PL/SQL Exercise:-

**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 rec IN (

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

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

) LOOP

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

UPDATE Loans

SET InterestRate = rec.InterestRate - (rec.InterestRate \* 0.01)

WHERE LoanID = rec.LoanID;

END IF;

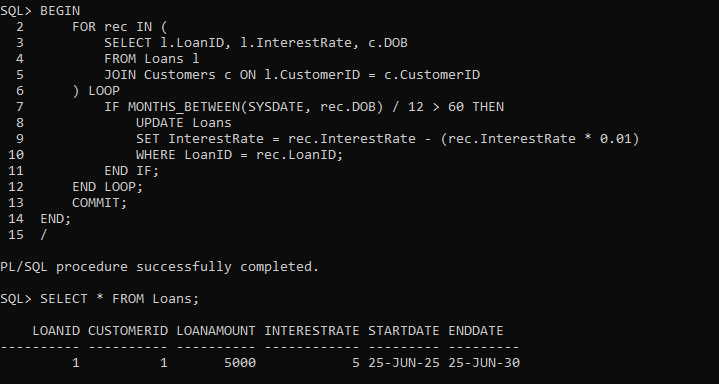
END LOOP;

COMMIT;

END;

/

Output:



Now there is no change in loans table as the only customer to have a loan is not above the age of 60

**Scenario 2:** A customer can be promoted to VIP status based on their balance.

**Question:** Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.

Code:

ALTER TABLE Customers ADD IsVIP CHAR(1);

BEGIN

FOR rec IN (SELECT CustomerID, Balance FROM Customers) LOOP

IF rec.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = rec.CustomerID;

ELSE

UPDATE Customers

SET IsVIP = 'N'

WHERE CustomerID = rec.CustomerID;

END IF;

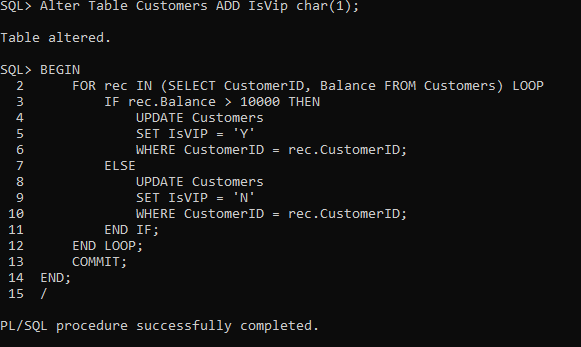
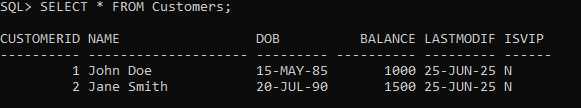
END LOOP;

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 rec IN (

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

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

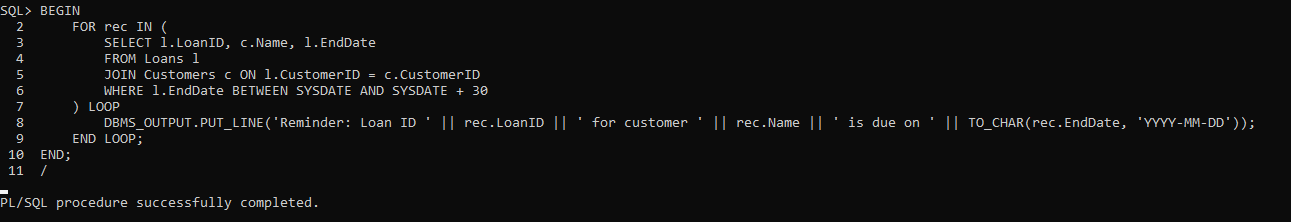
DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || rec.LoanID || ' for customer ' || rec.Name || ' is due on ' || TO\_CHAR(rec.EndDate, 'YYYY-MM-DD'));

END LOOP;

END;

/

Output:



There is not output because on loan is due in the given range.

**Exercise 2: Error Handling**

**Scenario 1:** Handle exceptions during fund transfers between accounts.

**Question:** Write a stored procedure **SafeTransferFunds** that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.

Code:

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER) IS

v\_from\_balance NUMBER;

BEGIN

SELECT balance INTO v\_from\_balance

FROM accounts

WHERE accountid = p\_from\_account\_id

FOR UPDATE;

IF v\_from\_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;

