**WEEK-2: PL/SQL**

## **Exercise-2 Error Handling**

**Question-1:** 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

**Solution:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_from\_account\_id;

IF v\_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;

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

EXCEPTION

WHEN OTHERS THEN

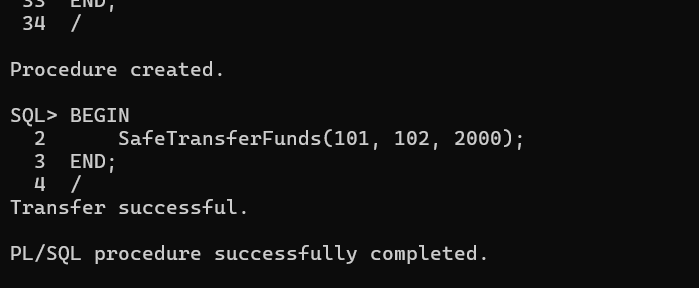
ROLLBACK;

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

END;

/

**Output:**



**Question-2:** 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.

**Solution:**

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 not found.');

END IF;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

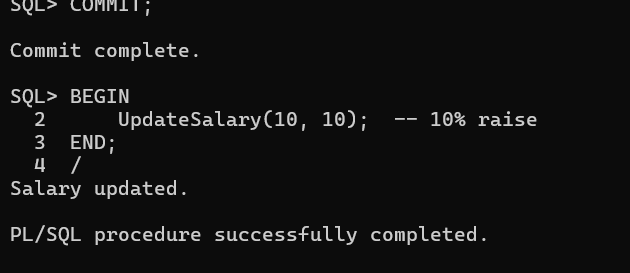
ROLLBACK;

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

END;

/

**Output:**



**Question-3:** 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.

**Solution:**

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER,

p\_isvip IN VARCHAR

) IS

BEGIN

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

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

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Customer added successfully.');

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

WHEN OTHERS THEN

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

ROLLBACK;

END;

/

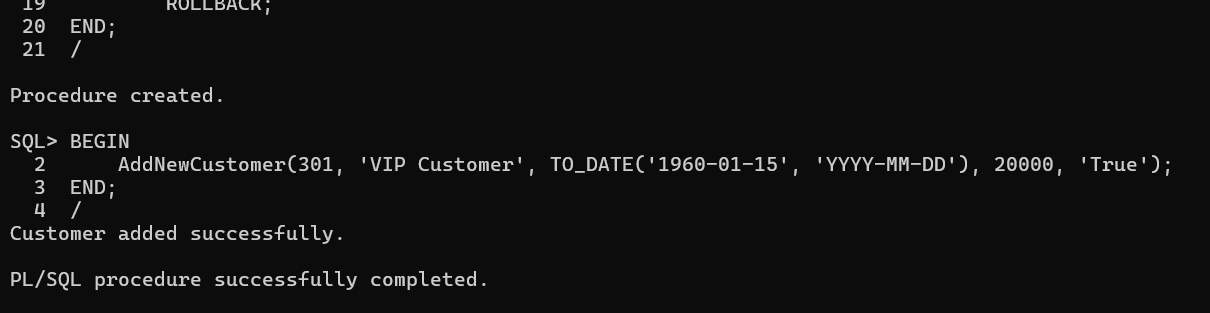
BEGIN

AddNewCustomer(301, 'VIP Customer', TO\_DATE('1960-01-15', 'YYYY-MM-DD'), 20000, 'True');

END;

/

**Output:**



**Exercise 4: Functions**

**Question-1:** Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

**Solution:**

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);

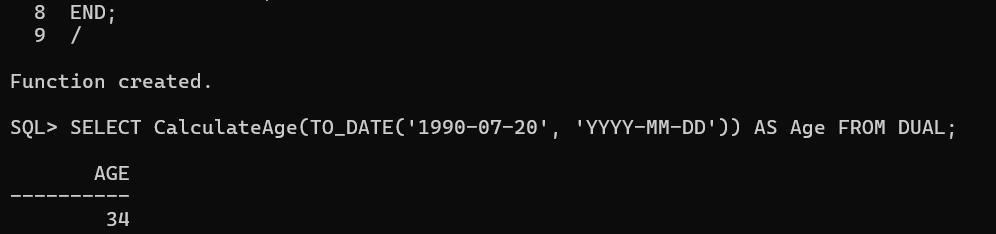
RETURN v\_age;

