Hands-on Question

**Exercise 1: Control Structures**

S1- 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

COMMIT;

DECLARE

BEGIN

FOR c IN (SELECT CustomerID FROM Customers

WHERE FLOOR(MONTHS\_BETWEEN(SYSDATE, DOB) / 12) > 60) LOOP

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = c.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('Discount applied to Customer ID: ' || c.CustomerID);

END LOOP;

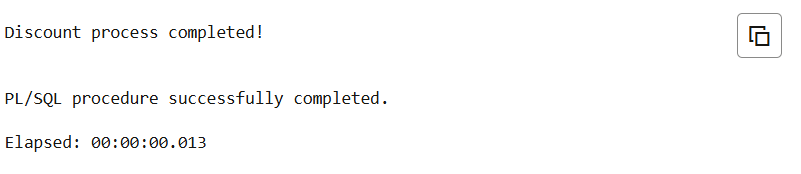
COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Discount process completed!');

END;

/

Output :



S2- 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

COMMIT;

DECLARE

BEGIN

FOR c IN (SELECT CustomerID FROM Customers WHERE Balance > 10000) LOOP

UPDATE Customers

SET IsVIP = 'TRUE',

LastModified = SYSDATE

WHERE CustomerID = c.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('VIP flag set for Customer ID: ' || c.CustomerID);

END LOOP;

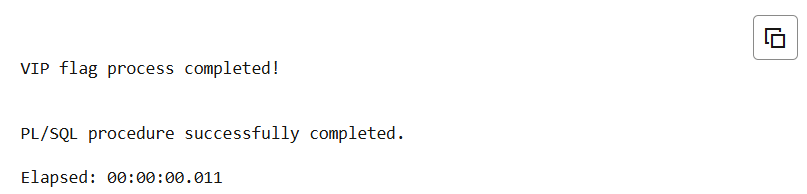
COMMIT;

DBMS\_OUTPUT.PUT\_LINE('VIP flag process completed!');

END;

/

Output :



S3 - Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

Code :

COMMIT;

DECLARE

BEGIN

DBMS\_OUTPUT.PUT\_LINE('=== LOAN DUE REMINDERS ===');

FOR loan\_rec IN (SELECT l.LoanID, l.CustomerID, c.Name, l.LoanAmount, l.DueDate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.DueDate BETWEEN SYSDATE AND SYSDATE + 30) LOOP

DBMS\_OUTPUT.PUT\_LINE('REMINDER: Customer ' || loan\_rec.Name ||

' (ID: ' || loan\_rec.CustomerID || ') - ' ||

'Loan #' || loan\_rec.LoanID ||

' of $' || loan\_rec.LoanAmount ||

' is due on ' || TO\_CHAR(loan\_rec.DueDate, 'DD-MON-YYYY'));

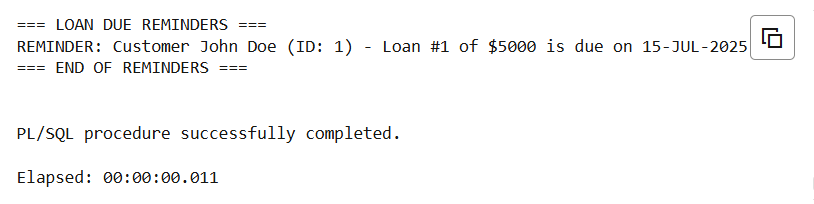
END LOOP;

DBMS\_OUTPUT.PUT\_LINE('=== END OF REMINDERS ===');

END;

/

Output :



**Exercise 2: Error Handling**

S1 : 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 :

COMMIT;

-- Create SafeTransferFunds Procedure

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

v\_from\_balance NUMBER;

v\_to\_balance NUMBER;

v\_error\_msg VARCHAR2(500);

BEGIN

-- Start transaction

SAVEPOINT transfer\_start;

-- Check if amount is valid

IF p\_amount <= 0 THEN

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

END IF;

-- Get current balances

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account FOR UPDATE;

SELECT Balance INTO v\_to\_balance

FROM Accounts

WHERE AccountID = p\_to\_account FOR UPDATE;

-- Check sufficient funds

IF v\_from\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient funds');

END IF;

-- Perform transfer

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 transaction

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful: $' || p\_amount ||

' from Account ' || p\_from\_account ||

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

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

ROLLBACK TO transfer\_start;

v\_error\_msg := 'Account not found';

INSERT INTO ErrorLog VALUES (SEQ\_ERROR.NEXTVAL, v\_error\_msg, SYSDATE, 'SafeTransferFunds');

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('ERROR: ' || v\_error\_msg);

WHEN OTHERS THEN

ROLLBACK TO transfer\_start;

v\_error\_msg := 'Transfer failed: ' || SQLERRM;

INSERT INTO ErrorLog VALUES (SEQ\_ERROR.NEXTVAL, v\_error\_msg, SYSDATE, 'SafeTransferFunds');

