**WEEK 2**

**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.**

ALTER TABLE Customers ADD IsVIP VARCHAR2(5);

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

VALUES (3, 'Elder John', TO\_DATE('1950-01-01', 'YYYY-MM-DD'), 8000, SYSDATE);

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

VALUES (2, 3, 10000, 6, SYSDATE, ADD\_MONTHS(SYSDATE, 60));

SET SERVEROUTPUT ON;

DECLARE

v\_age NUMBER;

BEGIN

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

v\_age := FLOOR(MONTHS\_BETWEEN(SYSDATE, cust.DOB) / 12);

IF v\_age > 60 THEN

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = cust.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('Applied 1% discount for Customer ID: ' || cust.CustomerID);

END IF;

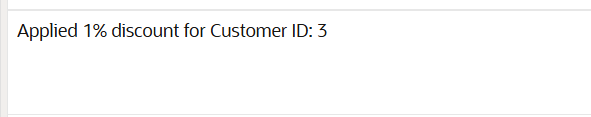
END LOOP;

COMMIT;

END;

/

**OUTPUT:**



**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.**

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

VALUES (4, 'VIP Lisa', TO\_DATE('1995-02-10', 'YYYY-MM-DD'), 15000, SYSDATE);

BEGIN

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

IF cust.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = cust.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('Customer ' || cust.CustomerID || ' marked as VIP');

ELSE

UPDATE Customers

SET IsVIP = 'FALSE'

WHERE CustomerID = cust.CustomerID;

END IF;

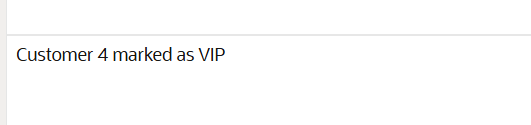
END LOOP;

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.

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

VALUES (3, 4, 3000, 5, SYSDATE - 330, SYSDATE + 10); -- due in 10 days

BEGIN

FOR l IN (

SELECT l.LoanID, l.CustomerID, 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 ' || l.LoanID || ' for customer ' || l.Name ||

' is due on ' || TO\_CHAR(l.EndDate, 'YYYY-MM-DD')

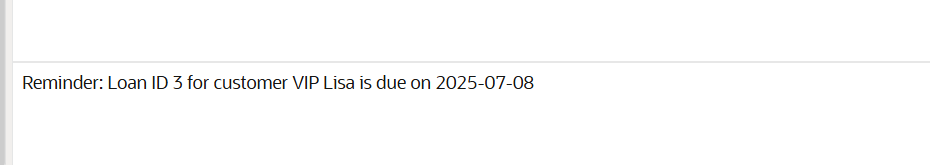
);

END LOOP;

END;

/

**OUTPUT:**

****

**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.**

UPDATE Accounts SET Balance = 1000 WHERE AccountID = 1;

UPDATE Accounts SET Balance = 1500 WHERE AccountID = 2;

COMMIT;

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

v\_balance NUMBER;

BEGIN

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

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;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

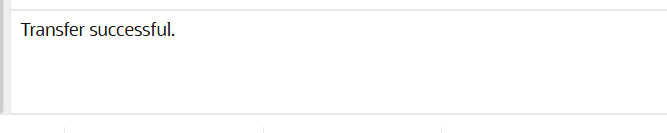
BEGIN

SafeTransferFunds(1, 2, 500);

END;

/

**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.

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_emp\_id IN NUMBER,

p\_percent IN NUMBER

) AS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* p\_percent / 100)

WHERE EmployeeID = p\_emp\_id;

IF SQL%ROWCOUNT = 0 THEN

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

END IF;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

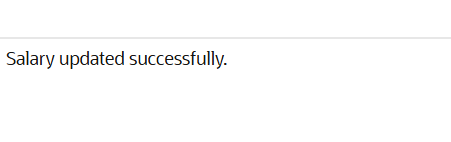
BEGIN

UpdateSalary(1, 10);

END;

/

**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.**

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) AS

BEGIN

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

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

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

BEGIN

AddNewCustomer(10, 'New Customer', TO\_DATE('1999-01-01', 'YYYY-MM-DD'), 1200);

END;

/

**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.**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

FOR acc IN (

SELECT AccountID, Balance

FROM Accounts

WHERE AccountType = 'Savings'

) LOOP

UPDATE Accounts

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

LastModified = SYSDATE

WHERE AccountID = acc.AccountID;

DBMS\_OUTPUT.PUT\_LINE('Interest added to Account ID: ' || acc.AccountID);

END LOOP;

COMMIT;

END;

/

UPDATE Accounts SET Balance = 1000 WHERE AccountID = 1; -- Savings

COMMIT;

BEGIN

ProcessMonthlyInterest;

END;

