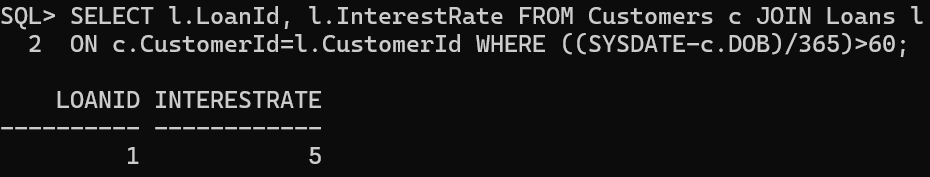
**Exercise 1: Control Structures**

**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.
  + **Before:**



* + **PL/SQL Code:**

BEGIN

FOR loan IN (SELECT l.LoanId, l.InterestRate FROM Customers c JOIN Loans l ON c.CustomerId=l.CustomerId WHERE ((SYSDATE-c.DOB)/365)>60) LOOP

UPDATE loans SET InterestRate=loan.InterestRate-1 WHERE LoanId=loan.LoanId;

END LOOP;

COMMIT;

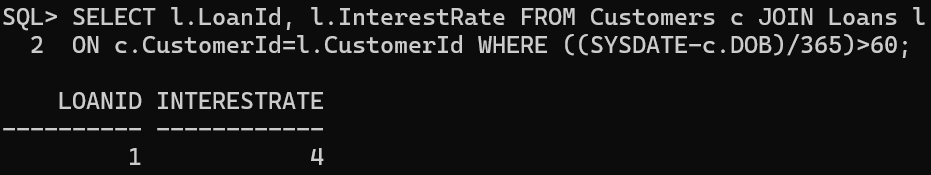
END;

/

* + **PL/SQL Output:**

****

* + **After:**



**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:**

DECLARE

isVIP BOOLEAN:=FALSE;

BEGIN

FOR c IN (SELECT CustomerId,Balance FROM customers) LOOP

isVIP:=FALSE;

IF c.Balance>10000 THEN

isVIP:=TRUE;

DBMS\_OUTPUT.PUT\_LINE('isVIP flag set to TRUE for Customer with CustomerId: '||c.CustomerId);

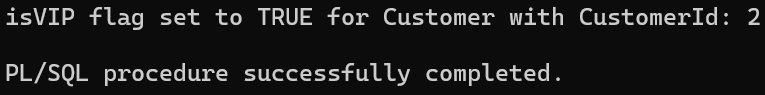
END IF;

END LOOP;

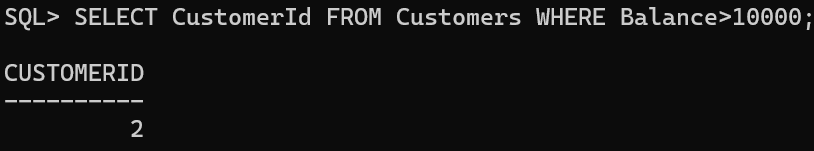
END;

/

* + **PL/SQL Code Output:**

****

* + **Checking in Customer Table:**

****

**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.
  + **PL/SQL Code:**

BEGIN

FOR loan IN (SELECT c.CustomerId,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

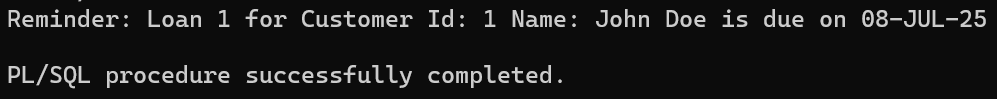
DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan '||loan.LoanId||' for Customer Id: '||loan.CustomerId||' Name: '||loan.Name||' is due on '||loan.EndDate);

END LOOP;

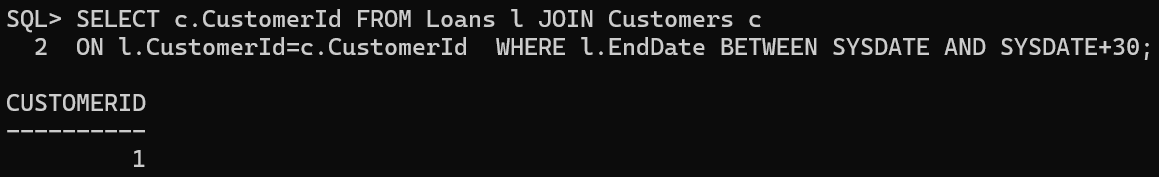
END;

/

* + **PL/SQL Code Output:**



* + **Checking Output:**

****

**Exercise 2: Error Handling**

**Schema:**

CREATE TABLE ErrorLog(

ErrorId NUMBER GENERATED ALWAYS AS IDENTITY,

ErrorMessage VARCHAR2(100),

LogTime DATE DEFAULT SYSDATE

);

**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(

FromAccountId IN NUMBER,

ToAccountId IN NUMBER,

Amount IN NUMBER

) AS

Balance NUMBER;

ErrorMessage VARCHAR2(100);

BEGIN

SELECT a.Balance INTO Balance FROM Accounts a WHERE a.AccountId=FromAccountId;

IF Balance<Amount THEN

RAISE\_APPLICATION\_ERROR(-20001,'Insufficient Funds');

END IF;

UPDATE Accounts a SET a.Balance=Balance-Amount,a.LastModified=SYSDATE WHERE a.AccountId=FromAccountId;

UPDATE Accounts a SET a.Balance=Balance+Amount,a.LastModified=SYSDATE WHERE a.AccountId=ToAccountId;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

ErrorMessage:='Transfer failed: '||SQLERRM;

INSERT INTO ErrorLog(ErrorMessage) VALUES(ErrorMessage);

END;

/

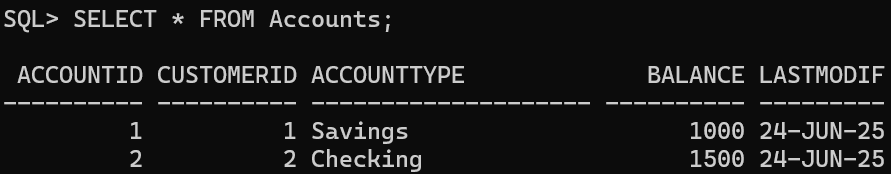
* + **Procedure Output:**

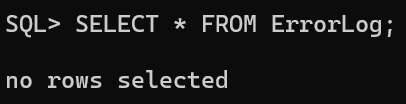
****

* + **Case-1:**

There are sufficient funds in source account.

* Before:





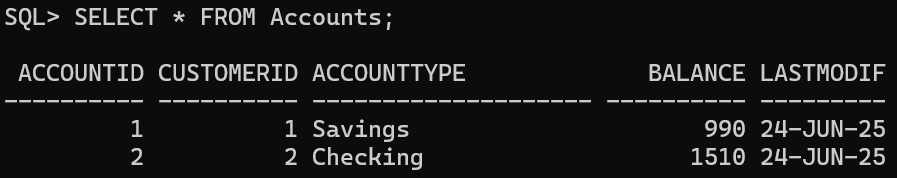
* Code:

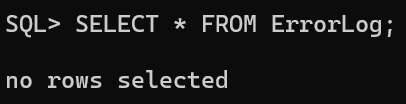
EXEC SafeTransferFunds(1,2,10);

* + - Code Output:



* After:

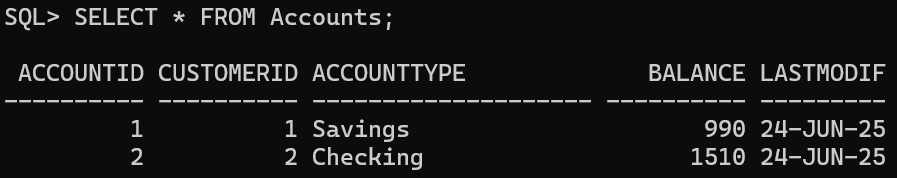


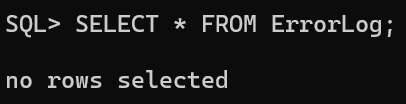


* + **Case-2:**

When there are insufficient funds in source account.