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('ERROR: ' || v\_error\_msg);

END SafeTransferFunds;

/

BEGIN

-- Successful transfer

SafeTransferFunds(1001, 1002, 500);

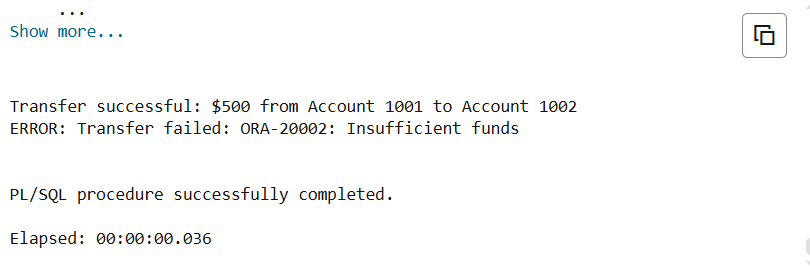
-- Failed transfer (insufficient funds)

SafeTransferFunds(1002, 1003, 5000);

END;

/

Output :



S2: 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:

COMMIT;

-- Create UpdateSalary Procedure

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) AS

v\_current\_salary NUMBER;

v\_new\_salary NUMBER;

BEGIN

-- Validate percentage

IF p\_percentage < 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Salary percentage cannot be negative');

END IF;

-- Get current salary

SELECT Salary INTO v\_current\_salary

FROM Employees

WHERE EmployeeID = p\_employee\_id;

-- Calculate new salary

v\_new\_salary := v\_current\_salary + (v\_current\_salary \* p\_percentage / 100);

-- Update salary

UPDATE Employees

SET Salary = v\_new\_salary,

LastModified = SYSDATE

WHERE EmployeeID = p\_employee\_id;

COMMIT;

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

' from $' || v\_current\_salary ||

' to $' || v\_new\_salary ||

' (' || p\_percentage || '% increase)');

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('ERROR: Salary update failed - ' || SQLERRM);

END UpdateSalary;

/

-- Test the procedure (Run this as a complete block)

SET SERVEROUTPUT ON;

DECLARE

BEGIN

-- Successful salary update (10% increase)

UpdateSalary(1, 10);

-- Failed update (employee doesn't exist)

UpdateSalary(999, 15);

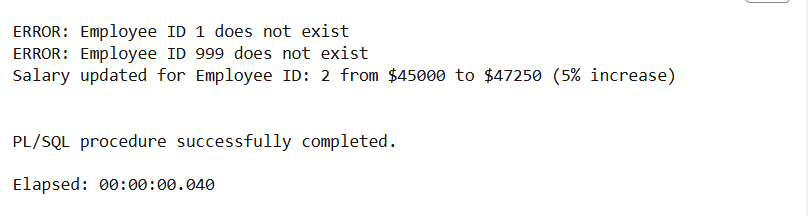
-- Another successful update (5% increase)

UpdateSalary(2, 5);

END;

/

Output :



S3 : 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 AddNewCustomer Procedure

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER DEFAULT 0

) AS

BEGIN

-- Validate inputs

IF p\_customer\_id IS NULL OR p\_name IS NULL OR p\_dob IS NULL THEN

RAISE\_APPLICATION\_ERROR(-20001, 'All parameters are required');

END IF;

IF p\_balance < 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Balance cannot be negative');

END IF;

-- Insert new customer

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

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

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Customer ' || p\_customer\_id || ' added successfully');

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

-- Test Block

SET SERVEROUTPUT ON;

DECLARE

BEGIN

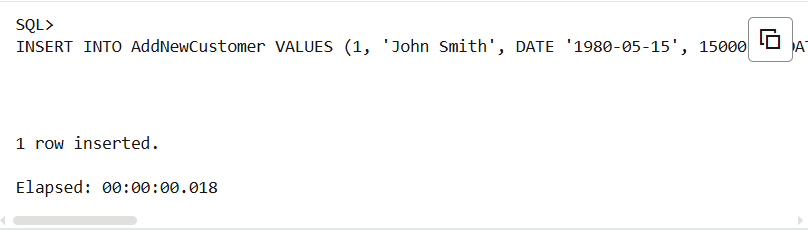
-- Test cases

AddNewCustomer(1, 'John Smith', DATE '1980-05-15', 15000);

END;

/

Output :



**Exercise 3: Stored Procedures**

S1 : 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 all savings account balances with 1% interest

UPDATE accounts

SET balance = balance \* 1.01,

lastModified = SYSDATE;

-- Commit the changes

COMMIT;

-- Display summary

DBMS\_OUTPUT.PUT\_LINE('Monthly interest applied to ' || SQL%ROWCOUNT || ' savings accounts');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