EXCEPTION

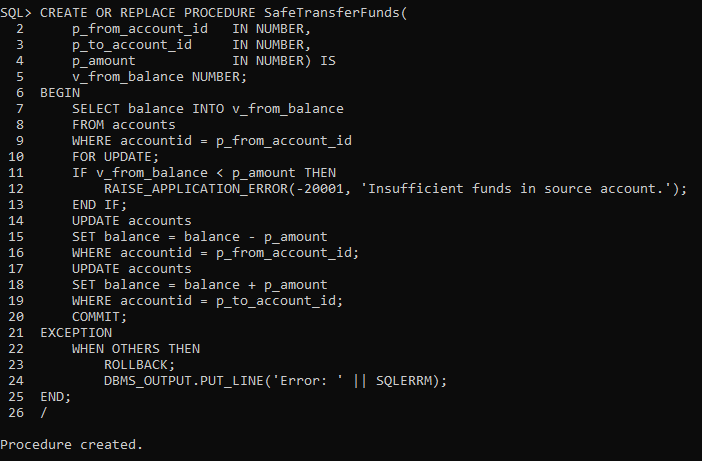
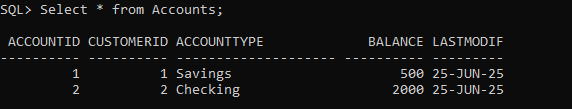
WHEN OTHERS THEN

ROLLBACK;

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

END;

EXEC SafeTransferFunds(1, 2, 500);

Output:  

**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

RAISE\_APPLICATION\_ERROR(-20002, 'Employee ID not found.');

END IF;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

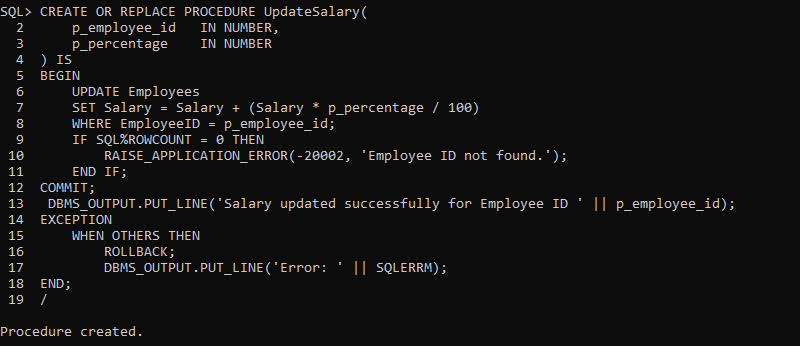
DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

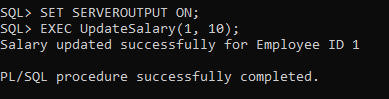
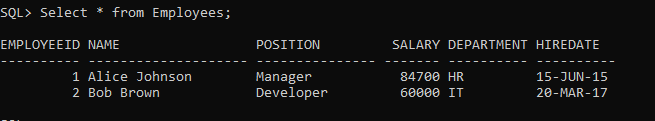
END;

/

SET SERVEROUTPUT ON;

EXEC UpdateSalary(1, 10);

Output: 

**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 inserted successfully: ' || p\_customer\_id);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

ROLLBACK;

WHEN OTHERS THEN

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

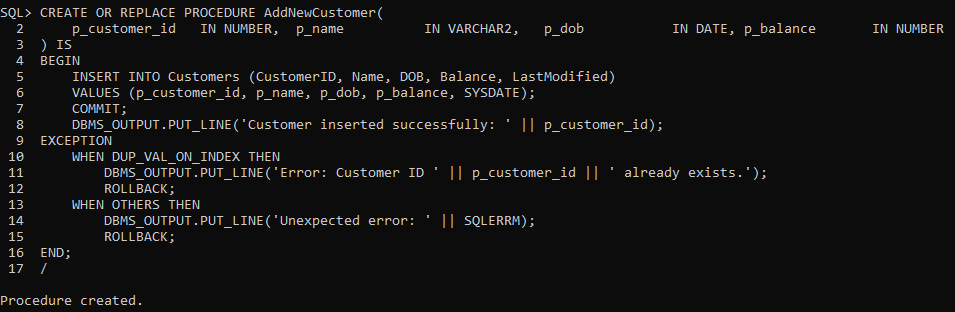
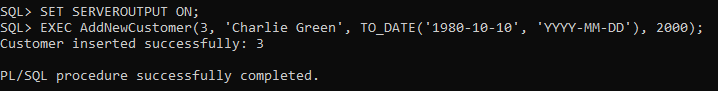
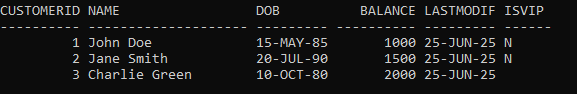
ROLLBACK;

END;

SET SERVEROUTPUT ON;

EXEC AddNewCustomer(3, 'Charlie Green', TO\_DATE('1980-10-10', 'YYYY-MM-DD'), 2000);

Output:

**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

FOR acc IN (

SELECT AccountID, Balance

FROM Accounts

WHERE AccountType = 'Savings'

FOR UPDATE

) LOOP

UPDATE Accounts

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

LastModified = SYSDATE

WHERE AccountID = acc.AccountID;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest processed for savings accounts.');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