* Before:





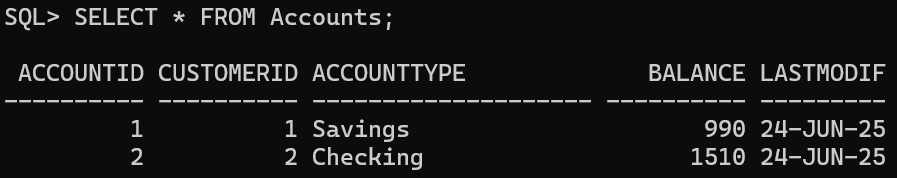
* Code:

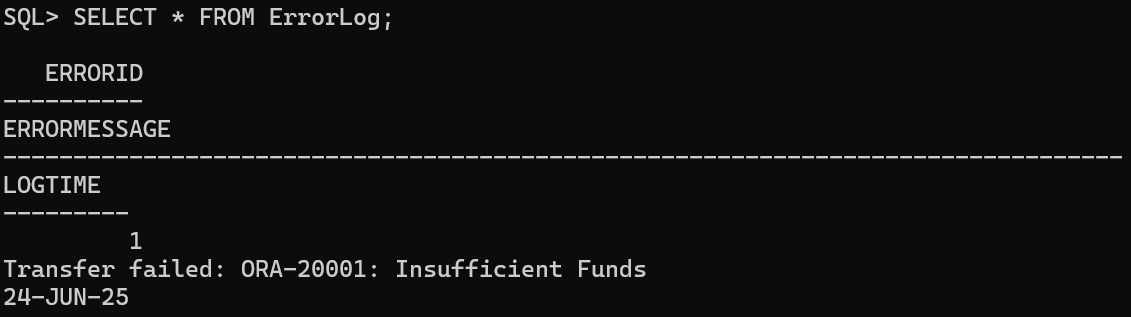
EXEC SafeTransferFunds(1,2,1000);

* + - Code Output:



* After:





**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(

PEmployeeId IN NUMBER,

Percentage IN NUMBER

) AS

ErrorMessage VARCHAR(100);

BEGIN

UPDATE Employees SET Salary=Salary+(Salary\*Percentage/100) WHERE EmployeeId=PEmployeeId;

IF SQL%ROWCOUNT=0 THEN

RAISE\_APPLICATION\_ERROR(-20001,'Employee ID doesn''t exist');

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ErrorMessage:='Salary Update Failed: ' || SQLERRM;

ROLLBACK;

INSERT INTO ErrorLog(ErrorMessage) VALUES(ErrorMessage);

COMMIT;

END;

/

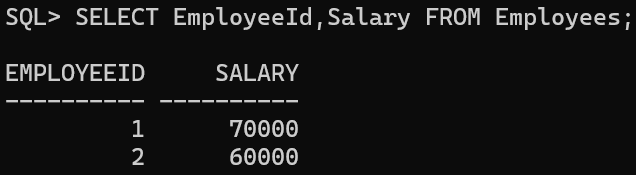
* + **Procedure Output:**

****

* + **Case-1:**

Employee Id which is used in Procedure exists in the table.

* Before:





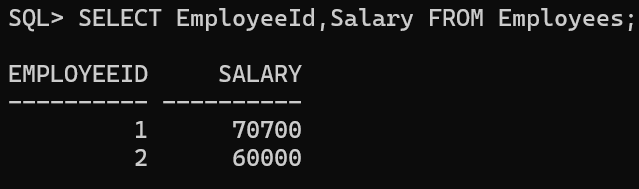
* Code:

EXEC UpdateSalary(1,1);

* Code Output:



* After:

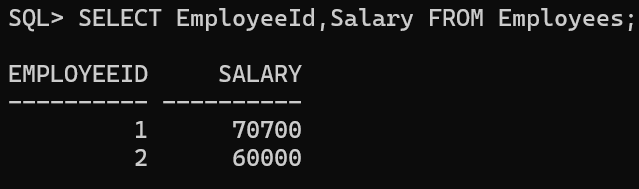




* + **Case-2:**

Employee Id which is used in Procedure doesn’t exist in the table.

* Before:





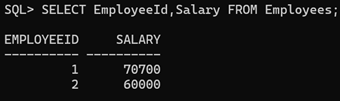
* Code:

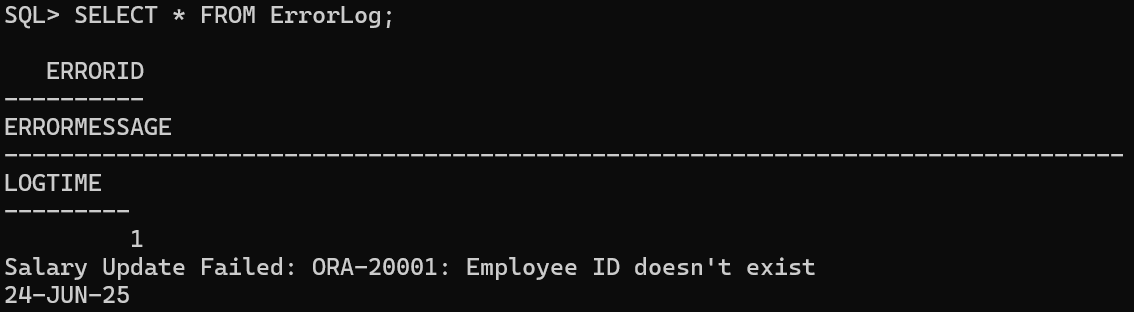
EXEC UpdateSalary(3,1);

* Code Output:



* After:





**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(

PCustomerID NUMBER,

PName VARCHAR2,

PDOB DATE,

PBalance NUMBER

) AS

ErrorMessage VARCHAR2(100);

BEGIN

INSERT INTO Customers VALUES(PCustomerId,PName,PDOB,PBalance,SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ErrorMessage:='Customer Insert Failed: Duplicate Id '||PCustomerId;

INSERT INTO ErrorLog(ErrorMessage) VALUES(ErrorMessage);

COMMIT;

END;

/

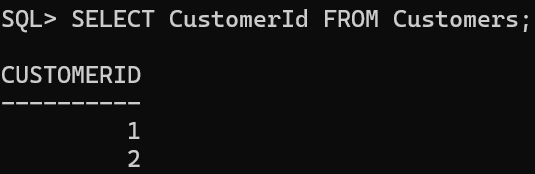
* + **Procedure Output:**

****

* + **Case-1:**

New Customer Id is not present in Customers Table.