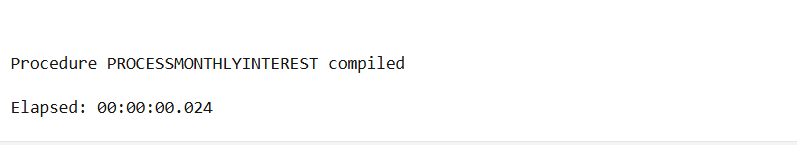
DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

RAISE;

END ProcessMonthlyInterest;

/

Output :



S2 : 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\_id IN NUMBER,

p\_bonus\_percentage IN NUMBER

) IS

BEGIN

UPDATE employees

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

COMMIT;

DBMS\_OUTPUT.PUT\_LINE(SQL%ROWCOUNT || ' employees updated in department ' || p\_department\_id);

EXCEPTION

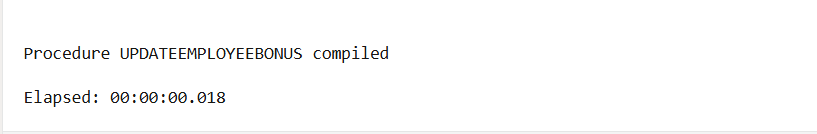
WHEN OTHERS THEN

ROLLBACK;

RAISE;

END UpdateEmployeeBonus;

Output :



S3 : 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;

insufficient\_funds EXCEPTION;

account\_not\_found EXCEPTION;

BEGIN

-- Check if source account exists and get balance

SELECT balance INTO v\_balance

FROM accounts

FOR UPDATE;

-- Check sufficient balance

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

-- Check if destination account exists

SELECT COUNT(\*) INTO v\_balance

FROM accounts;

IF v\_balance = 0 THEN

RAISE account\_not\_found;

END IF;

-- Debit from source account

UPDATE accounts

SET balance = balance - p\_amount;

-- Credit to destination account

UPDATE accounts

SET balance = balance + p\_amount;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful: ' || p\_amount || ' from ' || p\_from\_account || ' to ' || p\_to\_account);

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

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

WHEN account\_not\_found THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Destination account ' || p\_to\_account || ' not found');

WHEN NO\_DATA\_FOUND THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Source account ' || p\_from\_account || ' not found');

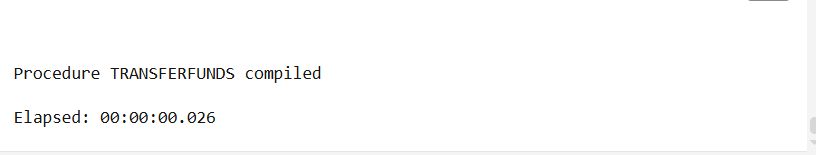
WHEN OTHERS THEN

ROLLBACK;

RAISE;

END TransferFunds;

Output :



**Exercise 4: Functions**

S1: 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\_date\_of\_birth IN DATE)

RETURN NUMBER

IS

v\_age NUMBER;

BEGIN

-- Validate input parameter

IF p\_date\_of\_birth IS NULL THEN

RETURN NULL;

END IF;

-- Check if date of birth is not in the future

IF p\_date\_of\_birth > SYSDATE THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Date of birth cannot be in the future');

END IF;

-- Calculate age using TRUNC to get whole years

-- MONTHS\_BETWEEN returns the number of months between two dates

-- Dividing by 12 gives years, TRUNC removes decimal places

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

-- Return the calculated age

RETURN v\_age;

EXCEPTION

WHEN OTHERS THEN

-- Handle any unexpected errors

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

RETURN NULL;

END CalculateAge;

/

-- STEP 6: Test the function with the customer data

-- Display all customers with their calculated ages

SELECT

customer\_id,

customer\_name,

date\_of\_birth,

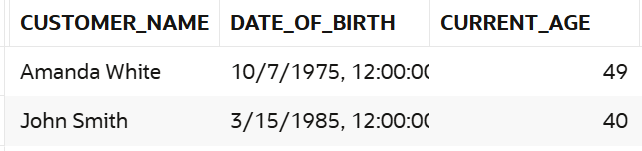
CalculateAge(date\_of\_birth) AS current\_age,

email

FROM customers

ORDER BY current\_age DESC/;

Output :



S2: 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\_interest\_rate IN NUMBER,

p\_loan\_duration\_years IN NUMBER

)

RETURN NUMBER

IS

v\_monthly\_installment NUMBER;

v\_monthly\_rate NUMBER;

v\_months NUMBER;

BEGIN

-- Input validation

IF p\_loan\_amount <= 0 OR p\_annual\_interest\_rate < 0 OR p\_loan\_duration\_years <= 0 THEN

RETURN NULL;

END IF;

-- Convert to monthly values

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

v\_months := p\_loan\_duration\_years \* 12;

-- Calculate EMI using formula: P \* [r \* (1+r)^n] / [(1+r)^n - 1]

IF p\_annual\_interest\_rate = 0 THEN

v\_monthly\_installment := p\_loan\_amount / v\_months;