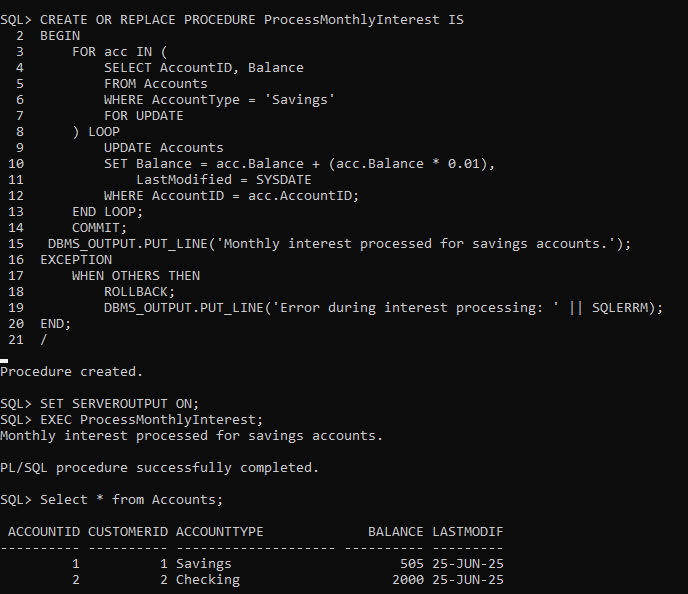
END;

/

SET SERVEROUTPUT ON;

EXEC ProcessMonthlyInterest;

Output:



**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

) IS

BEGIN

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 ' || SQL%ROWCOUNT || ' employee(s) in ' || p\_department || ' department.');

COMMIT;

ELSE

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

END IF;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

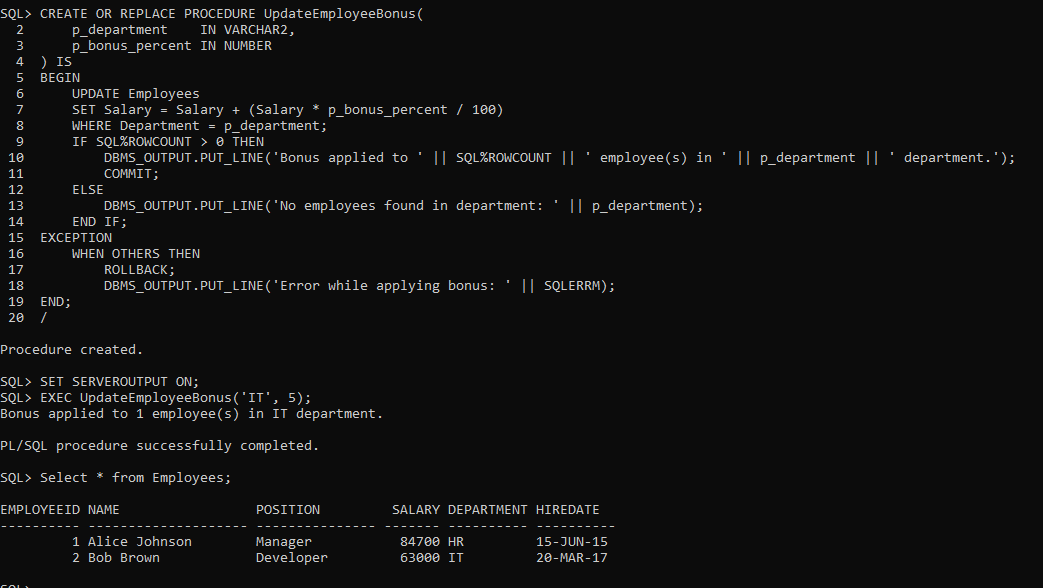
DBMS\_OUTPUT.PUT\_LINE('Error while applying bonus: ' || SQLERRM);

END;

/

SET SERVEROUTPUT ON;

EXEC UpdateEmployeeBonus('IT', 5);

Output: 

**Scenario 3:** Customers should be able to transfer funds between their accounts.

**Question:** Write a stored procedure **TransferFunds** that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.

Code:

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_from\_account\_id IN NUMBER, p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) IS

v\_from\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_from\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id

FOR UPDATE;

IF v\_from\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Insufficient funds in source account.');

END IF;

UPDATE Accounts

SET Balance = Balance - p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_from\_account\_id;

UPDATE Accounts

SET Balance = Balance + p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_to\_account\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Funds transferred successfully from Account ' ||

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

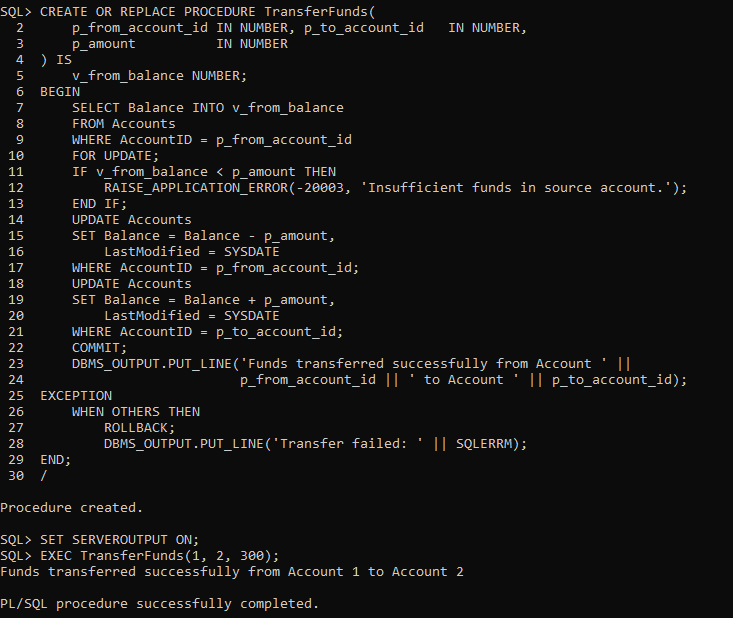
DBMS\_OUTPUT.PUT\_LINE('Transfer failed: ' || SQLERRM);