/

**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**.

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department IN VARCHAR2,

p\_bonus\_percent IN NUMBER

) IS

BEGIN

FOR emp IN (

SELECT EmployeeID, Salary

FROM Employees

WHERE Department = p\_department

) LOOP

UPDATE Employees

SET Salary = Salary + (emp.Salary \* p\_bonus\_percent / 100)

WHERE EmployeeID = emp.EmployeeID;

DBMS\_OUTPUT.PUT\_LINE('Bonus applied to Employee ID: ' || emp.EmployeeID);

END LOOP;

COMMIT;

END;

/

**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.**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_acc IN NUMBER,

p\_to\_acc IN NUMBER,

p\_amount IN NUMBER

) IS

v\_balance NUMBER;

BEGIN

-- Check if balance is sufficient

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

IF v\_balance < p\_amount THEN

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

END IF;

UPDATE Accounts

SET Balance = Balance - p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_from\_acc;

UPDATE Accounts

SET Balance = Balance + p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_to\_acc;

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END;

/

UPDATE Accounts SET Balance = 1000 WHERE AccountID = 1; -- Source

UPDATE Accounts SET Balance = 500 WHERE AccountID = 2; -- Destination

COMMIT;

**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.**

CREATE OR REPLACE FUNCTION CalculateAge (

p\_dob IN DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

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

RETURN v\_age;

END;

/

SET SERVEROUTPUT ON;

DECLARE

v\_age NUMBER;

BEGIN

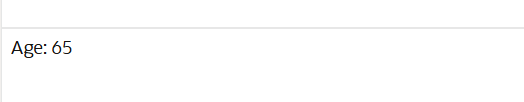
v\_age := CalculateAge(TO\_DATE('1960-06-15', 'YYYY-MM-DD'));

DBMS\_OUTPUT.PUT\_LINE('Age: ' || v\_age);

END;

/

**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.**

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;

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

IF v\_monthly\_rate = 0 THEN

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

ELSE

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

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

END IF;

RETURN ROUND(v\_emi, 2);

END;

/

DECLARE

v\_emi NUMBER;

BEGIN

v\_emi := CalculateMonthlyInstallment(10000, 12, 2); -- 2 years, 12% annual

DBMS\_OUTPUT.PUT\_LINE('Monthly Installment: $' || v\_emi);

END;

/

**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.**

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_account\_id IN NUMBER,

p\_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\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END;

/

DECLARE

v\_ok BOOLEAN;

BEGIN

v\_ok := HasSufficientBalance(1, 200); -- Assuming Account 1 exists

IF v\_ok THEN

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

ELSE

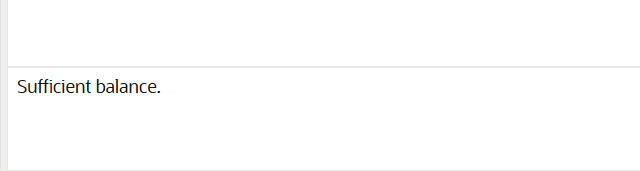
DBMS\_OUTPUT.PUT\_LINE('Insufficient balance.');

END IF;

END;

/

**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.**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

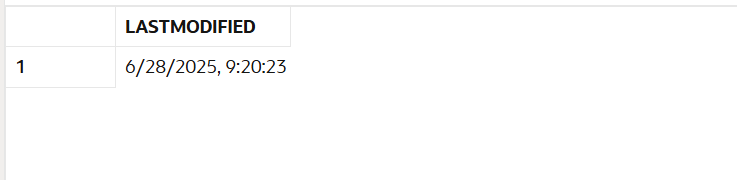
/

UPDATE Customers

SET Name = 'John D. Doe'

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.**

CREATE TABLE AuditLog (

LogID NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

AccountID NUMBER,

Amount NUMBER,

TransactionType VARCHAR2(10),

LogDate DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, AccountID, Amount, TransactionType, LogDate)

VALUES (:NEW.TransactionID, :NEW.AccountID, :NEW.Amount, :NEW.TransactionType, SYSDATE);

END;

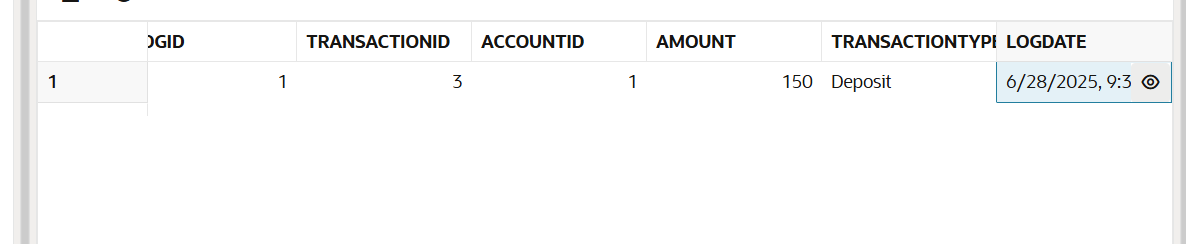
/

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

VALUES (3, 1, SYSDATE, 150, 'Deposit');