* Before:





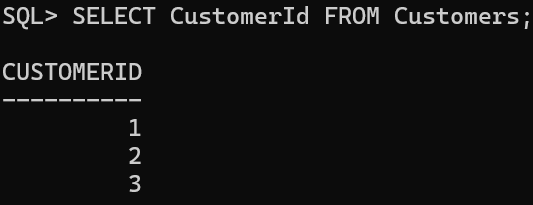
* Code:

EXEC AddNewCustomer(3, 'Alice',SYSDATE-18\*365-182,2000);

* Code Output:



* After:

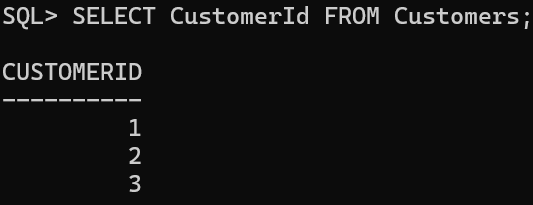


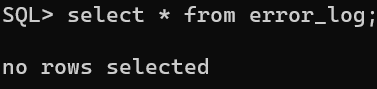


* + **Case-2:**

New Customer Id already exists in the Customers Table.

* Before:





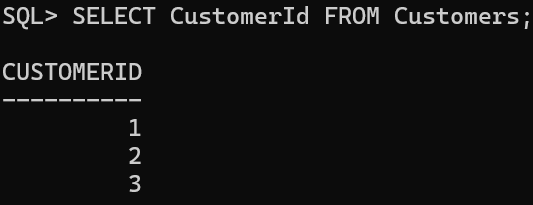
* Code:

EXEC AddNewCustomer(3, 'Alice',SYSDATE-18\*365-182,2000);

* Code Output:



* After:





**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 AS

BEGIN

UPDATE Accounts SET Balance=Balance+(Balance/100);

COMMIT;

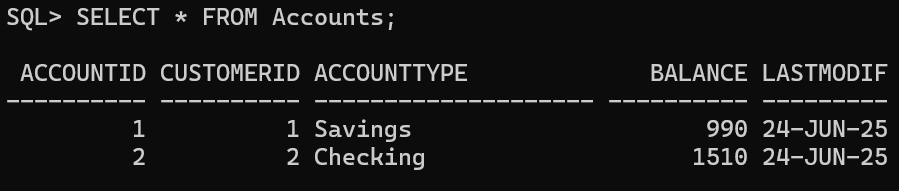
END;

/

* + **Procedure Output:**

****

* + **Before:**



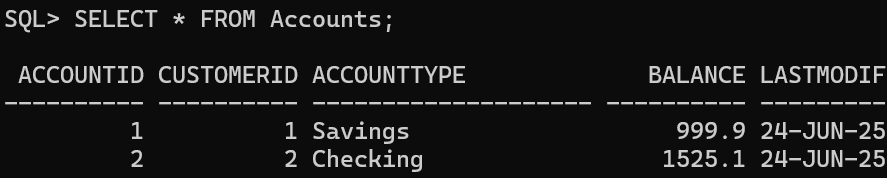
* + **Code:**

EXEC ProcessMonthlyInterest();

* + **Code Output:**



* + **After:**



**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(

pdept VARCHAR2,

percent NUMBER

) AS

BEGIN

UPDATE employees SET salary=salary+(salary\*percent/100) WHERE dept=pdept;

COMMIT;

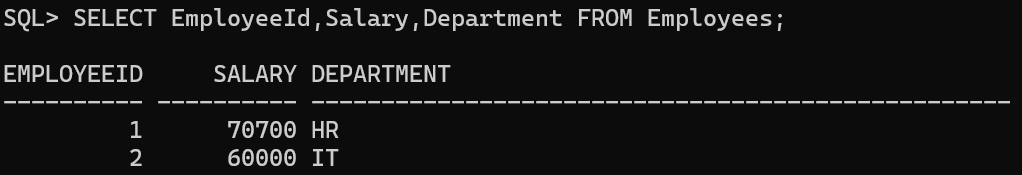
END;

/

* + **Procedure Output:**



* + **Before:**



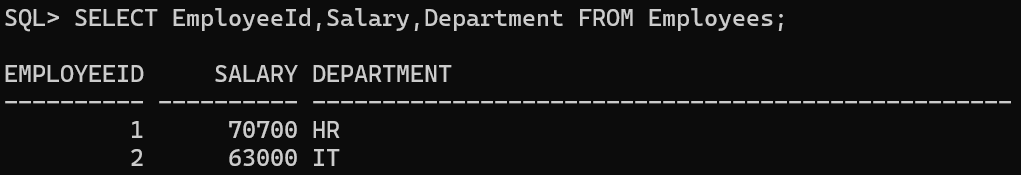
* + **Code:**

EXEC UpdateEmployeeBonus('IT', 5);

* + **Code Output:**



* + **After:**



**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(

FromAccountId IN NUMBER,

ToAccountId IN NUMBER,

Amount IN NUMBER

) AS

Balance NUMBER;

BEGIN

SELECT a.Balance INTO Balance FROM Accounts a WHERE a.AccountId=FromAccountId;

IF Balance<Amount THEN

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

RETURN;

END IF;

UPDATE accounts a SET a.Balance=Balance-Amount,LastModified=SYSDATE WHERE a.AccountId=FromAccountId;

UPDATE accounts a SET a.Balance=Balance+Amount,LastModified=SYSDATE WHERE a.AccountId=ToAccountId;

COMMIT;

END;

/

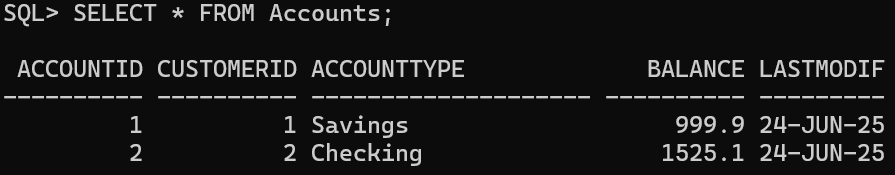
* + **Procedure Output:**



* + **Case-1:**

Sufficient Funds Available in Source Account

* + - Before:



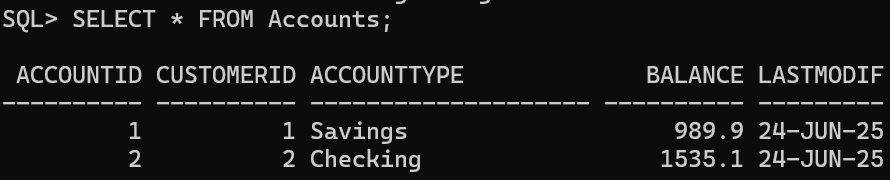
* + - Code:

EXEC TransferFunds(1,2,10);

* + - Code Output:



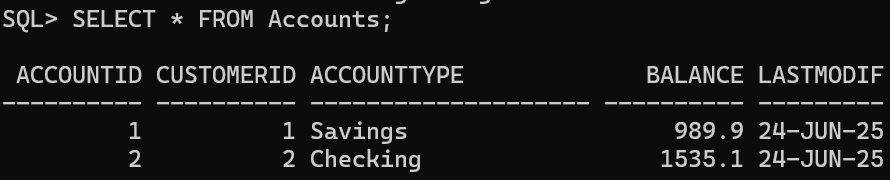
* + - After:



* + **Case-2:**

Sufficient Funds Available in Source Account

* + - Before:

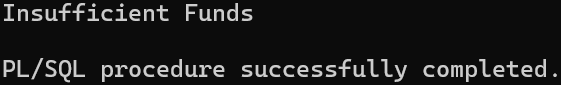


* + - Code:

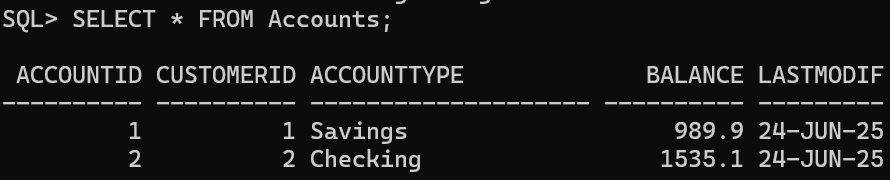
SET SERVEROUTPUT ON;

EXEC TransferFunds(1,2,1000);

* + - Code Output:



* + - After:



**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.
  + **Function Code:**

CREATE OR REPLACE FUNCTION CalculateAge(dob DATE)

RETURN NUMBER IS

age NUMBER:=0;

BEGIN

age:=TRUNC((SYSDATE-dob)/365);

RETURN age;

END;

/

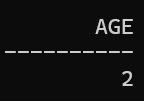
* + **Function Output:**

****

* + **Execution Code:**

SELECT CalculateAge(SYSDATE-800) FROM DUAL;

* + **Execution 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.
  + **Function Code:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

amount NUMBER,

interest NUMBER,

duration NUMBER

) RETURN NUMBER IS

emi NUMBER(10,2):=0;

p NUMBER(10,2):=0;

r NUMBER(10,5):=0;

n NUMBER(10):=0;

BEGIN

p:=amount;

r:=interest/12/100;

n:=duration\*12;

emi:=(p\*r\*((1+r)\*\*n))/(((1+r)\*\*n)-1);

RETURN emi;

END;

/

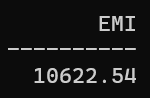
* + **Function Output:**

****

* + **Execution Code:**

SELECT CalculateMonthlyInstallment(500000,10,5) AS EMI FROM DUAL;

* + **Execution 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.
  + **Function Code:**

CREATE OR REPLACE FUNCTION HasInsufficientBalance(

FAccountId NUMBER,

FAmount NUMBER

) RETURN BOOLEAN IS

FBalance NUMBER(10,2);

BEGIN

SELECT Balance INTO FBalance FROM Accounts WHERE AccountId=FAccountId;

IF FAmount>FBalance THEN

RETURN TRUE;

END IF;

RETURN FALSE;

END;

/

* + **Function Output:**



* + **Case-1:**

Sufficient Balance in Specified Account.

* Execution Code:

BEGIN

IF HasInsufficientBalance(1,900) THEN

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

ELSE

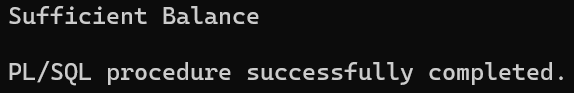
DBMS\_OUTPUT.PUT\_LINE('Sufficient Balance');

END IF;

END;

/

* Execution Output:



* + **Case-2:**

Insufficient Funds in Specified Account

* + - Execution Code:

BEGIN

IF HasInsufficientBalance(1,1000) THEN

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

ELSE

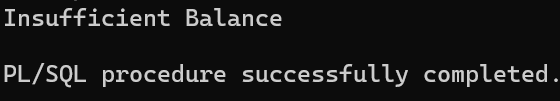
DBMS\_OUTPUT.PUT\_LINE('Sufficient Balance');

END IF;

END;

/

* + - Execution Output:



**Exercise 5: Triggers**

**Schema:**

CREATE TABLE AuditLog (

AuditID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

AccountID NUMBER,

TransactionType VARCHAR2(10),

Amount NUMBER,

ActionType VARCHAR2(20),

LoggedAt DATE DEFAULT SYSDATE,

Description VARCHAR2(100)

);

**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.
  + **Trigger Code:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE OF CustomerId,Name,DOB,Balance ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified:=SYSDATE;

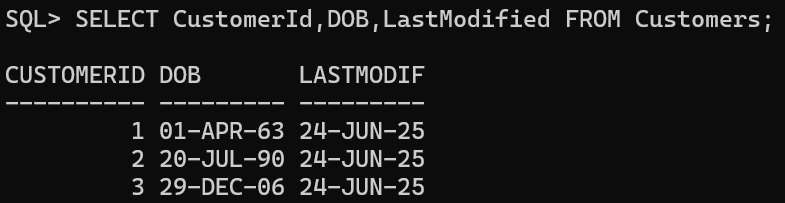
END;

/

* + **Trigger Output:**

****

* + **Before:**

****

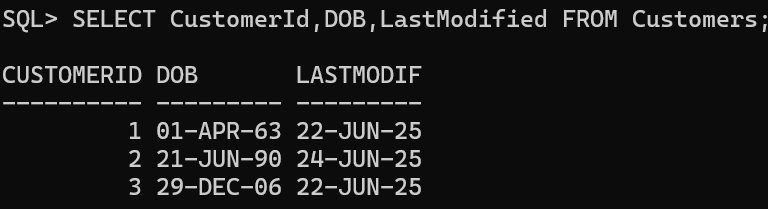
* + **Update Code:**

UPDATE customers SET DOB=TO\_DATE('1990-06-21','YYYY-MM-DD') WHERE CustomerId=2;

* + **Update Code Output:**

****

* + **After:**

****

**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.
  + **Trigger Code:**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, AccountID, TransactionType, Amount, ActionType, Description) VALUES (:NEW.TransactionID, :NEW.AccountID, :NEW.TransactionType, :NEW.Amount, 'INSERT', 'Transaction logged successfully');

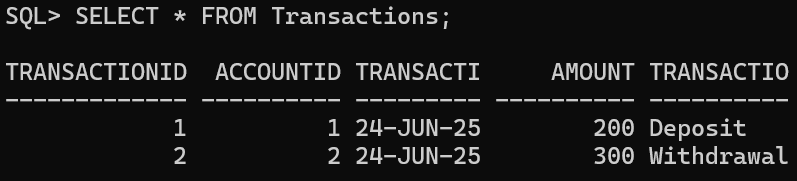
END;

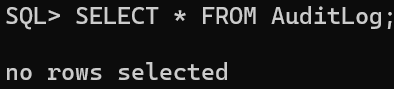
/

* + **Trigger Output:**

****

* + **Before:**

****

****

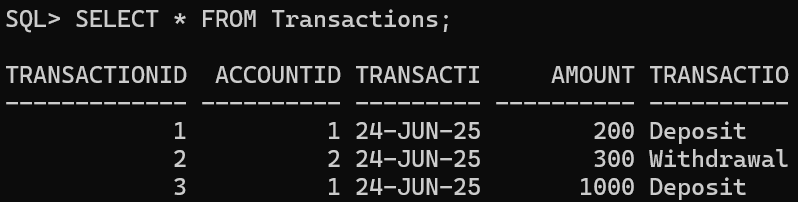
* + **Insert Code:**

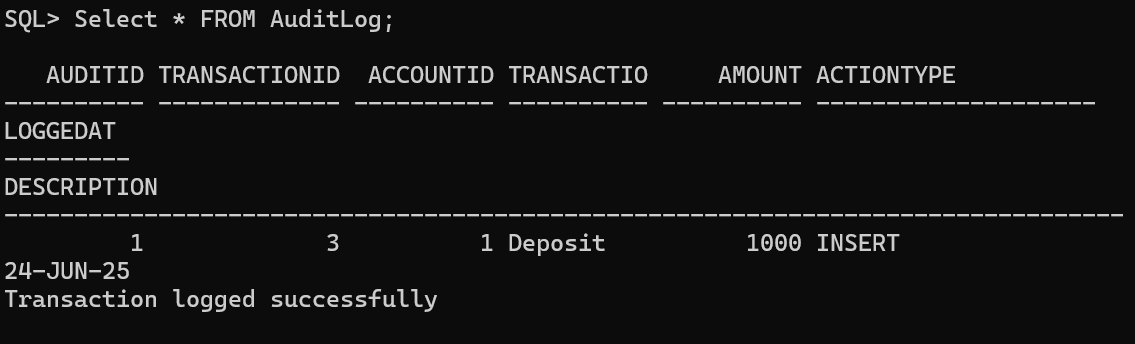
INSERT INTO Transactions VALUES (3, 1, SYSDATE, 1000, 'Deposit');