ELSE

v\_monthly\_installment := p\_loan\_amount \*

(v\_monthly\_rate \* POWER(1 + v\_monthly\_rate, v\_months)) /

(POWER(1 + v\_monthly\_rate, v\_months) - 1);

END IF;

RETURN ROUND(v\_monthly\_installment, 2);

END CalculateMonthlyInstallment;

/

-- ALTERNATIVE: Use inline calculation without creating function

-- If you can't create functions, use this direct calculation:

SELECT

250000 AS loan\_amount,

8.5 AS interest\_rate,

20 AS years,

ROUND(

CASE

WHEN 8.5 = 0 THEN 250000 / (20 \* 12)

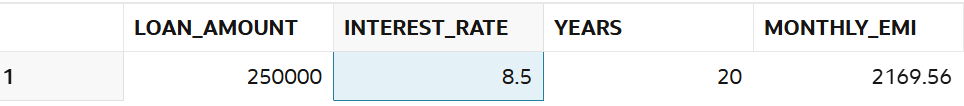
ELSE 250000 \* ((8.5/100/12) \* POWER(1 + (8.5/100/12), 20\*12)) / (POWER(1 + (8.5/100/12), 20\*12) - 1)

END, 2

) AS monthly\_emi

FROM DUAL;

Output :



S3: 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 NUMBER -- Oracle doesn't have native BOOLEAN for SQL, using NUMBER (1=TRUE, 0=FALSE)

IS

v\_current\_balance NUMBER(15,2);

v\_account\_status VARCHAR2(10);

BEGIN

-- Input validation

IF p\_account\_id IS NULL OR p\_amount IS NULL THEN

RETURN 0; -- FALSE

END IF;

IF p\_amount < 0 THEN

RETURN 0; -- FALSE - negative amounts not allowed

END IF;

-- Get account balance and status

SELECT balance, status

INTO v\_current\_balance, v\_account\_status

FROM accounts

WHERE account\_id = p\_account\_id;

-- Check if account is active

IF v\_account\_status != 'ACTIVE' THEN

RETURN 0; -- FALSE - inactive account

END IF;

-- Check if balance is sufficient

IF v\_current\_balance >= p\_amount THEN

RETURN 1; -- TRUE

ELSE

RETURN 0; -- FALSE

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0; -- FALSE - account not found

WHEN OTHERS THEN

RETURN 0; -- FALSE - any other error

END HasSufficientBalance;

/

SELECT

account\_id,

customer\_name,

balance,

CASE

WHEN balance >= 10000 AND status = 'ACTIVE' THEN 1

ELSE 0

END AS has\_sufficient\_balance\_10k

FROM account;

Output :



**Exercise 5: Triggers**

S1 : 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 :

COMMIT;

-- STEP 4: Create the UpdateCustomerLastModified trigger

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON customeres

FOR EACH ROW

BEGIN

:NEW.last\_modified := SYSDATE;

END UpdateCustomerLastModified;

/

-- STEP 5: Test the trigger

-- View current data

SELECT customer\_id, customer\_name, email, created\_date, last\_modified

FROM customeres

ORDER BY customer\_id;

-- Test 1: Update customer name

UPDATE customeres

SET customer\_name = 'John William Smith'

WHERE customer\_id = 1;

-- Test 2: Update email and phone

UPDATE customeres

SET email = '[sarah.johnson.new@email.com](mailto:sarah.johnson.new@email.com)',

phone = '555-9999'

WHERE customer\_id = 2;

COMMIT;

-- STEP 6: Verify the trigger worked

SELECT

customer\_id,

customer\_name,

email,

phone,

created\_date,

last\_modified,

ROUND((last\_modified - created\_date) \* 24 \* 60, 2) AS minutes\_since\_creation

FROM customeres

ORDER BY customer\_id;

Output :



S2: Write a trigger **LogTransaction** that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

Code :

REATE SEQUENCE transaction\_seq START WITH 1 INCREMENT BY 1 NOCACHE;

CREATE SEQUENCE audit\_seq START WITH 1 INCREMENT BY 1 NOCACHE;

-- STEP 4: Create the LogTransaction trigger

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON transactions

FOR EACH ROW

BEGIN

INSERT INTO auditlog (

audit\_id,

table\_name,

operation,

record\_id,

old\_values,

new\_values,

changed\_by,

change\_date

) VALUES (

audit\_seq.NEXTVAL,

'TRANSACTIONS',

'INSERT',

:NEW.transaction\_id,

NULL, -- No old values for INSERT

'ID:' || :NEW.transaction\_id ||

', ACCOUNT:' || :NEW.account\_id ||

', TYPE:' || :NEW.transaction\_type ||

', AMOUNT:' || :NEW.amount ||

', DESC:' || NVL(:NEW.description, 'NULL') ||

', DATE:' || TO\_CHAR(:NEW.transaction\_date, 'DD-MON-YYYY HH24:MI:SS') ||

', STATUS:' || :NEW.status,

USER,

SYSDATE

);