END;

/

SELECT CalculateAge(TO\_DATE('1990-07-20', 'YYYY-MM-DD')) AS Age FROM DUAL;

**Output:**



**Question-2:** Write a function **CalculateMonthlyInstallment** that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

**Solution:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount IN NUMBER,

p\_annual\_rate IN NUMBER,

p\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_rate NUMBER;

v\_months NUMBER;

v\_emi NUMBER;

BEGIN

v\_monthly\_rate := p\_annual\_rate / (12 \* 100); -- Convert annual rate to monthly decimal

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

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

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

RETURN ROUND(v\_emi, 2);

END;

/

**PL/SQL Block to Call:**

DECLARE

v\_loan\_amount Loans.LoanAmount%TYPE;

v\_interest\_rate Loans.InterestRate%TYPE;

v\_start\_date Loans.StartDate%TYPE;

v\_end\_date Loans.EndDate%TYPE;

v\_years NUMBER;

v\_emi NUMBER;

BEGIN

FOR rec IN (

SELECT LoanID, LoanAmount, InterestRate, StartDate, EndDate

FROM Loans

) LOOP

-- Calculate duration in years

v\_years := ROUND(MONTHS\_BETWEEN(rec.EndDate, rec.StartDate) / 12, 1);

-- Calculate EMI

v\_emi := CalculateMonthlyInstallment(rec.LoanAmount, rec.InterestRate, v\_years);

-- Output result

DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || rec.LoanID ||

' | EMI: ' || v\_emi ||

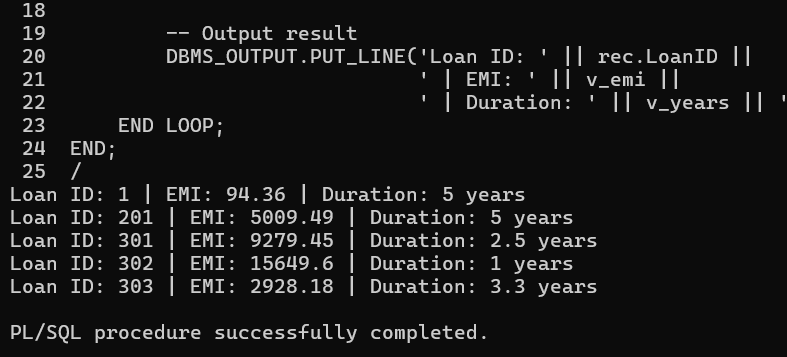
' | Duration: ' || v\_years || ' years');

END LOOP;

END;

/

**Output:**



**Question-3:** 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

**Solution:**

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_account\_id IN NUMBER,

p\_required\_amount IN NUMBER

) RETURN BOOLEAN IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_account\_id;

RETURN v\_balance >= p\_required\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

WHEN OTHERS THEN

RETURN FALSE;

END;

/

**PL/SQL Block to Call:**

DECLARE

result BOOLEAN;

BEGIN

result := HasSufficientBalance(101, 500);

IF result THEN

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

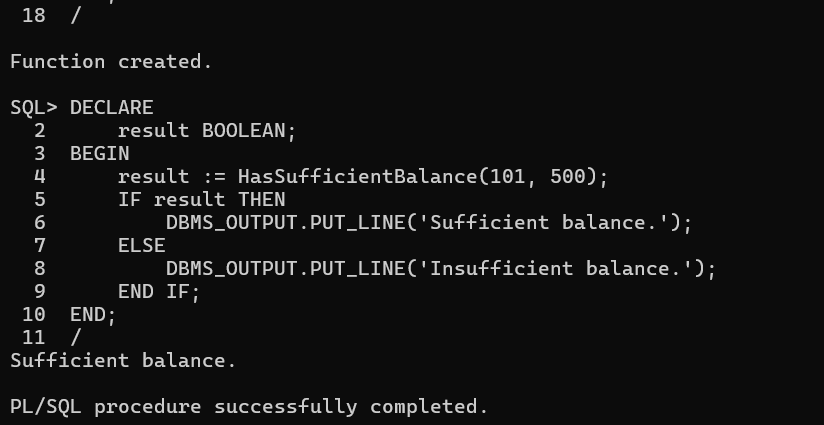
ELSE

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

END IF;