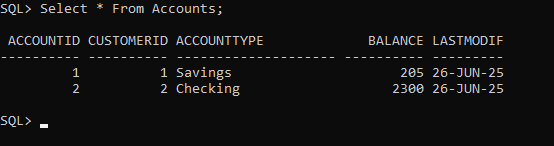
END;

/

SET SERVEROUTPUT ON;

EXEC TransferFunds(1, 2, 300);

Output: 



**Exercise 4: Functions**

**Scenario 1:** Calculate the age of customers for eligibility checks.

**Question:** Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

Code:

CREATE OR REPLACE FUNCTION CalculateAge(

p\_dob IN DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

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

RETURN v\_age;

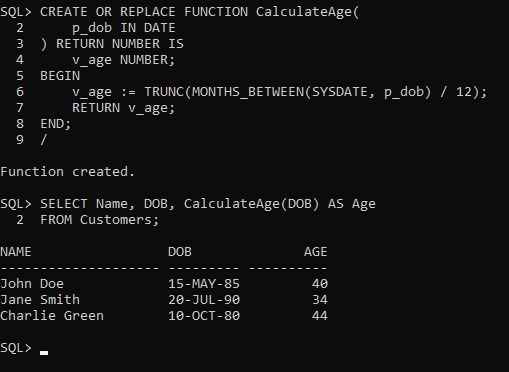
END;

/

SELECT Name, DOB, CalculateAge(DOB) AS Age

FROM Customers;

Output:



**Scenario 2:** The bank needs to compute the monthly installment for a loan.

**Question:** Write a function **CalculateMonthlyInstallment** that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

Code:

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

p\_loan\_amount IN NUMBER, p\_annual\_rate IN NUMBER, p\_loan\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_rate NUMBER; v\_total\_months NUMBER; v\_emi NUMBER;

BEGIN

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

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

IF v\_monthly\_rate = 0 THEN

v\_emi := p\_loan\_amount / v\_total\_months;

ELSE

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

END IF;

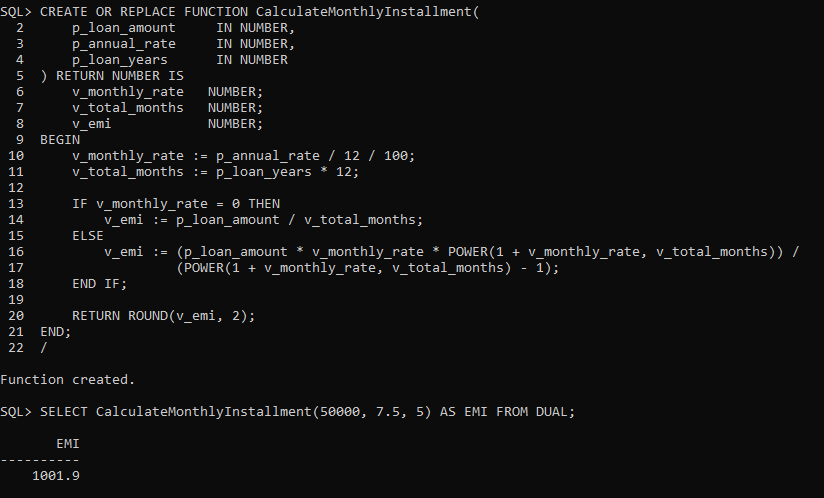
RETURN ROUND(v\_emi, 2);

END;

/

SELECT CalculateMonthlyInstallment(50000, 7.5, 5) AS EMI FROM DUAL;

Output:



**Scenario 3:** Check if a customer has sufficient balance before making a transaction.

**Question:** Write a function **HasSufficientBalance** that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.

Code:

CREATE OR REPLACE FUNCTION HasSufficientBalance(

p\_account\_id IN NUMBER,

p\_amount IN NUMBER

) RETURN VARCHAR2 IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_account\_id;

IF v\_balance >= p\_amount THEN

RETURN 'TRUE';

ELSE

RETURN 'FALSE';

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 'FALSE'; -- Account not found = insufficient