* + **Insert Output:**

****

* + **After:**

****

****

**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.
  + **Trigger Code:**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

TBalance NUMBER;

BEGIN

SELECT Balance INTO TBalance FROM Accounts WHERE AccountId = :NEW.AccountId;

IF :NEW.TransactionType = 'Withdraw' AND :NEW.Amount > TBalance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Withdrawal amount exceeds account balance');

END IF;

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

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

END IF;

END;

/

* + **Trigger Code Output:**



* + **Case-1:**

Valid Deposit. Amount>0

* Code:

INSERT INTO Transactions VALUES (3, 1, SYSDATE, 500, 'Deposit');

* Output:



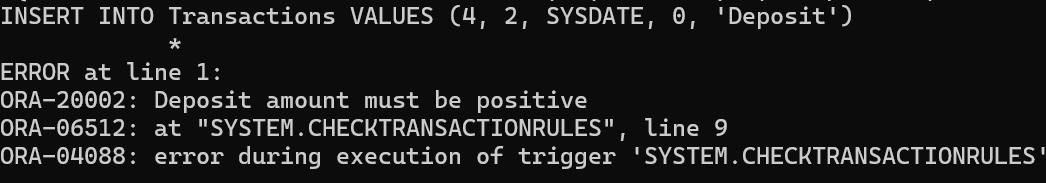
* + **Case-2:**

Invalid Deposit. Amount<=0

* Code:

INSERT INTO Transactions VALUES (4, 2, SYSDATE, 0, 'Deposit');

* Output:



* + **Case-3:**

Valid Withdrawal. Amount ≤ Balance

* Code:

INSERT INTO Transactions VALUES (4, 1, SYSDATE, 10, 'Withdraw');

* Output:



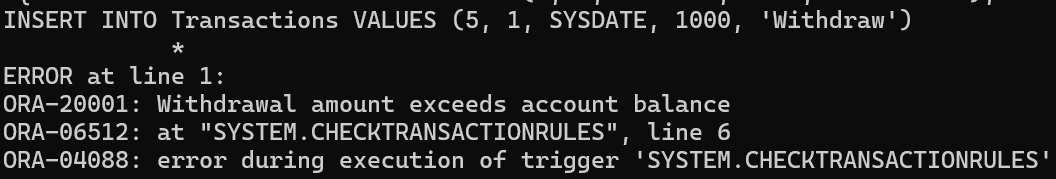
* + **Case-4:**

Invalid Withdrawal. Amount > Balance

* Code:

INSERT INTO transactions2 VALUES (4, 'WITHDRAWAL', 101, 2000, SYSDATE);

* 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:**

DECLARE

PTransactionId NUMBER;

PAccountId VARCHAR2(20);

PTransactionDate DATE;

PAmount NUMBER;

PTransactionType VARCHAR2(20);

CURSOR GenerateMonthlyStatements IS

SELECT \* FROM Transactions WHERE TRUNC(TransactionDate,'MM')=TRUNC(SYSDATE,'MM');

BEGIN

OPEN GenerateMonthlyStatements;

LOOP

FETCH GenerateMonthlyStatements INTO PTransactionId, PAccountId, PTransactionDate, PAmount, PTransactionType;

EXIT WHEN GenerateMonthlyStatements%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Monthly Statement for Transaction with Transaction Id: '||PTransactionId||', Account Id: '||PAccountId||', Transaction Date: '||PTransactionDate||', Amount: '||PAmount||', Transaction Type: '||PTransactionType);

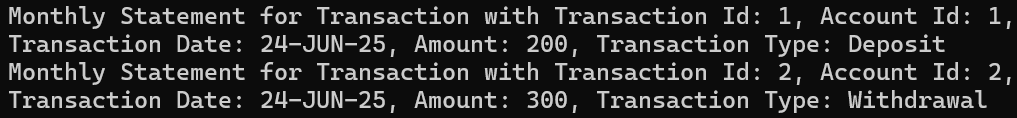
END LOOP;

CLOSE GenerateMonthlyStatements;

END;

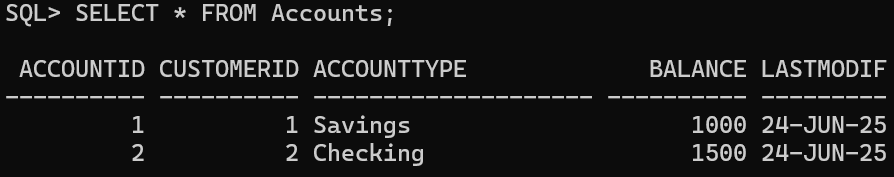
/

* + **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.
  + **Before:**

****

* + **Code:**

DECLARE

PAccountId NUMBER;

CURSOR ApplyAnnualFee IS

SELECT AccountId FROM accounts;

BEGIN

OPEN ApplyAnnualFee;

LOOP

FETCH ApplyAnnualFee INTO PAccountId;

EXIT WHEN ApplyAnnualFee%NOTFOUND;

-- Annual Fee of 50

UPDATE Accounts SET Balance=Balance-50 WHERE AccountId=PAccountId;

END LOOP;

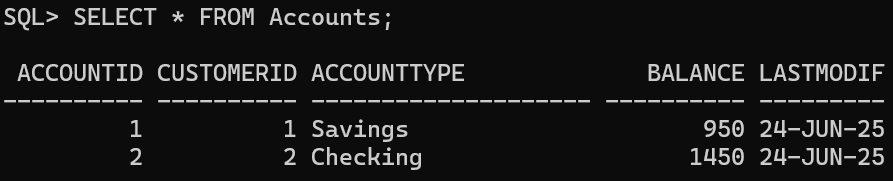
END;

/

* + **Output:**

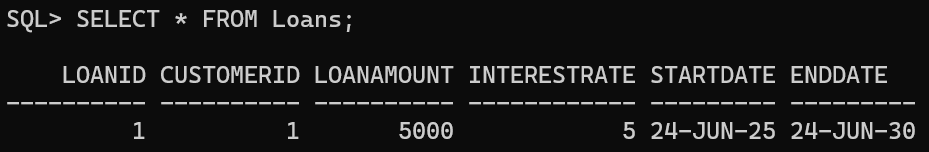
****

* + **After:**

****

**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.
  + **Before:**



* + **Code:**

DECLARE

PLoanId NUMBER;

CURSOR UpdateLoanInterestRates IS

SELECT LoanId FROM loans;

BEGIN

OPEN UpdateLoanInterestRates;

LOOP

FETCH UpdateLoanInterestRates INTO PLoanId;

EXIT WHEN UpdateLoanInterestRates%NOTFOUND;