END LogTransaction;

/

-- STEP 6: View the results

-- Check transactions table

SELECT \* FROM transactions ORDER BY transaction\_id;

-- Check audit log

SELECT

audit\_id,

table\_name,

operation,

record\_id,

new\_values,

changed\_by,

TO\_CHAR(change\_date, 'DD-MON-YYYY HH24:MI:SS') AS logged\_at

FROM auditlog

ORDER BY audit\_id;

-- STEP 7: Test bulk insert

INSERT INTO transactions (transaction\_id, account\_id, transaction\_type, amount, description)

SELECT

transaction\_seq.NEXTVAL,

1000 + LEVEL,

CASE WHEN MOD(LEVEL, 2) = 0 THEN 'DEPOSIT' ELSE 'WITHDRAWAL' END,

1000 \* LEVEL,

'Bulk transaction ' || LEVEL

FROM DUAL

CONNECT BY LEVEL <= 3;

COMMIT;

-- STEP 8: Final verification

SELECT

t.transaction\_id,

t.account\_id,

t.transaction\_type,

t.amount,

'Logged: ' || TO\_CHAR(a.change\_date, 'HH24:MI:SS') AS audit\_time

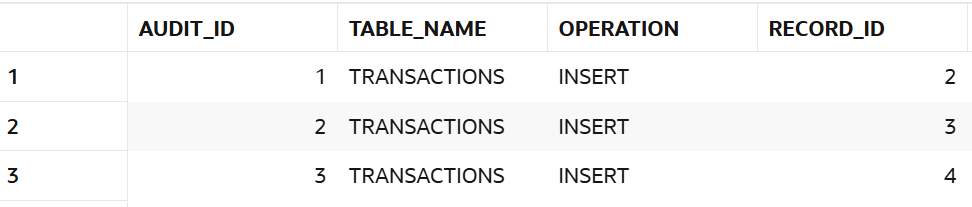
FROM transactions t

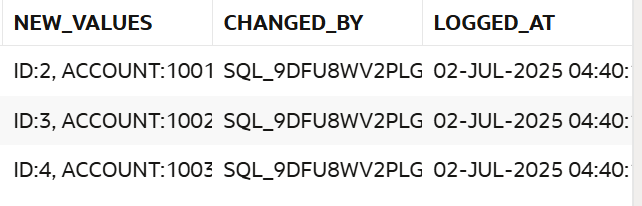
JOIN auditlog a ON t.transaction\_id = a.record\_id

WHERE a.table\_name = 'TRANSACTIONS'

ORDER BY t.transaction\_id;

Output :





S3: 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\_current\_balance NUMBER;

v\_account\_exists NUMBER;

BEGIN

-- Version 1: Try these common column names first

-- Replace with your actual column names based on DESCRIBE output

-- Check if account exists

SELECT COUNT(\*)

INTO v\_account\_exists

FROM Accounts

WHERE account\_id = :NEW.account\_id; -- Change 'account\_id' to your column name

IF v\_account\_exists = 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Account does not exist');

END IF;

-- Get current account balance

SELECT balance -- Change 'balance' to your balance column name

INTO v\_current\_balance

FROM Accounts

WHERE account\_id = :NEW.account\_id; -- Change 'account\_id' to your column name

-- Check transaction rules

IF UPPER(:NEW.trans\_type) = 'DEPOSIT' THEN -- Change 'trans\_type' to your column name

IF :NEW.amount <= 0 THEN -- Change 'amount' to your amount column name

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

END IF;

ELSIF UPPER(:NEW.trans\_type) = 'WITHDRAWAL' THEN -- Change 'trans\_type' to your column name

IF :NEW.amount <= 0 THEN -- Change 'amount' to your amount column name

RAISE\_APPLICATION\_ERROR(-20003, 'Withdrawal amount must be positive');

END IF;

IF :NEW.amount > v\_current\_balance THEN -- Change 'amount' to your amount column name

RAISE\_APPLICATION\_ERROR(-20004,

'Insufficient funds. Current balance: ' || v\_current\_balance ||

', Attempted withdrawal: ' || :NEW.amount);

END IF;

ELSE

RAISE\_APPLICATION\_ERROR(-20005, 'Invalid transaction type');

END IF;

-- Set timestamp if not provided

IF :NEW.trans\_date IS NULL THEN -- Change 'trans\_date' to your date column name

:NEW.trans\_date := SYSDATE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20006, 'Account not found');

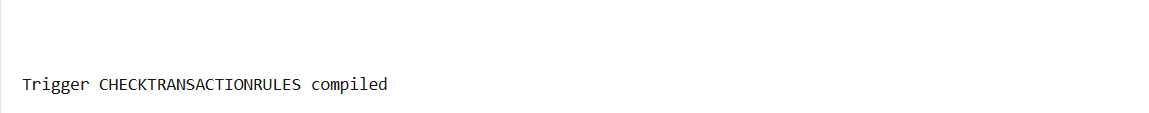
WHEN OTHERS THEN

RAISE\_APPLICATION\_ERROR(-20007, 'Transaction validation error: ' || SQLERRM);