WHEN OTHERS THEN

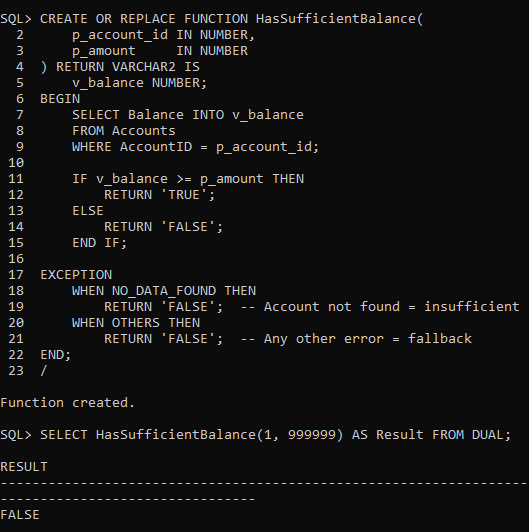
RETURN 'FALSE'; -- Any other error = fallback

END;

/

SELECT HasSufficientBalance(1, 100) AS Result FROM DUAL;

Output:



**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;

/

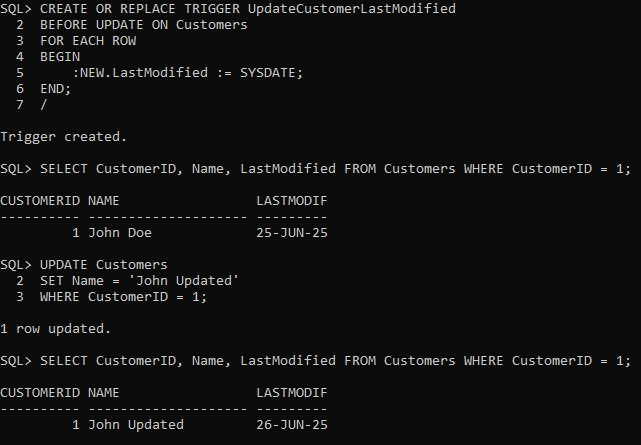
* SELECT CustomerID, Name, LastModified FROM Customers WHERE CustomerID = 1;
* UPDATE Customers

SET Name = 'John Updated'

WHERE CustomerID = 1;

* SELECT CustomerID, Name, LastModified FROM Customers WHERE CustomerID = 1;

Output:



**Scenario 2:** Maintain an audit log for all transactions.

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

Code:

* CREATE TABLE AuditLog (

LogID NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

AccountID NUMBER,

Action VARCHAR2(50),

LogDate DATE

);

* CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, AccountID, Action, LogDate)

VALUES (:NEW.TransactionID, :NEW.AccountID, 'Transaction Inserted', SYSDATE);

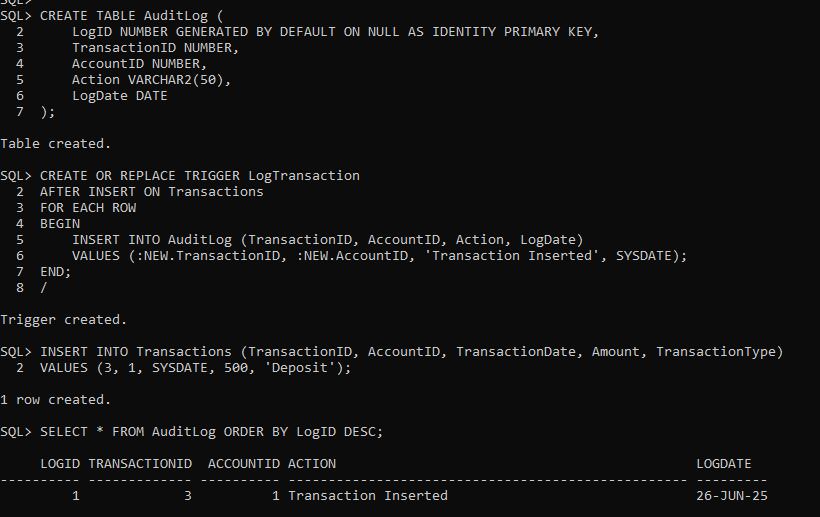
END;

/

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

VALUES (3, 1, SYSDATE, 500, 'Deposit');

* SELECT \* FROM AuditLog ORDER BY LogID DESC;

Output: 

**Scenario 3:** Enforce business rules on deposits and withdrawals.

**Question:** Write a trigger **CheckTransactionRules** that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.

Code:

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

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

RAISE\_APPLICATION\_ERROR(-20010, 'Withdrawal amount exceeds available balance.');

END IF;

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

RAISE\_APPLICATION\_ERROR(-20011, 'Deposit amount must be positive.');

END IF;

END;

