**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.

QUERY🡪

SET SERVEROUTPUT ON;

DECLARE

v\_customer\_id Customers.CustomerID%TYPE;

v\_dob Customers.DOB%TYPE;

v\_age NUMBER;

BEGIN

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

v\_customer\_id := rec.CustomerID;

v\_dob := rec.DOB;

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

IF v\_age > 60 THEN

UPDATE Loans SET InterestRate = InterestRate - 1 WHERE CustomerID = v\_customer\_id;

DBMS\_OUTPUT.PUT\_LINE('1% DISCOUNT APPLIED TO CUSTOMER ID: ' || v\_customer\_id);

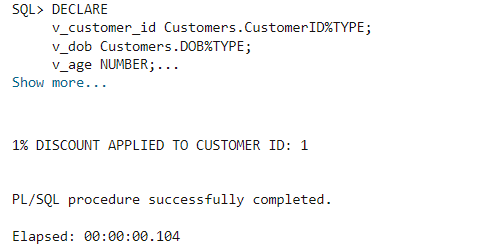
END IF;

END LOOP;

COMMIT;

END;

/



**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.

QUERY🡪

BEGIN

FOR cust\_rec IN (SELECT CustomerID, Balance FROM Customers) LOOP

IF cust\_rec.Balance > 10000 THEN

UPDATE Customers SET IsVIP = 'TRUE' WHERE CustomerID = cust\_rec.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || cust\_rec.CustomerID || ' promoted to VIP.');

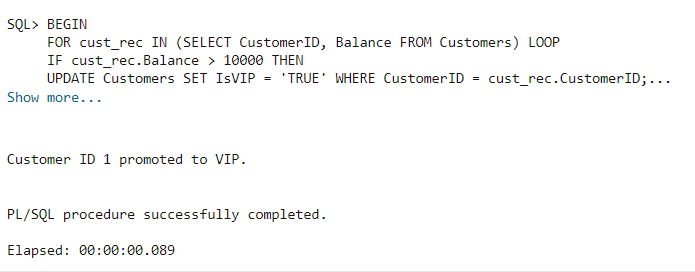
END IF;

END LOOP;

COMMIT;

END;

/



**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.

QUERY🡪

SET SERVEROUTPUT ON

DECLARE

BEGIN

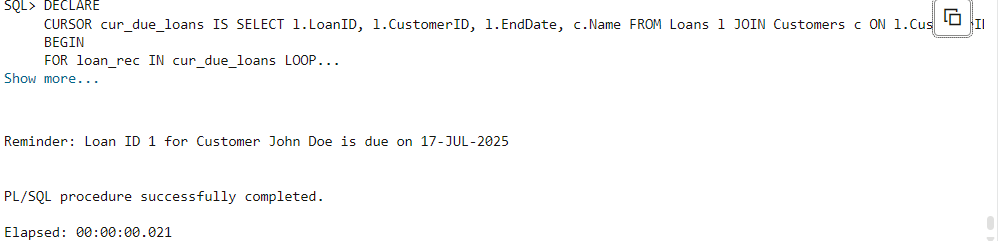
FOR loan\_rec IN (SELECT l.LoanID, l.CustomerID, l.EndDate, c.Name FROM Loans l JOIN Customers c ON l.CustomerID = c.CustomerID WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30) LOOP

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

END LOOP;

END;

/



**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.

QUERY🡪

CREATE OR REPLACE PROCEDURE SafeTransferFunds(p\_from NUMBER, p\_to NUMBER, p\_amount NUMBER) IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from;

IF v\_balance < p\_amount THEN

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

END IF;

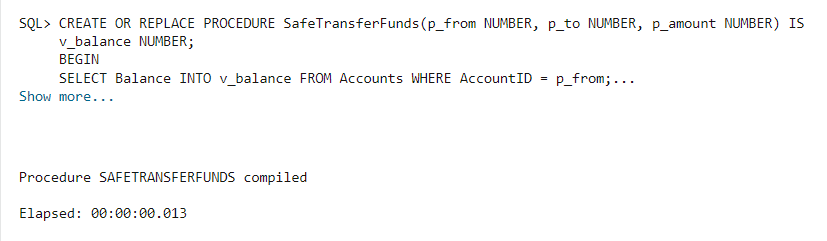
UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from;

UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to;

COMMIT;

END;

/



**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.

QUERY🡪

CREATE OR REPLACE PROCEDURE UpdateSalary(p\_emp\_id NUMBER, p\_percentage NUMBER) IS

BEGIN

UPDATE Employees SET Salary = Salary + (Salary \* p\_percentage / 100) WHERE EmployeeID = p\_emp\_id;

IF SQL%ROWCOUNT = 0 THEN

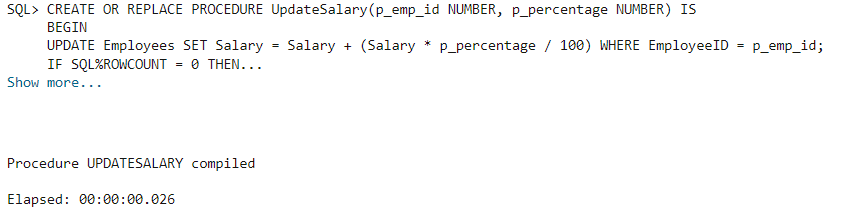
RAISE\_APPLICATION\_ERROR(-20002, 'Employee does not exist');

END IF;

COMMIT;

END;

/



**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.

QUERY🡪

CREATE OR REPLACE PROCEDURE AddNewCustomer(p\_id NUMBER, p\_name VARCHAR2, p\_dob DATE, p\_balance NUMBER) IS

BEGIN

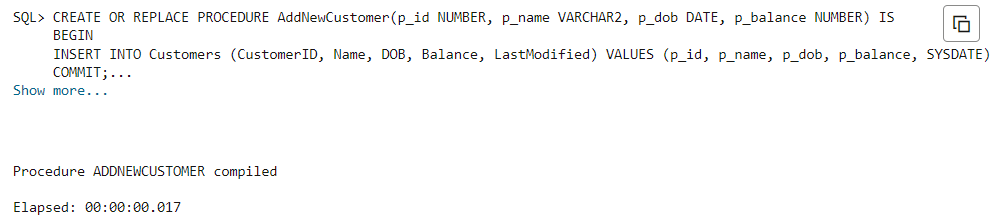
INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (p\_id, p\_name, p\_dob, p\_balance, SYSDATE);

COMMIT;

EXCEPTION WHEN DUP\_VAL\_ON\_INDEX THEN RAISE\_APPLICATION\_ERROR(-20003, 'Customer with same ID already exists');

END;

/



**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.

QUERY🡪

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

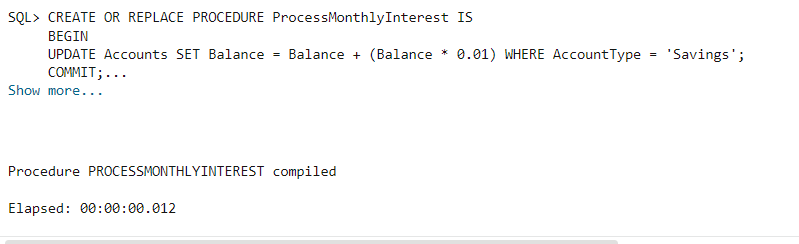
BEGIN

UPDATE Accounts SET Balance = Balance + (Balance \* 0.01) WHERE AccountType = 'Savings';

COMMIT;

END;

/



**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

QUERY🡪

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(p\_dept VARCHAR2, p\_bonus NUMBER) IS