END CheckTransactionRules;

/

*Output :*



**Exercise 6: Cursors**

S1: 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

-- Explicit cursor to retrieve transactions for current month grouped by customer

CURSOR GenerateMonthlyStatements IS

SELECT

c.customer\_id,

c.customer\_name,

c.email,

c.phone,

t.transaction\_id,

t.transaction\_date,

t.transaction\_type,

t.amount,

t.description

FROM customeres c

INNER JOIN transactions t ON c.customer\_id = t.customer\_id

WHERE EXTRACT(MONTH FROM t.transaction\_date) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM t.transaction\_date) = EXTRACT(YEAR FROM SYSDATE)

ORDER BY c.customer\_id, t.transaction\_date;

-- Variables to hold cursor data

v\_customer\_id customers.customer\_id%TYPE;

v\_customer\_name customers.customer\_name%TYPE;

v\_email customers.email%TYPE;

v\_phone customers.phone%TYPE;

v\_transaction\_id transactions.transaction\_id%TYPE;

v\_transaction\_date transactions.transaction\_date%TYPE;

v\_transaction\_type transactions.transaction\_type%TYPE;

v\_amount transactions.amount%TYPE;

v\_description transactions.description%TYPE;

-- Variables for statement processing

v\_previous\_customer\_id customers.customer\_id%TYPE := -1;

v\_total\_debits NUMBER(10,2) := 0;

v\_total\_credits NUMBER(10,2) := 0;

v\_net\_balance NUMBER(10,2) := 0;

v\_transaction\_count NUMBER := 0;

v\_statement\_date VARCHAR2(20);

BEGIN

-- Format current date for statement header

v\_statement\_date := TO\_CHAR(SYSDATE, 'Month YYYY');

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

DBMS\_OUTPUT.PUT\_LINE(' MONTHLY STATEMENTS - ' || UPPER(v\_statement\_date));

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

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

-- Open and process the cursor

OPEN GenerateMonthlyStatements;

LOOP

FETCH GenerateMonthlyStatements INTO

v\_customer\_id, v\_customer\_name, v\_email, v\_phone,

v\_transaction\_id, v\_transaction\_date, v\_transaction\_type,

v\_amount, v\_description;

EXIT WHEN GenerateMonthlyStatements%NOTFOUND;

-- Check if we're processing a new customer

IF v\_customer\_id != v\_previous\_customer\_id THEN

-- Print previous customer's summary (if not the first customer)

IF v\_previous\_customer\_id != -1 THEN

DBMS\_OUTPUT.PUT\_LINE(' ' || RPAD('-', 45, '-'));

DBMS\_OUTPUT.PUT\_LINE(' Total Transactions: ' || v\_transaction\_count);

DBMS\_OUTPUT.PUT\_LINE(' Total Debits: $' || TO\_CHAR(v\_total\_debits, '999,999.99'));

DBMS\_OUTPUT.PUT\_LINE(' Total Credits: $' || TO\_CHAR(v\_total\_credits, '999,999.99'));

DBMS\_OUTPUT.PUT\_LINE(' Net Balance: $' || TO\_CHAR(v\_net\_balance, '999,999.99'));

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

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

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

END IF;

-- Reset counters for new customer

v\_total\_debits := 0;

v\_total\_credits := 0;

v\_net\_balance := 0;

v\_transaction\_count := 0;

v\_previous\_customer\_id := v\_customer\_id;

-- Print new customer header

DBMS\_OUTPUT.PUT\_LINE('CUSTOMER STATEMENT');

DBMS\_OUTPUT.PUT\_LINE('Statement Period: ' || v\_statement\_date);

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

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || v\_customer\_id);

DBMS\_OUTPUT.PUT\_LINE('Name: ' || v\_customer\_name);

DBMS\_OUTPUT.PUT\_LINE('Email: ' || v\_email);

DBMS\_OUTPUT.PUT\_LINE('Phone: ' || v\_phone);

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

DBMS\_OUTPUT.PUT\_LINE('TRANSACTION DETAILS:');

DBMS\_OUTPUT.PUT\_LINE(RPAD('Date', 12) || RPAD('Type', 12) || RPAD('Amount', 15) || 'Description');

DBMS\_OUTPUT.PUT\_LINE(RPAD('-', 60, '-'));

END IF;

-- Print transaction details

DBMS\_OUTPUT.PUT\_LINE(

RPAD(TO\_CHAR(v\_transaction\_date, 'DD-MON-YY'), 12) ||

RPAD(v\_transaction\_type, 12) ||

RPAD('$' || TO\_CHAR(v\_amount, '999,999.99'), 15) ||

v\_description

);