SELECT \* FROM AuditLog WHERE TransactionID = 3;

**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.

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 available balance.');

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

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

END IF;

END;

/

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

VALUES (4, 1, SYSDATE, 100, 'Deposit');

BEGIN

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

VALUES (5, 1, SYSDATE, -50, 'Deposit');

END;

/

BEGIN

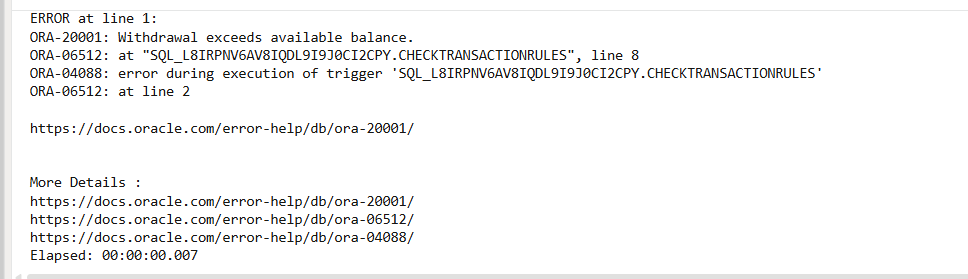
INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (6, 1, SYSDATE, 9999, 'Withdrawal');

END;

/

**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.**

SET SERVEROUTPUT ON;

DECLARE

CURSOR txn\_cursor IS

SELECT c.Name, a.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 EXTRACT(MONTH FROM t.TransactionDate) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM t.TransactionDate) = EXTRACT(YEAR FROM SYSDATE)

ORDER BY c.CustomerID, t.TransactionDate;

v\_record txn\_cursor%ROWTYPE;

BEGIN

OPEN txn\_cursor;

LOOP

FETCH txn\_cursor INTO v\_record;

EXIT WHEN txn\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(

'Customer: ' || v\_record.Name ||

' | Account: ' || v\_record.AccountID ||

' | Type: ' || v\_record.TransactionType ||

' | Amount: ' || v\_record.Amount ||

' | Date: ' || TO\_CHAR(v\_record.TransactionDate, 'YYYY-MM-DD')

);

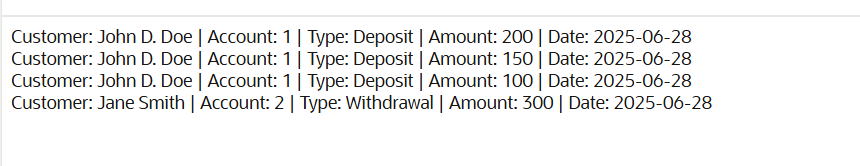
END LOOP;

CLOSE txn\_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.**

SET SERVEROUTPUT ON;

DECLARE

CURSOR acc\_cursor IS

SELECT AccountID, Balance

FROM Accounts;

v\_acc acc\_cursor%ROWTYPE;

v\_fee CONSTANT NUMBER := 50; -- Example flat fee

BEGIN

OPEN acc\_cursor;

LOOP

FETCH acc\_cursor INTO v\_acc;

EXIT WHEN acc\_cursor%NOTFOUND;

IF v\_acc.Balance >= v\_fee THEN

UPDATE Accounts

SET Balance = Balance - v\_fee,

LastModified = SYSDATE

WHERE AccountID = v\_acc.AccountID;

DBMS\_OUTPUT.PUT\_LINE(

'Annual fee applied to Account ID: ' || v\_acc.AccountID

);

ELSE

DBMS\_OUTPUT.PUT\_LINE(

'Skipped Account ID: ' || v\_acc.AccountID || ' due to low balance.'

);

END IF;

END LOOP;

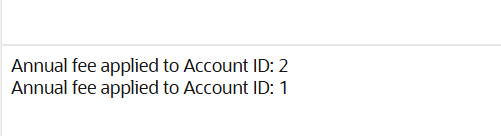
CLOSE acc\_cursor;

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.