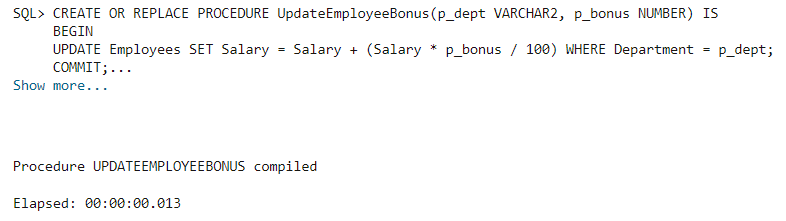
BEGIN

UPDATE Employees SET Salary = Salary + (Salary \* p\_bonus / 100) WHERE Department = p\_dept;

COMMIT;

END;

/



**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.

QUERY🡪

CREATE OR REPLACE PROCEDURE TransferFunds(p\_from NUMBER, p\_to NUMBER, p\_amount NUMBER) IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_from;

IF v\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Insufficient balance');

END IF;

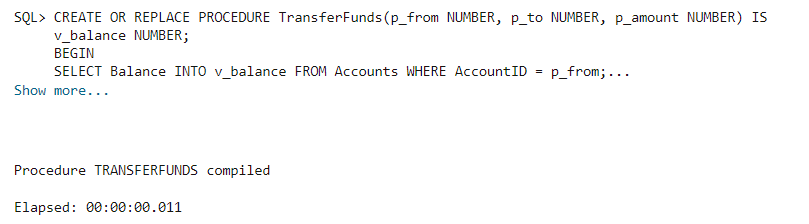
UPDATE Accounts SET Balance = Balance - p\_amount WHERE AccountID = p\_from;

UPDATE Accounts SET Balance = Balance + p\_amount WHERE AccountID = p\_to;

COMMIT;

END;

/



**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.

QUERY🡪

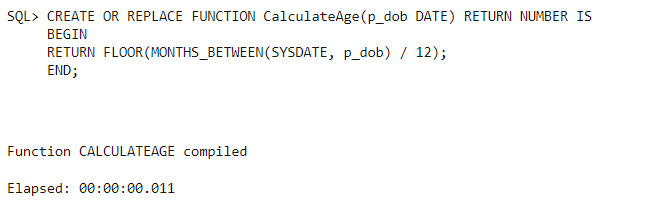
CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE) RETURN NUMBER IS

BEGIN

RETURN FLOOR(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

END;

/



**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.

QUERY🡪

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(p\_amount NUMBER, p\_rate NUMBER, p\_years NUMBER) RETURN NUMBER IS

v\_monthly NUMBER;

v\_n NUMBER;

v\_r NUMBER;

BEGIN

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

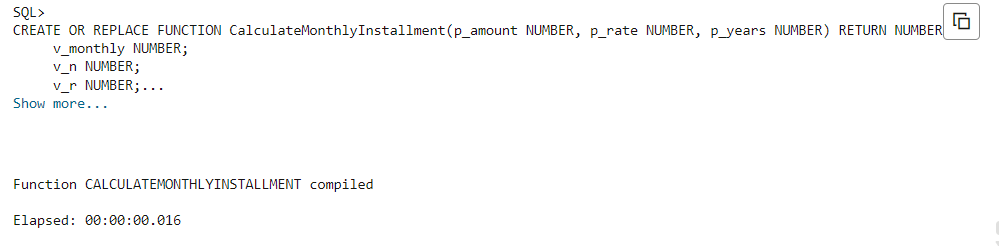
v\_r := p\_rate / 1200;

v\_monthly := (p\_amount \* v\_r) / (1 - POWER(1 + v\_r, -v\_n));

RETURN v\_monthly;

END;

/



**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.

QUERY🡪

CREATE OR REPLACE FUNCTION HasSufficientBalance(p\_acc NUMBER, p\_amt NUMBER) RETURN BOOLEAN IS

v\_balance NUMBER;