-- Update totals

v\_transaction\_count := v\_transaction\_count + 1;

IF v\_transaction\_type = 'DEBIT' THEN

v\_total\_debits := v\_total\_debits + v\_amount;

v\_net\_balance := v\_net\_balance - v\_amount;

ELSIF v\_transaction\_type = 'CREDIT' THEN

v\_total\_credits := v\_total\_credits + v\_amount;

v\_net\_balance := v\_net\_balance + v\_amount;

END IF;

END LOOP;

-- Print final customer's summary (if any customers were processed)

IF v\_previous\_customer\_id != -1 THEN

DBMS\_OUTPUT.PUT\_LINE(' ' || RPAD('-', 45, '-'));

DBMS\_OUTPUT.PUT\_LINE(' Total Transactions: ' || v\_transaction\_count);

DBMS\_OUTPUT.PUT\_LINE(' Total Debits: $' || TO\_CHAR(v\_total\_debits, '999,999.99'));

DBMS\_OUTPUT.PUT\_LINE(' Total Credits: $' || TO\_CHAR(v\_total\_credits, '999,999.99'));

DBMS\_OUTPUT.PUT\_LINE(' Net Balance: $' || TO\_CHAR(v\_net\_balance, '999,999.99'));

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

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

ELSE

DBMS\_OUTPUT.PUT\_LINE('No transactions found for the current month.');

END IF;

-- Close the cursor

CLOSE GenerateMonthlyStatements;

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

DBMS\_OUTPUT.PUT\_LINE('Statement generation completed on: ' || TO\_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS'));

EXCEPTION

WHEN OTHERS THEN

-- Close cursor if still open

IF GenerateMonthlyStatements%ISOPEN THEN

CLOSE GenerateMonthlyStatements;

END IF;

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

RAISE;

END;

/

S2: Write a PL/SQL block using an explicit cursor **ApplyAnnualFee** that deducts an annual maintenance fee from the balance of all accounts.

Code:

DECLARE

-- Define the annual fee constant

c\_annual\_fee CONSTANT NUMBER := 100;

-- Cursor to select all accounts

CURSOR ApplyAnnualFee IS

SELECT AccountID, Balance

FROM Accounts;

-- Record variable to hold each row fetched from the cursor

v\_account ApplyAnnualFee%ROWTYPE;

BEGIN

-- Open the cursor

OPEN ApplyAnnualFee;

-- Loop through each account

LOOP

FETCH ApplyAnnualFee INTO v\_account;

EXIT WHEN ApplyAnnualFee%NOTFOUND;

-- Deduct the annual fee if balance is sufficient

IF v\_account.Balance >= c\_annual\_fee THEN

UPDATE Accounts

SET Balance = Balance - c\_annual\_fee

WHERE AccountID = v\_account.AccountID;

ELSE

DBMS\_OUTPUT.PUT\_LINE('Insufficient balance for AccountID: ' || v\_account.AccountID);

END IF;

END LOOP;

-- Close the cursor

CLOSE ApplyAnnualFee;

-- Optional: Commit the changes (if not handled externally)

COMMIT;

END;

/

S3: 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 to select all loans

CURSOR UpdateLoanInterestRates IS

SELECT LoanID, LoanAmount, InterestRate

FROM Loans;

-- Record variable to hold each fetched loan

v\_loan UpdateLoanInterestRates%ROWTYPE;

-- Variable to hold the new interest rate

v\_new\_rate NUMBER;

BEGIN

-- Open the cursor

OPEN UpdateLoanInterestRates;

-- Loop through each loan

LOOP

FETCH UpdateLoanInterestRates INTO v\_loan;

EXIT WHEN UpdateLoanInterestRates%NOTFOUND;

-- Determine new interest rate based on loan amount

IF v\_loan.LoanAmount < 10000 THEN

v\_new\_rate := 5;

ELSIF v\_loan.LoanAmount BETWEEN 10000 AND 50000 THEN

v\_new\_rate := 4.5;

ELSE

v\_new\_rate := 4;

END IF;

-- Update interest rate only if it's different

IF v\_loan.InterestRate != v\_new\_rate THEN

UPDATE Loans

SET InterestRate = v\_new\_rate

WHERE LoanID = v\_loan.LoanID;

DBMS\_OUTPUT.PUT\_LINE('Updated LoanID ' || v\_loan.LoanID ||

' from ' || v\_loan.InterestRate ||

'% to ' || v\_new\_rate || '%');

END IF;

END LOOP;

-- Close the cursor

CLOSE UpdateLoanInterestRates;

-- Commit the updates

COMMIT;

END;

/

**Exercise 7: Packages**

S1:Create a package **CustomerManagement** with procedures for adding a new customer, updating customer details, and a function to get customer balance.