SET SERVEROUTPUT ON;

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate, LoanAmount

FROM Loans;

v\_loan loan\_cursor%ROWTYPE;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_loan;

EXIT WHEN loan\_cursor%NOTFOUND;

IF v\_loan.LoanAmount > 10000 THEN

UPDATE Loans

SET InterestRate = InterestRate - 0.5

WHERE LoanID = v\_loan.LoanID;

DBMS\_OUTPUT.PUT\_LINE(

'Interest reduced for Loan ID: ' || v\_loan.LoanID

);

ELSE

DBMS\_OUTPUT.PUT\_LINE(

'No change for Loan ID: ' || v\_loan.LoanID

);

END IF;

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.**

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, p\_balance NUMBER);

FUNCTION GetCustomerBalance(p\_id NUMBER) RETURN NUMBER;

END CustomerManagement;

/

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);

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

ROLLBACK;

END;

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

BEGIN

UPDATE Customers

SET Name = p\_name, Balance = p\_balance, LastModified = SYSDATE

WHERE CustomerID = p\_id;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Customer not found.');

ELSE

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

END IF;

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;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN -1;

END;

END CustomerManagement;

/

BEGIN

CustomerManagement.AddCustomer(20, 'Charlie King', TO\_DATE('1990-02-10', 'YYYY-MM-DD'), 2500);

END;

/

BEGIN

CustomerManagement.UpdateCustomer(20, 'Charles King', 3000);

END;

/

DECLARE

v\_bal NUMBER;

BEGIN

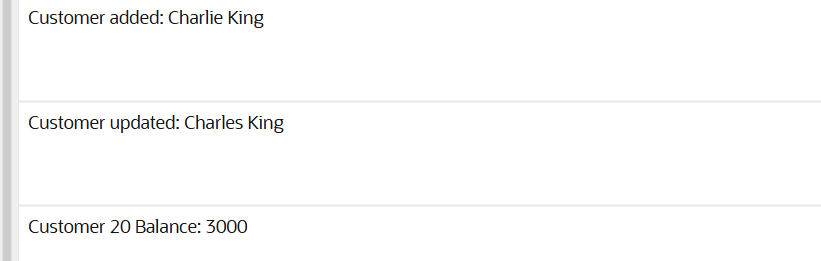
v\_bal := CustomerManagement.GetCustomerBalance(20);

DBMS\_OUTPUT.PUT\_LINE('Customer 20 Balance: ' || v\_bal);

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.

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee(p\_id NUMBER, p\_name VARCHAR2, p\_position VARCHAR2,

p\_salary NUMBER, p\_department VARCHAR2, p\_hire\_date DATE);

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

FUNCTION GetAnnualSalary(p\_id NUMBER) RETURN NUMBER;

END EmployeeManagement;

/

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\_hire\_date DATE) IS

BEGIN

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

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

ROLLBACK;

END;

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

BEGIN

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

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Employee not found.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Employee salary updated.');

END IF;

COMMIT;

END;

FUNCTION GetAnnualSalary(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 -1;

END;

END EmployeeManagement;

/

BEGIN

EmployeeManagement.HireEmployee(10, 'Diana Prince', 'Analyst', 5000, 'Finance', SYSDATE);

END;

/

BEGIN

EmployeeManagement.UpdateEmployee(10, 5500);

END;

/

DECLARE

v\_annual NUMBER;

BEGIN

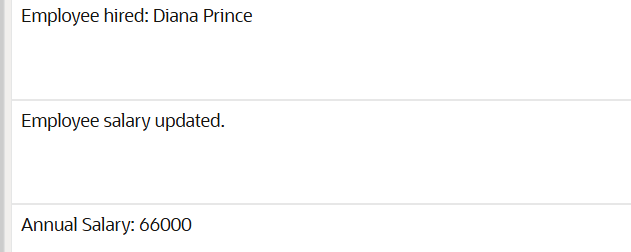
v\_annual := EmployeeManagement.GetAnnualSalary(10);

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

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.

CREATE OR REPLACE PACKAGE AccountOperations AS

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

PROCEDURE CloseAccount(p\_account\_id NUMBER);

FUNCTION GetTotalCustomerBalance(p\_cust\_id NUMBER) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

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

BEGIN

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

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

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

ROLLBACK;

END;

PROCEDURE CloseAccount(p\_account\_id NUMBER) IS

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_account\_id;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Account not found.');

ELSE

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

END IF;

COMMIT;

END;

FUNCTION GetTotalCustomerBalance(p\_cust\_id NUMBER) RETURN NUMBER IS

v\_total NUMBER;

BEGIN

SELECT NVL(SUM(Balance), 0) INTO v\_total

FROM Accounts WHERE CustomerID = p\_cust\_id;

RETURN v\_total;

END;

END AccountOperations;

/

BEGIN

AccountOperations.OpenAccount(10, 20, 'Savings', 800);

END;

/

DECLARE

v\_total NUMBER;

BEGIN

v\_total := AccountOperations.GetTotalCustomerBalance(20);

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

END;

/

BEGIN

AccountOperations.CloseAccount(10);

END;

/

**OUTPUT:**