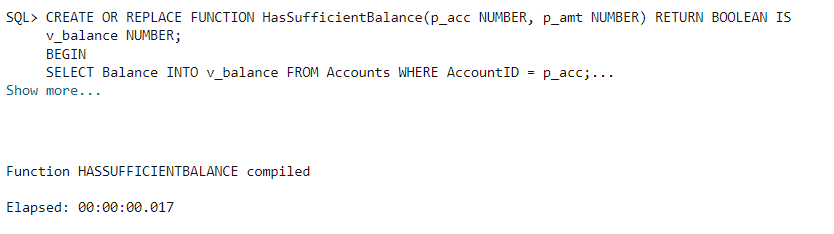
BEGIN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = p\_acc;

RETURN v\_balance >= p\_amt;

END;

/



**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.

QUERY🡪

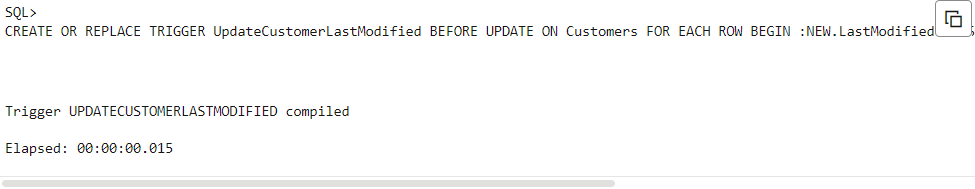
CREATE OR REPLACE TRIGGER UpdateCustomerLastModified BEFORE UPDATE ON Customers FOR EACH ROW

BEGIN

NEW.LastModified := SYSDATE;

END;

/



**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.

QUERY🡪

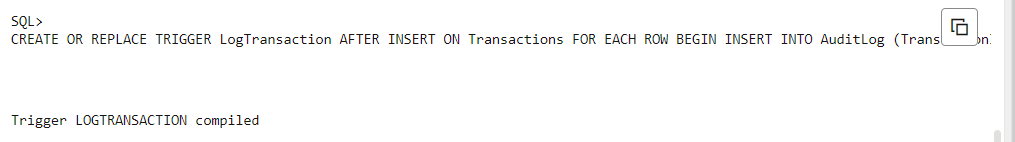
CREATE OR REPLACE TRIGGER LogTransaction AFTER INSERT ON Transactions

FOR EACH ROW BEGIN

INSERT INTO AuditLog (TransactionID, LogDate) VALUES (:NEW.TransactionID, SYSDATE);

END;

/



**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.

QUERY🡪

SET SERVEROUTPUT ON

DECLARE

BEGIN

FOR loan\_rec IN (SELECT l.LoanID, l.CustomerID, l.EndDate, c.Name 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 ' || loan\_rec.LoanID || ' for Customer ' || loan\_rec.Name || ' is due on ' || TO\_CHAR(loan\_rec.EndDate, 'DD-MON-YYYY'));

END LOOP;

END;

/

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

IF :NEW.TransactionType = 'Withdrawal' THEN

SELECT Balance INTO v\_balance FROM Accounts WHERE AccountID = :NEW.AccountID;

IF :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20005, 'Withdrawal exceeds balance');

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

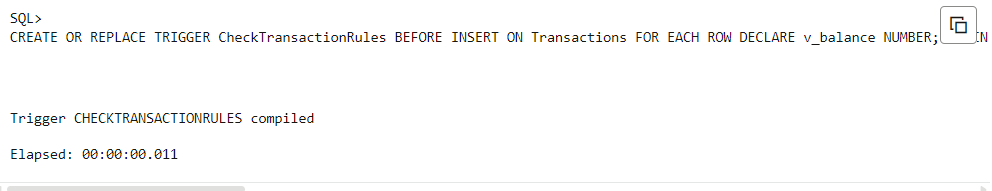
RAISE\_APPLICATION\_ERROR(-20006, 'Deposit must be positive');

END IF;

END IF;

END;

/



**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.

QUERY🡪

DECLARE

CURSOR cur\_trans IS SELECT t.TransactionID, c.Name, t.Amount FROM Transactions t JOIN Accounts a ON t.AccountID = a.AccountID JOIN Customers c ON a.CustomerID = c.CustomerID WHERE EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE) AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE);