/

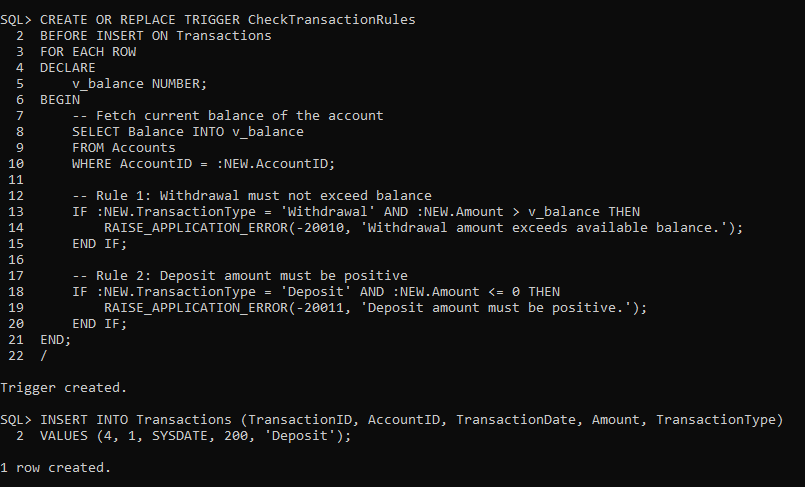
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (4, 1, SYSDATE, 200, 'Deposit');

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

VALUES (5, 1, SYSDATE, 999999, 'Withdrawal');

Output:



**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 monthly\_cursor IS

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

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\_name Customers.Name%TYPE;

v\_account\_id Accounts.AccountID%TYPE;

v\_date Transactions.TransactionDate%TYPE;

v\_amount Transactions.Amount%TYPE;

v\_type Transactions.TransactionType%TYPE;

BEGIN

OPEN monthly\_cursor;

LOOP

FETCH monthly\_cursor INTO v\_name, v\_account\_id, v\_date, v\_amount, v\_type;

EXIT WHEN monthly\_cursor%NOTFOUND;

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

', Account: ' || v\_account\_id ||

', Date: ' || TO\_CHAR(v\_date, 'DD-MON-YYYY') ||

', Amount: ' || v\_amount ||

', Type: ' || v\_type);

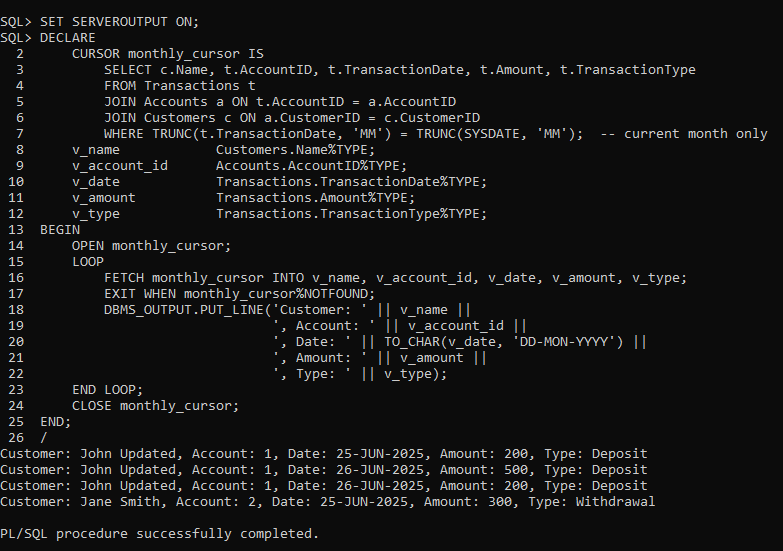
END LOOP;

CLOSE monthly\_cursor;

END;

/

Output:



**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:

SET SERVEROUTPUT ON;

DECLARE

CURSOR account\_cursor IS

SELECT AccountID, Balance

FROM Accounts

FOR UPDATE;

v\_account\_id Accounts.AccountID%TYPE;

v\_balance Accounts.Balance%TYPE;

v\_fee CONSTANT NUMBER := 100;

BEGIN

OPEN account\_cursor;

LOOP

FETCH account\_cursor INTO v\_account\_id, v\_balance;

EXIT WHEN account\_cursor%NOTFOUND;

IF v\_balance >= v\_fee THEN

UPDATE Accounts

SET Balance = Balance - v\_fee,

LastModified = SYSDATE

WHERE AccountID = v\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Fee deducted from Account ' || v\_account\_id ||

'. New Balance: ' || (v\_balance - v\_fee));

ELSE

DBMS\_OUTPUT.PUT\_LINE('Insufficient funds in Account ' || v\_account\_id ||

'. No fee deducted.');

END IF;

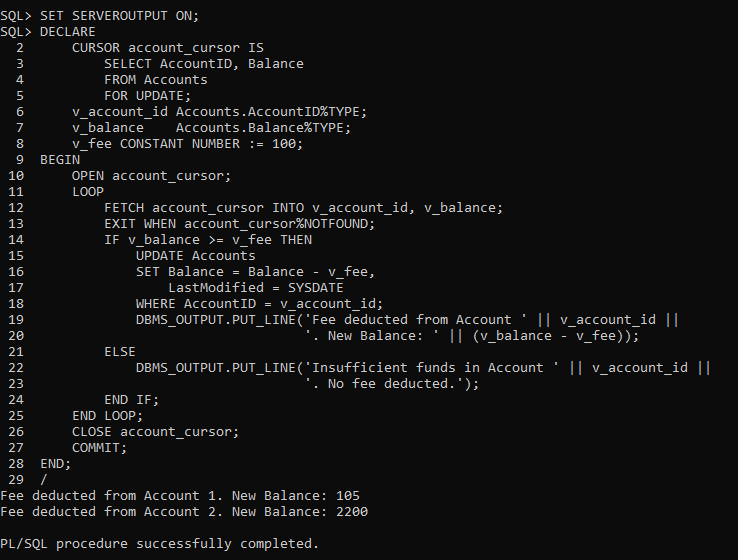
END LOOP;