END;

/

**Output:**

### Exercise 5: Triggers

**Question-1:** Write a trigger UpdateCustomerLastModified that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.

**Solution:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

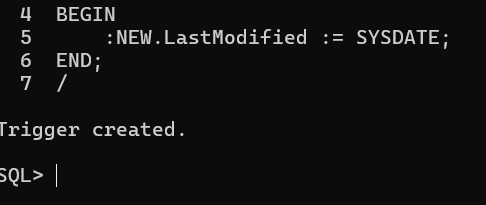
BEGIN

:NEW.LastModified := SYSDATE;

END;

/

**Output:**



**Question-2:** Write a trigger **LogTransaction** that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

**Solution:**

**Step 1: Create AuditLog table:**

CREATE TABLE AuditLog (

LogID NUMBER PRIMARY KEY,

TransactionID NUMBER,

LogDate DATE,

Message VARCHAR2(200)

);

**Step 2: Create Trigger:**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (LogID, TransactionID, LogDate, Message)

VALUES (

AuditLog\_Seq.NEXTVAL,

:NEW.TransactionID,

SYSDATE,

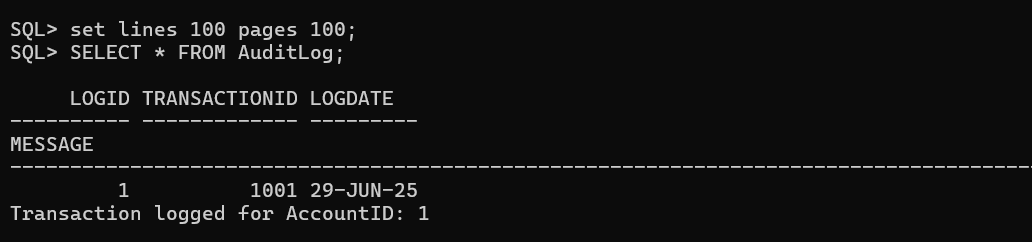
'Transaction logged for account ' || :NEW.AccountID

);

END;

/

**Output:**



**Question-3:** Write a trigger CheckTransactionRules that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.

**Solution:**

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(-20001, 'Withdrawal exceeds account balance.');

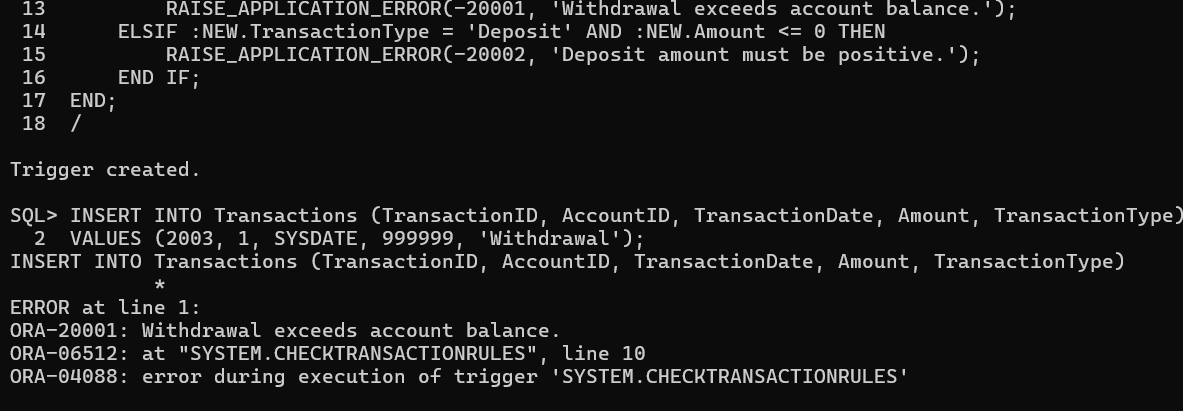
ELSIF :NEW.TransactionType = 'Deposit' AND :NEW.Amount <= 0 THEN