BEGIN

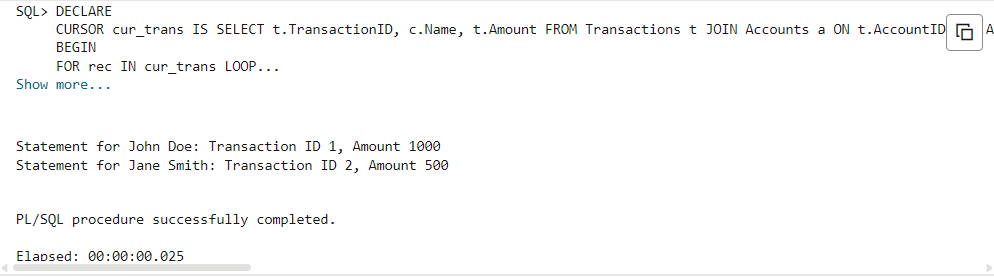
FOR rec IN cur\_trans LOOP

DBMS\_OUTPUT.PUT\_LINE('Statement for ' || rec.Name || ': Transaction ID ' || r ec.TransactionID || ', Amount ' || rec.Amount);

END LOOP;

END;

/



**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.

QUERY🡪

DECLARE

CURSOR cur\_accounts IS SELECT AccountID FROM Accounts;

BEGIN

FOR rec IN cur\_accounts LOOP

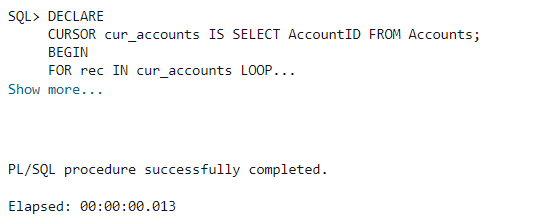
UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = rec.AccountID;

END LOOP;

COMMIT;

END;

/



**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.

QUERY🡪

DECLARE

CURSOR cur\_loans IS SELECT LoanID FROM Loans;

BEGIN

FOR rec IN cur\_loans LOOP

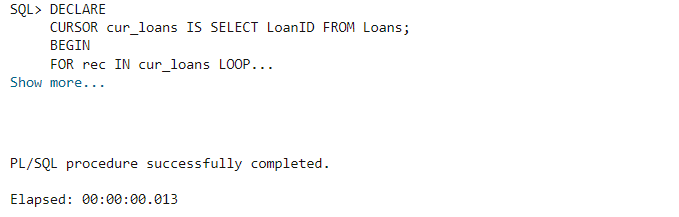
UPDATE Loans SET InterestRate = InterestRate + 0.5 WHERE LoanID = rec.LoanID;

END LOOP;

COMMIT;

END;

/



**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.

QUERY🡪

CREATE OR REPLACE PACKAGE CustomerManagement IS

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;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement IS

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 WHERE CustomerID = p\_id;

END;

FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER IS v\_balance NUMBER;

BEGIN

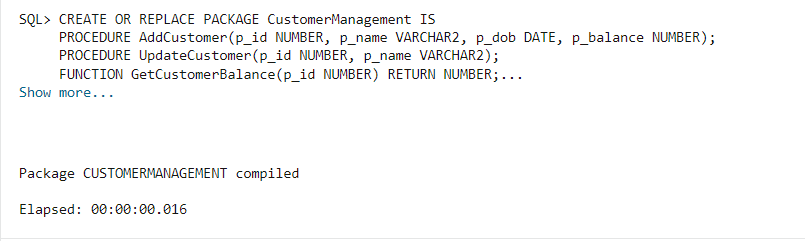
SELECT Balance INTO v\_balance FROM Customers WHERE CustomerID = p\_id;