-- Adding extra 1% to interest rate

UPDATE loans SET InterestRate=InterestRate+1 WHERE LoanId=PLoanId;

END LOOP;

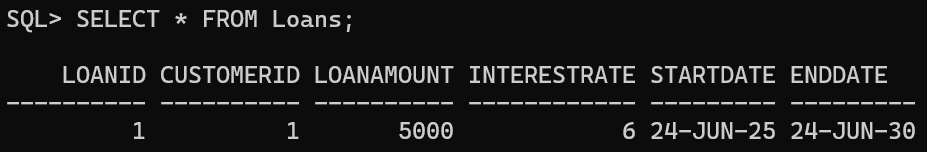
END;

/

* + **Output:**

****

* + **After:**

****

**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.
  + **Package Specification:**

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER);

PROCEDURE UpdateCustomer(p\_id NUMBER, p\_name VARCHAR2);

FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

* + **Package Body:**

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER) IS

BEGIN

INSERT INTO Customers (CUSTOMERID, NAME, DOB, BALANCE, LASTMODIFIED)

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

END;

PROCEDURE UpdateCustomer(p\_id NUMBER, p\_name VARCHAR2) IS

BEGIN

UPDATE Customers

SET NAME = p\_name, LASTMODIFIED = SYSDATE

WHERE CUSTOMERID = p\_id;

END;

FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT BALANCE INTO v\_balance

FROM Customers

WHERE CUSTOMERID = p\_id;

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

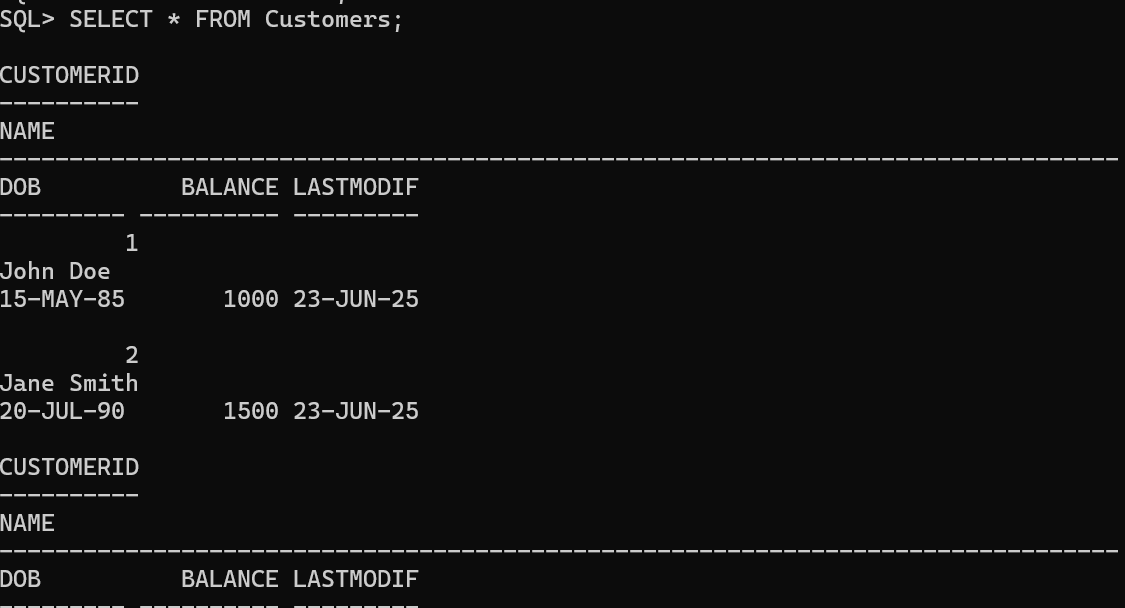
RETURN NULL;

END;

END CustomerManagement;

/

* + **AddCustomer Procedure:**
    - Before:



* + - PL/SQL Code:

BEGIN

CustomerManagement.AddCustomer(

p\_id => 3,

p\_name => 'Mark Wilson',

p\_dob => TO\_DATE('1992-11-10', 'YYYY-MM-DD'),

p\_balance => 2000

);

END;

/

* + - After:



* + **UpdateCustomer Procedure:**
    - **Before:**



* + - **PL/SQL Code:**

BEGIN

CustomerManagement.UpdateCustomer(3, 'Mark W.');

END;

/

* + - **After:**

****

* + **GetCustomerBalance Function:**
    - PL/SQL Code:

DECLARE

v\_balance NUMBER;

BEGIN

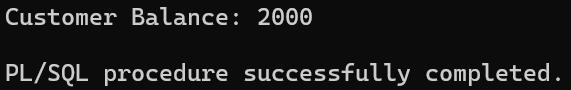
v\_balance := CustomerManagement.GetCustomerBalance(3);

DBMS\_OUTPUT.PUT\_LINE('Customer Balance: ' || v\_balance);

END;

/

* + - 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.
  + **Package Specification:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2, p\_hiredate DATE);

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_position VARCHAR2, p\_salary NUMBER);

FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

* + **Package Body:**

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2, p\_salary NUMBER, p\_department VARCHAR2, p\_hiredate DATE) IS

BEGIN

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

VALUES (p\_id, p\_name, p\_position, p\_salary, p\_department, p\_hiredate);

END;

PROCEDURE UpdateEmployee(p\_id NUMBER, p\_position VARCHAR2, p\_salary NUMBER) IS

BEGIN

UPDATE Employees

SET Position = p\_position, Salary = p\_salary

WHERE EmployeeID = p\_id;

END;

FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

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

RETURN v\_salary \* 12;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

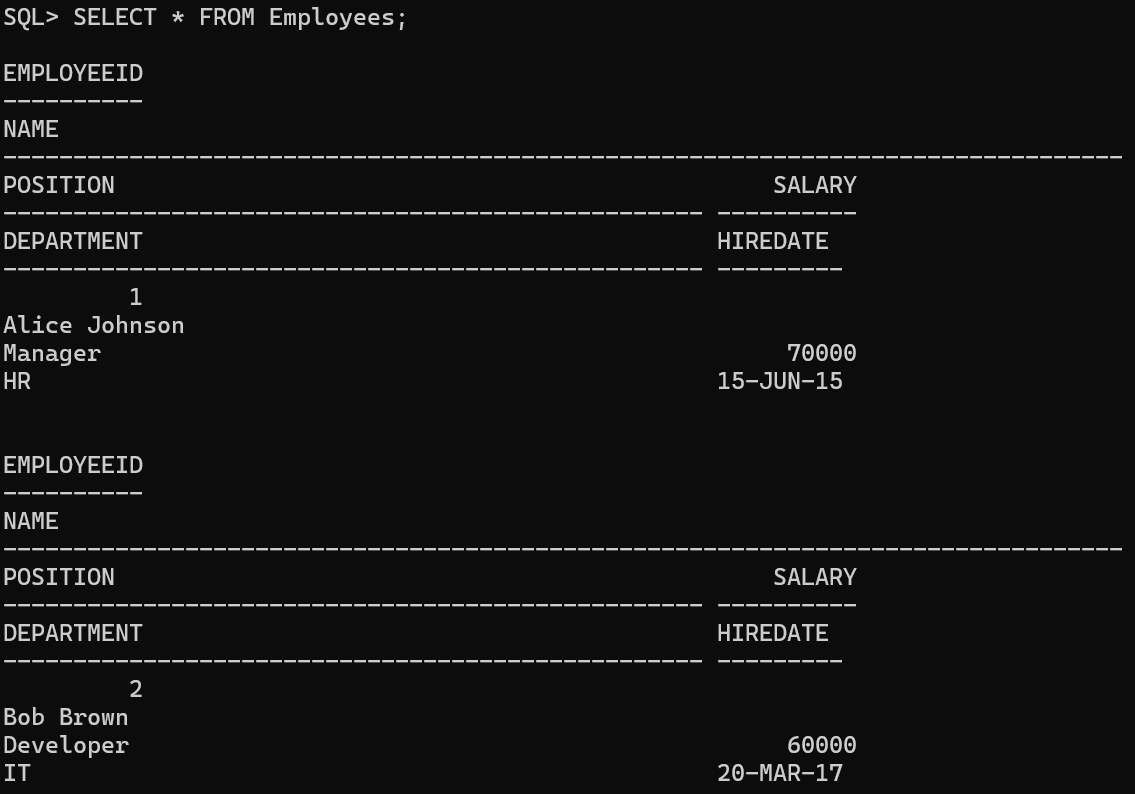
RETURN NULL;