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

END IF;

END;

/

**Output:**

### Exercise 6: Cursors

**Question-1:** Write a PL/SQL block using an explicit cursor **Generate Monthly Statements** that retrieves all transactions for the current month and prints a statement for each customer.

**Solution:**

DECLARE

CURSOR txn\_cursor IS

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

FROM Customers c

JOIN Accounts a ON c.CustomerID = a.CustomerID

JOIN Transactions t ON a.AccountID = t.AccountID

WHERE TO\_CHAR(t.TransactionDate, 'MM-YYYY') = TO\_CHAR(SYSDATE, 'MM-YYYY')

ORDER BY c.CustomerID;

BEGIN

FOR rec IN txn\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Customer: ' || rec.Name ||

', Account: ' || rec.AccountID ||

', ' || rec.TransactionType ||

' of ₹' || rec.Amount ||

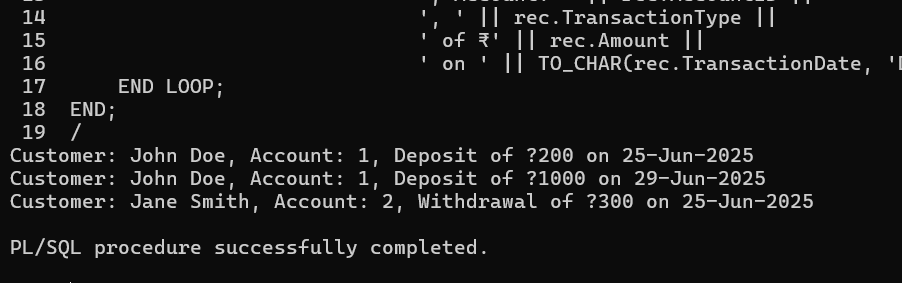
' on ' || TO\_CHAR(rec.TransactionDate, 'DD-Mon-YYYY'));

END LOOP;

END;

/

**Output:**



**Question-2:** Write a PL/SQL block using an explicit cursor **ApplyAnnualFee** that deducts an annual maintenance fee from the balance of all accounts.

**Solution:**

DECLARE

CURSOR acc\_cursor IS

SELECT AccountID, Balance FROM Accounts;

v\_fee CONSTANT NUMBER := 250;

BEGIN

FOR rec IN acc\_cursor LOOP

IF rec.Balance >= v\_fee THEN

UPDATE Accounts

SET Balance = Balance - v\_fee

WHERE AccountID = rec.AccountID;

END IF;

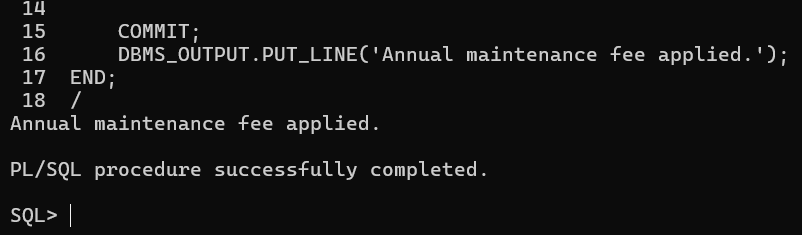
END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Annual maintenance fee applied.');

END;

/

**Output:**

**Question-3:** Write a PL/SQL block using an explicit cursor Update Loan InterestRates that fetches all loans and updates their interest rates based on the new policy.