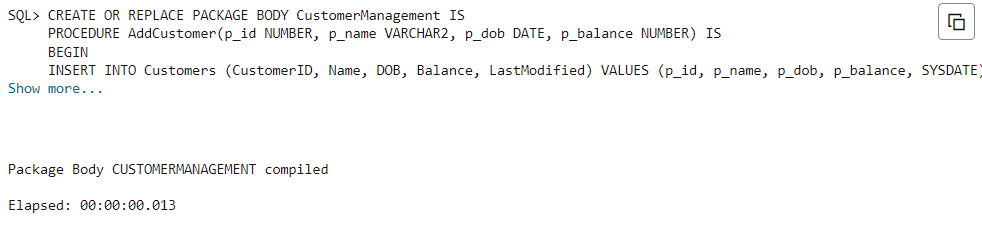
RETURN v\_balance;

END;

END CustomerManagement;

/





**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.

QUERY🡪

CREATE OR REPLACE PACKAGE EmployeeManagement IS

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

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

FUNCTION AnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement IS

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

BEGIN

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (p\_id, p\_name, p\_position, p\_salary, p\_department, SYSDATE);

END;

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

BEGIN

UPDATE Employees SET Name = p\_name WHERE EmployeeID = p\_id;

END;

FUNCTION AnnualSalary(p\_id NUMBER) RETURN NUMBER IS v\_salary NUMBER;

BEGIN

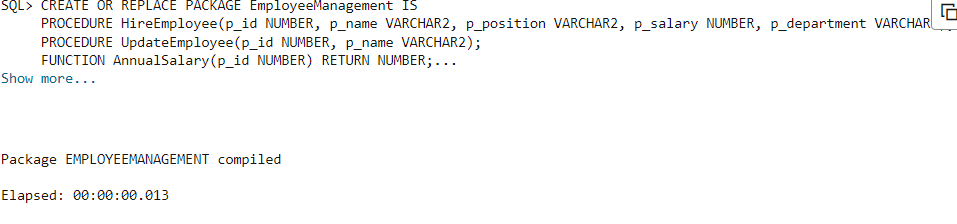
SELECT Salary INTO v\_salary FROM Employees WHERE EmployeeID = p\_id;

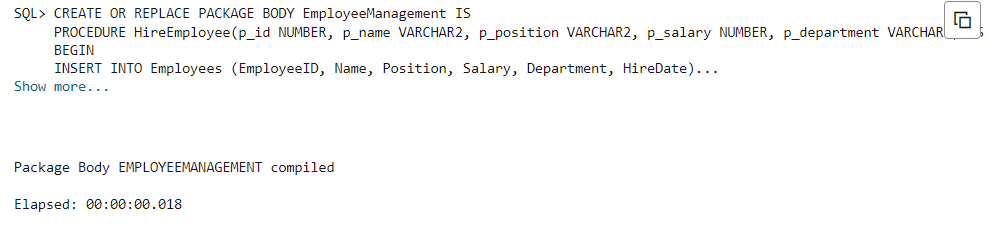
RETURN v\_salary \* 12;

END;

END EmployeeManagement;

/





**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.

QUERY🡪

CREATE OR REPLACE PACKAGE AccountOperations IS

PROCEDURE OpenAccount(p\_id NUMBER, p\_cust NUMBER, p\_type VARCHAR2, p\_balance NUMBER);

PROCEDURE CloseAccount(p\_id NUMBER);

FUNCTION TotalBalance(p\_cust NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations IS

PROCEDURE OpenAccount(p\_id NUMBER, p\_cust NUMBER, p\_type VARCHAR2, p\_balance NUMBER) IS

BEGIN

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

VALUES (p\_id, p\_cust, p\_type, p\_balance, SYSDATE);

END;

PROCEDURE CloseAccount(p\_id NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_id;

END;

FUNCTION TotalBalance(p\_cust NUMBER) RETURN NUMBER IS

v\_total NUMBER;

BEGIN

SELECT SUM(Balance) INTO v\_total FROM Accounts WHERE CustomerID = p\_cust;

RETURN v\_total;

END;

END AccountOperations;

/