END;

END EmployeeManagement;

/

* + **HireEmployee Procedure:**
    - Before:



* + - PL/SQL Code:

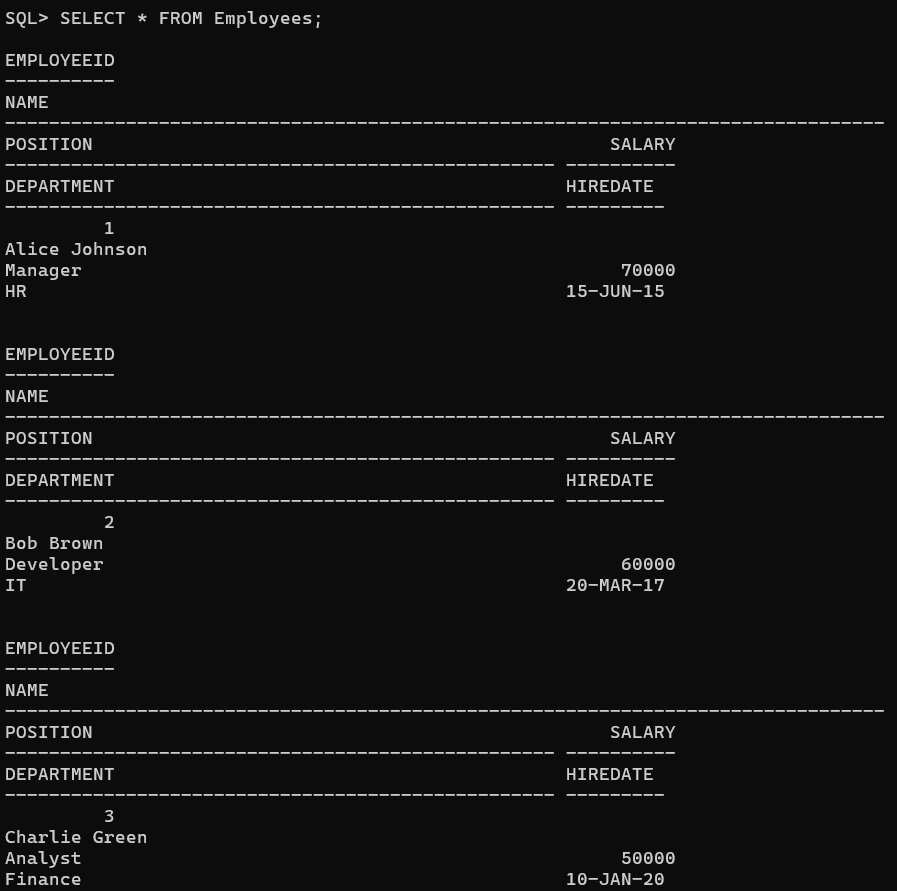
BEGIN

EmployeeManagement.HireEmployee(3, 'Charlie Green', 'Analyst', 50000, 'Finance', TO\_DATE('2020-01-10', 'YYYY-MM-DD'));

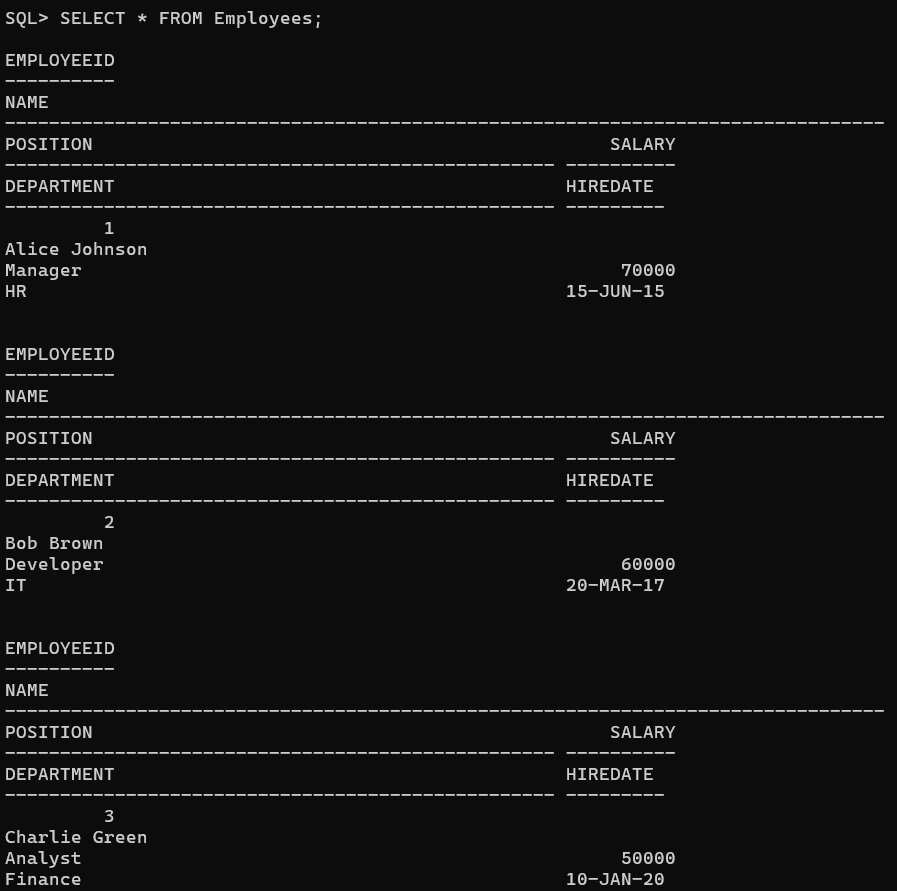
END;

/

* + - After:



* + **UpdateEmployee Procedure:**
    - Before:



* + - PL/SQL Code:

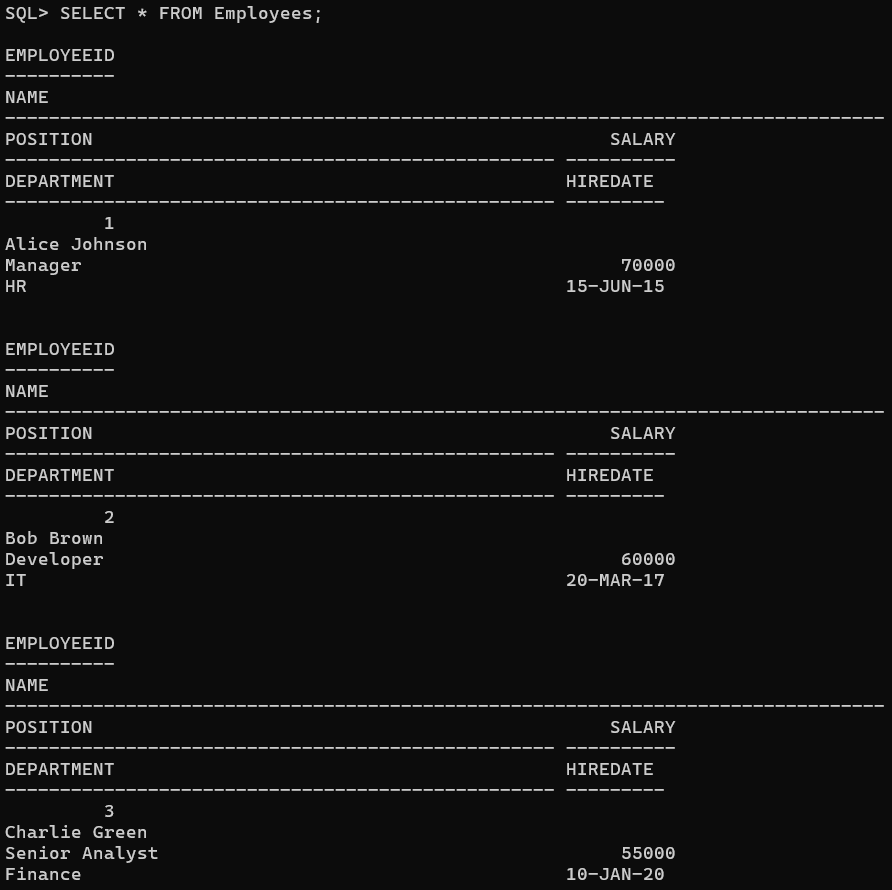
BEGIN

EmployeeManagement.UpdateEmployee(3, 'Senior Analyst', 55000);

END;

/

* + - After:



* + **CalculateAnnualSalary Function:**
    - Pl/SQL Code:

DECLARE

v\_annual\_salary NUMBER;

BEGIN

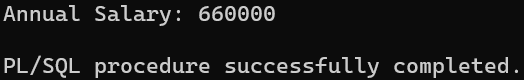
v\_annual\_salary := EmployeeManagement.CalculateAnnualSalary(3);

DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || v\_annual\_salary);

END;

/

* + - 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.
  + **Package Specification:**

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount(p\_accid NUMBER, p\_custid NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

PROCEDURE CloseAccount(p\_accid NUMBER);

FUNCTION GetTotalBalance(p\_custid NUMBER) RETURN NUMBER;

END AccountOperations;

/

* + **Package Body:**

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount(p\_accid NUMBER, p\_custid NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

BEGIN

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)

VALUES (p\_accid, p\_custid, p\_type, p\_balance, SYSDATE);

END;

PROCEDURE CloseAccount(p\_accid NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_accid;

END;

FUNCTION GetTotalBalance(p\_custid NUMBER) RETURN NUMBER IS

v\_total NUMBER;

BEGIN

SELECT SUM(Balance)

INTO v\_total

FROM Accounts

WHERE CustomerID = p\_custid;

RETURN NVL(v\_total, 0);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

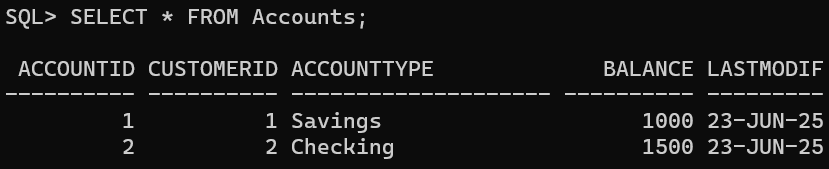
RETURN 0;

END;

END AccountOperations;

/

* + **OpenAccount Procedure:**
    - Before:



* + - PL/SQL Code:

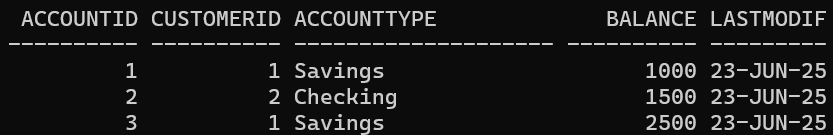
BEGIN

AccountOperations.OpenAccount(3, 1, 'Savings', 2500);

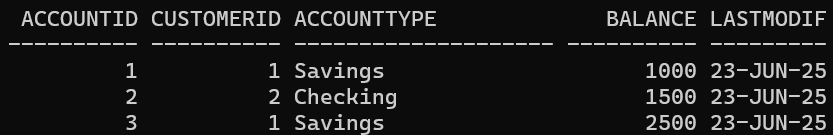
END;

/

* + - After:



* + **CloseAccount Procedure:**
    - Before:



* + - PL/SQL Code:

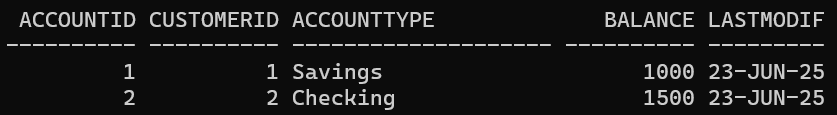
BEGIN

AccountOperations.CloseAccount(3);

END;

/

* + - After:



* + **GetTotalBalance Function:**
    - PL/SQL Code:

DECLARE

v\_total\_balance NUMBER;

BEGIN

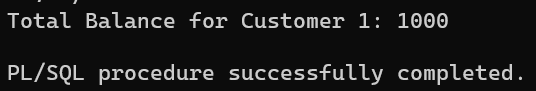
v\_total\_balance := AccountOperations.GetTotalBalance(1);

DBMS\_OUTPUT.PUT\_LINE('Total Balance for Customer 1: ' || v\_total\_balance);

END;

/

* + - Output:



**Schema to be Created**

*CREATE TABLE Customers (*

*CustomerID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*DOB DATE,*

*Balance NUMBER,*

*LastModified DATE*

*);*

*CREATE TABLE Accounts (*

*AccountID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*AccountType VARCHAR2(20),*

*Balance NUMBER,*

*LastModified DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Transactions (*

*TransactionID NUMBER PRIMARY KEY,*

*AccountID NUMBER,*

*TransactionDate DATE,*

*Amount NUMBER,*

*TransactionType VARCHAR2(10),*

*FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)*

*);*

*CREATE TABLE Loans (*

*LoanID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*LoanAmount NUMBER,*

*InterestRate NUMBER,*

*StartDate DATE,*

*EndDate DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Employees (*

*EmployeeID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*Position VARCHAR2(50),*

*Salary NUMBER,*

*Department VARCHAR2(50),*

*HireDate DATE*

*);*

**Example Scripts for Sample Data Insertion**

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

*VALUES (1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);*

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

*VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);*

*INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)*

*VALUES (1, 1, 'Savings', 1000, SYSDATE);*

*INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)*

*VALUES (2, 2, 'Checking', 1500, SYSDATE);*

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

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

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

*VALUES (2, 2, SYSDATE, 300, 'Withdrawal');*

*INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)*

*VALUES (1, 1, 5000, 5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));*

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

*VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO\_DATE('2015-06-15', 'YYYY-MM-DD'));*

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

*VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO\_DATE('2017-03-20', 'YYYY-MM-DD'));*