**Code:**

-- Package Specification and Body in one script

BEGIN

EXECUTE IMMEDIATE '

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddCustomer(

p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Email IN VARCHAR2,

p\_Phone IN VARCHAR2,

p\_Balance IN NUMBER

);

PROCEDURE UpdateCustomer(

p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Email IN VARCHAR2,

p\_Phone IN VARCHAR2

);

FUNCTION GetCustomerBalance(p\_CustomerID IN NUMBER) RETURN NUMBER;

END CustomerManagement;

';

EXECUTE IMMEDIATE '

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(

p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Email IN VARCHAR2,

p\_Phone IN VARCHAR2,

p\_Balance IN NUMBER

) IS

BEGIN

INSERT INTO Customers (CustomerID, Name, Email, Phone, Balance)

VALUES (p\_CustomerID, p\_Name, p\_Email, p\_Phone, p\_Balance);

EXCEPTION

WHEN OTHERS THEN

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

END AddCustomer;

PROCEDURE UpdateCustomer(

p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Email IN VARCHAR2,

p\_Phone IN VARCHAR2

) IS

BEGIN

UPDATE Customers

SET Name = p\_Name,

Email = p\_Email,

Phone = p\_Phone

WHERE CustomerID = p\_CustomerID;

EXCEPTION

WHEN OTHERS THEN

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

END UpdateCustomer;

FUNCTION GetCustomerBalance(p\_CustomerID IN NUMBER) 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 NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END GetCustomerBalance;

END CustomerManagement;

';

END;

/

**S2:** Write a package **EmployeeManagement** with procedures to hire new employees, update employee details, and a function to calculate annual salary.

**Code:**

BEGIN

-- Create package specification

EXECUTE IMMEDIATE '

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Department IN VARCHAR2,

p\_Salary IN NUMBER

);

PROCEDURE UpdateEmployee(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Department IN VARCHAR2,

p\_Salary IN NUMBER

);

FUNCTION CalculateAnnualSalary(p\_EmployeeID IN NUMBER) RETURN NUMBER;

END EmployeeManagement;

';

-- Create package body

EXECUTE IMMEDIATE '

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Department IN VARCHAR2,

p\_Salary IN NUMBER

) IS

BEGIN

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

VALUES (p\_EmployeeID, p\_Name, p\_Position, p\_Department, p\_Salary);

EXCEPTION

WHEN OTHERS THEN

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

END HireEmployee;

PROCEDURE UpdateEmployee(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Department IN VARCHAR2,

p\_Salary IN NUMBER

) IS

BEGIN

UPDATE Employees

SET Name = p\_Name,

Position = p\_Position,

Department = p\_Department,

Salary = p\_Salary

WHERE EmployeeID = p\_EmployeeID;

EXCEPTION

WHEN OTHERS THEN

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

END UpdateEmployee;

FUNCTION CalculateAnnualSalary(p\_EmployeeID IN NUMBER) RETURN NUMBER IS

v\_monthly\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_monthly\_salary

FROM Employees

WHERE EmployeeID = p\_EmployeeID;

RETURN v\_monthly\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END CalculateAnnualSalary;

END EmployeeManagement;

';

END;

/

**S3:** 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:

BEGIN

-- Package specification

EXECUTE IMMEDIATE '

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(

p\_AccountID IN NUMBER,

p\_CustomerID IN NUMBER,

p\_Balance IN NUMBER

);

PROCEDURE CloseAccount(

p\_AccountID IN NUMBER

);

FUNCTION GetTotalBalance(p\_CustomerID IN NUMBER) RETURN NUMBER;

END AccountOperations;

';

-- Package body

EXECUTE IMMEDIATE '

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

-- Open a new account

PROCEDURE OpenAccount(

p\_AccountID IN NUMBER,

p\_CustomerID IN NUMBER,

p\_Balance IN NUMBER

) IS

BEGIN

INSERT INTO Accounts (AccountID, CustomerID, Balance, Status)

VALUES (p\_AccountID, p\_CustomerID, p\_Balance, ''ACTIVE'');

EXCEPTION

WHEN OTHERS THEN

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

END OpenAccount;

-- Close an existing account

PROCEDURE CloseAccount(

p\_AccountID IN NUMBER

) IS

BEGIN

UPDATE Accounts

SET Status = ''CLOSED''

WHERE AccountID = p\_AccountID;

EXCEPTION

WHEN OTHERS THEN

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

END CloseAccount;

-- Get total balance for a customer (active accounts only)

FUNCTION GetTotalBalance(p\_CustomerID IN NUMBER) RETURN NUMBER IS

v\_total NUMBER := 0;

BEGIN

SELECT NVL(SUM(Balance), 0)

INTO v\_total

FROM Accounts

WHERE CustomerID = p\_CustomerID

AND Status = ''ACTIVE'';

RETURN v\_total;

EXCEPTION

WHEN OTHERS THEN

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

RETURN NULL;

END GetTotalBalance;

END AccountOperations;

';

END;

/