CLOSE account\_cursor;

COMMIT;

END;

/

Output: 

**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:

SET SERVEROUTPUT ON;

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, LoanAmount

FROM Loans

FOR UPDATE;

v\_loan\_id Loans.LoanID%TYPE;

v\_amount Loans.LoanAmount%TYPE;

v\_new\_rate NUMBER;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_loan\_id, v\_amount;

EXIT WHEN loan\_cursor%NOTFOUND;

IF v\_amount > 10000 THEN

v\_new\_rate := 4.5;

ELSE

v\_new\_rate := 5.5;

END IF;

UPDATE Loans

SET InterestRate = v\_new\_rate

WHERE LoanID = v\_loan\_id;

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

' to new interest rate: ' || v\_new\_rate || '%');

END LOOP;

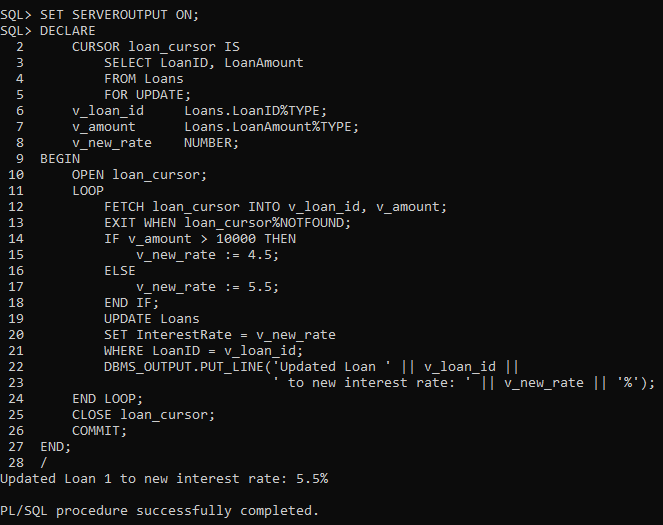
CLOSE loan\_cursor;

COMMIT;

END;

/

Output:



**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 IS

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 IS

PROCEDURE AddCustomer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) IS

BEGIN

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

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

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END AddCustomer;

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;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No customer found with ID: ' || p\_customer\_id);

ELSE

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

END IF;

END UpdateCustomerDetails;

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 NULL;

END GetCustomerBalance;

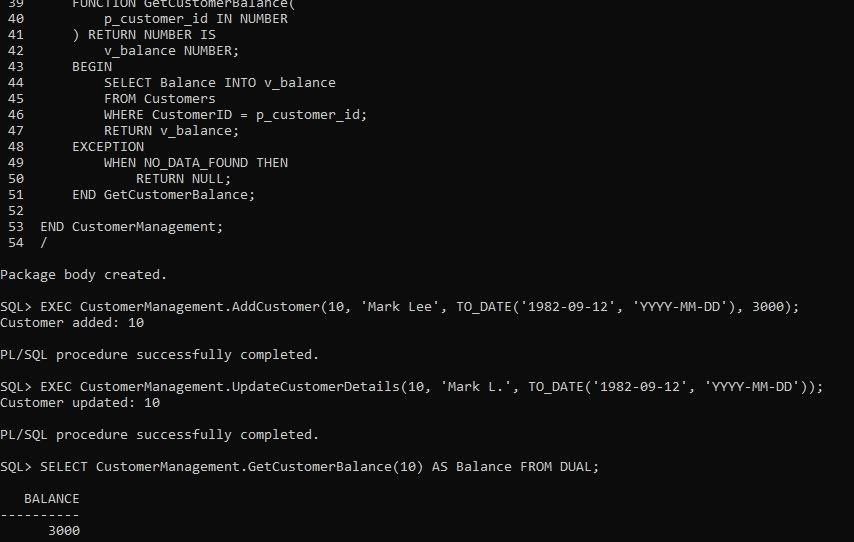
END CustomerManagement;

/

* EXEC CustomerManagement.AddCustomer(10, 'Mark Lee', TO\_DATE('1982-09-12', 'YYYY-MM-DD'), 3000);

EXEC CustomerManagement.UpdateCustomerDetails(10, 'Mark L.', TO\_DATE('1982-09-12', 'YYYY-MM-DD'));

SELECT CustomerManagement.GetCustomerBalance(10) AS Balance FROM DUAL;