**Solution:**

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate FROM Loans;

v\_discount CONSTANT NUMBER := 0.5;

BEGIN

FOR rec IN loan\_cursor LOOP

UPDATE Loans

SET InterestRate = rec.InterestRate - v\_discount

WHERE LoanID = rec.LoanID;

END LOOP;

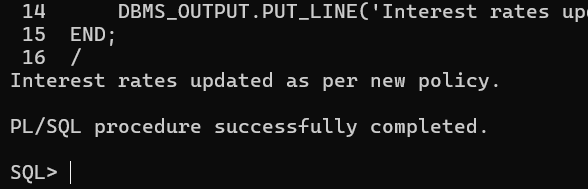
COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Interest rates updated as per new policy.');

END;

/

**Output:**



### Exercise 7: Packages

**Question-1:** Create a package **CustomerManagement** with procedures for adding a new customer, updating customer details, and a function to get customer balance.

**Solution:**

CREATE OR REPLACE PACKAGE CustomerManagement IS

PROCEDURE AddCustomer(

p\_id NUMBER,

p\_name VARCHAR2,

p\_dob DATE,

p\_balance NUMBER,

p\_isvip VARCHAR2

);

PROCEDURE UpdateCustomerName(p\_id NUMBER, p\_new\_name VARCHAR2);

FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;

END;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement IS

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

BEGIN

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

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

COMMIT;

END;

PROCEDURE UpdateCustomerName(p\_id NUMBER, p\_new\_name VARCHAR2) IS

BEGIN

UPDATE Customers SET Name = p\_new\_name WHERE CustomerID = p\_id;

COMMIT;

END;

FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Customers WHERE CustomerID = p\_id;

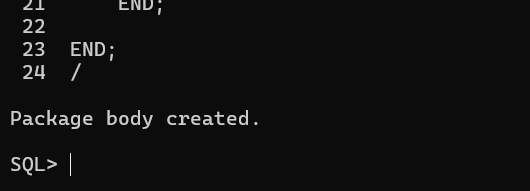
RETURN v\_balance;

END;

END;

/

**Output:**

****

**Question-2:** Write a package **EmployeeManagement** with procedures to hire new employees, update employee details, and a function to calculate annual salary.

**Solution:**

CREATE OR REPLACE PACKAGE EmployeeManagement IS

PROCEDURE HireEmployee(

p\_id NUMBER,

p\_name VARCHAR2,

p\_position VARCHAR2,

p\_salary NUMBER,

p\_dept VARCHAR2,

p\_hiredate DATE

);

PROCEDURE UpdateEmployeeDetails(p\_id NUMBER, p\_salary NUMBER, p\_dept VARCHAR2);

FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS

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

COMMIT;

END;

PROCEDURE UpdateEmployeeDetails(p\_id NUMBER, p\_salary NUMBER, p\_dept VARCHAR2) IS

BEGIN

UPDATE Employees SET Salary = p\_salary, Department = p\_dept WHERE EmployeeID = p\_id;

COMMIT;

END;

FUNCTION CalculateAnnualSalary(p\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

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

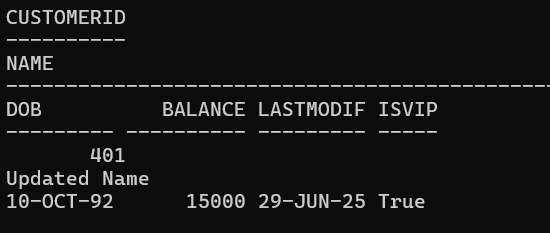
RETURN v\_salary \* 12;

END;

END;

/

**Output:**



**Question-3:** 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.

**Solution:**

CREATE OR REPLACE PACKAGE AccountOperations IS

PROCEDURE OpenAccount(

p\_accid NUMBER,

p\_custid NUMBER,

p\_type VARCHAR2,

p\_balance NUMBER

);

PROCEDURE CloseAccount(p\_accid NUMBER);

FUNCTION GetCustomerTotalBalance(p\_custid NUMBER) RETURN NUMBER;

END;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations IS

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);

COMMIT;

END;

PROCEDURE CloseAccount(p\_accid NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_accid;

COMMIT;

END;

FUNCTION GetCustomerTotalBalance(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);

END;

END;

/

**Output:**

