Updated Loan ID: ' || v_loan_id || ', New Interest Rate: ' ||
v_new_interest_rate);
    END LOOP;
    CLOSE loans_cursor;

    COMMIT;
END;
```

Exercise 7: Packages

Scenario 1 : CustomerManagement Package

Package specification:

```
CREATE OR REPLACE PACKAGE CustomerManagement AS
    PROCEDURE AddCustomer(p_customer_id NUMBER, p_name VARCHAR2, p_dob DATE,
p_address VARCHAR2);
    PROCEDURE UpdateCustomerDetails(p_customer_id NUMBER, p_name VARCHAR2,
p_dob DATE, p_address VARCHAR2);
    FUNCTION GetCustomerBalance(p_customer_id NUMBER) RETURN NUMBER;
END CustomerManagement;
```

Package body:

```
CREATE OR REPLACE PACKAGE BODY CustomerManagement AS
    PROCEDURE AddCustomer(p_customer_id NUMBER, p_name VARCHAR2, p_dob DATE,
p_address VARCHAR2) IS
    BEGIN
        INSERT INTO Customers (customer_id, name, dob, address)
        VALUES (p_customer_id, p_name, p_dob, p_address);
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error adding customer: ' || SQLERRM);
    END AddCustomer;

    PROCEDURE UpdateCustomerDetails(p_customer_id NUMBER, p_name VARCHAR2,
p_dob DATE, p_address VARCHAR2) IS
    BEGIN
```

```
        UPDATE Customers
        SET name = p_name, dob = p_dob, address = p_address
        WHERE customer_id = p_customer_id;
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error updating customer details: ' || SQLERRM);
    END UpdateCustomerDetails;

    FUNCTION GetCustomerBalance(p_customer_id NUMBER) RETURN NUMBER IS
        v_balance NUMBER;
    BEGIN
        SELECT SUM(balance) INTO v_balance
        FROM Accounts
        WHERE customer_id = p_customer_id;
        RETURN v_balance;
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error getting customer balance: ' || SQLERRM);
            RETURN NULL;
    END GetCustomerBalance;
END CustomerManagement;
```

Scenario 2 : EmployeeManagement Package

Package specification:

```
CREATE OR REPLACE PACKAGE EmployeeManagement AS
    PROCEDURE HireEmployee(p_employee_id NUMBER, p_name VARCHAR2, p_position
VARCHAR2, p_salary NUMBER);
    PROCEDURE UpdateEmployeeDetails(p_employee_id NUMBER, p_name VARCHAR2,
p_position VARCHAR2, p_salary NUMBER);
    FUNCTION CalculateAnnualSalary(p_employee_id NUMBER) RETURN NUMBER;
END EmployeeManagement;
```

Package Body:

```
CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS
    PROCEDURE HireEmployee(p_employee_id NUMBER, p_name VARCHAR2, p_position
VARCHAR2, p_salary NUMBER) IS
    BEGIN
        INSERT INTO Employees (employee_id, name, position, salary)
        VALUES (p_employee_id, p_name, p_position, p_salary);
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error hiring employee: ' || SQLERRM);
    END HireEmployee;
```

```
   PROCEDURE UpdateEmployeeDetails(p_employee_id NUMBER, p_name VARCHAR2,
p_position VARCHAR2, p_salary NUMBER) IS
   BEGIN
      UPDATE Employees
      SET name = p_name, position = p_position, salary = p_salary
      WHERE employee_id = p_employee_id;
   EXCEPTION
      WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('Error updating employee details: ' || SQLERRM);
   END UpdateEmployeeDetails;

   FUNCTION CalculateAnnualSalary(p_employee_id NUMBER) RETURN NUMBER IS
      v_salary NUMBER;
   BEGIN
      SELECT salary * 12 INTO v_salary
      FROM Employees
      WHERE employee_id = p_employee_id;
      RETURN v_salary;
   EXCEPTION
      WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('Error calculating annual salary: ' || SQLERRM);
         RETURN NULL;
   END CalculateAnnualSalary;
END EmployeeManagement;
```

Scenario 3 : AccountOperations Package

Packagespecification:

```
CREATE OR REPLACE PACKAGE AccountOperations AS
   PROCEDURE OpenAccount(p_account_id NUMBER, p_customer_id NUMBER, p_balance
NUMBER);
   PROCEDURE CloseAccount(p_account_id NUMBER);
   FUNCTION GetTotalBalance(p_customer_id NUMBER) RETURN NUMBER;
END AccountOperations;
```

Package body:

```
CREATE OR REPLACE PACKAGE BODY AccountOperations AS
   PROCEDURE OpenAccount(p_account_id NUMBER, p_customer_id NUMBER, p_balance
NUMBER) IS
   BEGIN
      INSERT INTO Accounts (account_id, customer_id, balance)
      VALUES (p_account_id, p_customer_id, p_balance);
```

```
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error opening account: ' || SQLERRM);
    END OpenAccount;

    PROCEDURE CloseAccount(p_account_id NUMBER) IS
    BEGIN
        DELETE FROM Accounts
        WHERE account_id = p_account_id;
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error closing account: ' || SQLERRM);
    END CloseAccount;

    FUNCTION GetTotalBalance(p_customer_id NUMBER) RETURN NUMBER IS
        v_total_balance NUMBER;
    BEGIN
        SELECT SUM(balance) INTO v_total_balance
        FROM Accounts
        WHERE customer_id = p_customer_id;
        RETURN v_total_balance;
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error getting total balance: ' || SQLERRM);
            RETURN NULL;
    END GetTotalBalance;
END AccountOperations;
```