Output: 

**Scenario 2:** Create a package to manage employee data.

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

Code:

* CREATE OR REPLACE PACKAGE EmployeeManagement IS

PROCEDURE HireEmployee(

p\_employee\_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 UpdateEmployeeDetails(

p\_employee\_id IN NUMBER,

p\_position IN VARCHAR2,

p\_salary IN NUMBER

);

FUNCTION CalculateAnnualSalary(

p\_employee\_id IN NUMBER

) RETURN NUMBER;

END EmployeeManagement;

/

* CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS PROCEDURE HireEmployee(

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

DBMS\_OUTPUT.PUT\_LINE('Employee hired: ' || p\_employee\_id);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END HireEmployee;

PROCEDURE UpdateEmployeeDetails(

p\_employee\_id IN NUMBER,

p\_position IN VARCHAR2,

p\_salary IN NUMBER

) IS

BEGIN

UPDATE Employees

SET Position = p\_position,

Salary = p\_salary

WHERE EmployeeID = p\_employee\_id;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No employee found with ID: ' || p\_employee\_id);

ELSE

DBMS\_OUTPUT.PUT\_LINE('Employee updated: ' || p\_employee\_id);

END IF;

END UpdateEmployeeDetails;

FUNCTION CalculateAnnualSalary(

p\_employee\_id IN NUMBER

) RETURN NUMBER IS

v\_monthly\_salary NUMBER;

BEGIN

SELECT Salary INTO v\_monthly\_salary

FROM Employees

WHERE EmployeeID = p\_employee\_id;

RETURN v\_monthly\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN NULL;

END CalculateAnnualSalary;

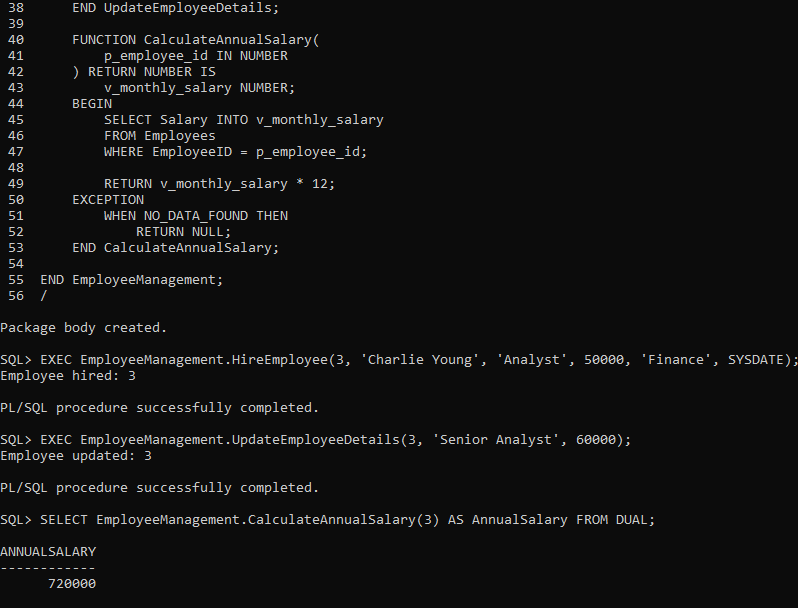
END EmployeeManagement;

/

* EXEC EmployeeManagement.HireEmployee(3, 'Charlie Young', 'Analyst', 50000, 'Finance', SYSDATE);

EXEC EmployeeManagement.UpdateEmployeeDetails(3, 'Senior Analyst', 60000);

SELECT EmployeeManagement.CalculateAnnualSalary(3) AS AnnualSalary FROM DUAL;

Output: 

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

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

Code:

* CREATE OR REPLACE PACKAGE AccountOperations IS

PROCEDURE OpenAccount(

p\_account\_id IN NUMBER,

p\_customer\_id IN NUMBER,

p\_account\_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 IS

PROCEDURE OpenAccount(

p\_account\_id IN NUMBER,

p\_customer\_id IN NUMBER,

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

DBMS\_OUTPUT.PUT\_LINE('Account opened: ' || p\_account\_id);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

END OpenAccount;

PROCEDURE CloseAccount(

p\_account\_id IN NUMBER

) IS

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_account\_id;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No account found with ID: ' || p\_account\_id);

ELSE

DBMS\_OUTPUT.PUT\_LINE('Account closed: ' || p\_account\_id);

END IF;

END CloseAccount;

FUNCTION GetTotalBalance(

p\_customer\_id IN NUMBER

) RETURN NUMBER IS

v\_total\_balance NUMBER;

BEGIN

SELECT NVL(SUM(Balance), 0)

INTO v\_total\_balance

FROM Accounts

WHERE CustomerID = p\_customer\_id;

RETURN v\_total\_balance;

END GetTotalBalance;

END AccountOperations;

/

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

EXEC AccountOperations.CloseAccount(5);

SELECT AccountOperations.GetTotalBalance(1) AS TotalBalance FROM DUAL;

Output: 