**Cognizant Deep Nurture 4.0 Deep Skilling - 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.

**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.

**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.

**Scenario 1**

**Apply 1% discount to loan interest rate for customers above 60 years old**

DECLARE

CURSOR cust\_cursor IS

SELECT customer\_id, age, loan\_interest\_rate

FROM customers;

BEGIN

FOR cust IN cust\_cursor LOOP

IF cust.age > 60 THEN

UPDATE customers

SET loan\_interest\_rate = loan\_interest\_rate - 0.01

WHERE customer\_id = cust.customer\_id;

END IF;

END LOOP;

COMMIT;

END;

/

**Scenario 2**

**Set IsVIP flag TRUE for customers with balance > $10,000**

DECLARE

CURSOR cust\_cursor IS

SELECT customer\_id, balance

FROM customers;

BEGIN

FOR cust IN cust\_cursor LOOP

IF cust.balance > 10000 THEN

UPDATE customers

SET IsVIP = 'TRUE'

WHERE customer\_id = cust.customer\_id;

END IF;

END LOOP;

COMMIT;

END;

/

**Scenario 3**

**Send reminders for loans due in the next 30 days**

DECLARE

CURSOR loan\_cursor IS

SELECT loan\_id, customer\_id, due\_date

FROM loans

WHERE due\_date BETWEEN SYSDATE AND SYSDATE + 30;

BEGIN

FOR loan IN loan\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan #' || loan.loan\_id ||

' for customer ' || loan.customer\_id ||

' is due on ' || TO\_CHAR(loan.due\_date, '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.

**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.

**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.

   
**Scenario 1**

**SafeTransferFunds**: Transfer funds between accounts, rollback on errors, log messages

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

insufficient\_funds EXCEPTION;

PRAGMA EXCEPTION\_INIT(insufficient\_funds, -20001);

v\_from\_balance NUMBER;

BEGIN

-- Check balance

SELECT balance INTO v\_from\_balance FROM accounts WHERE account\_id = p\_from\_account;

IF v\_from\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

-- Perform transfer

UPDATE accounts SET balance = balance - p\_amount WHERE account\_id = p\_from\_account;

UPDATE accounts SET balance = balance + p\_amount WHERE account\_id = p\_to\_account;

COMMIT;

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

EXCEPTION

WHEN insufficient\_funds THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in account ' || p\_from\_account);

WHEN NO\_DATA\_FOUND THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

END SafeTransferFunds;

/

**Scenario 2**

**UpdateSalary**: Increase salary by percentage, handle non-existent employee

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_emp\_id IN NUMBER,

p\_increment IN NUMBER

) AS

v\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO v\_count FROM employees WHERE employee\_id = p\_emp\_id;

IF v\_count = 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Employee ID ' || p\_emp\_id || ' does not exist.');

END IF;

UPDATE employees

SET salary = salary + (salary \* p\_increment / 100)

WHERE employee\_id = p\_emp\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salary updated for employee ' || p\_emp\_id);

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END UpdateSalary;

/

**Scenario 3**

**AddNewCustomer**: Insert customer, handle duplicate IDs by logging error

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_email IN VARCHAR2

) AS

BEGIN

INSERT INTO customers (customer\_id, name, email)

VALUES (p\_customer\_id, p\_name, p\_email);

COMMIT;

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

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

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

END AddNewCustomer;

/

**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.

**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.

**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.

**Scenario 1**

**ProcessMonthlyInterest** — Apply 1% interest to all savings accounts

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

UPDATE accounts

SET balance = balance + (balance \* 0.01)

WHERE account\_type = 'SAVINGS';

COMMIT;

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

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END ProcessMonthlyInterest;

/

**Scenario 2**

**UpdateEmployeeBonus** — Increase salaries by bonus % for employees in a department

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department\_id IN NUMBER,

p\_bonus\_percent IN NUMBER

) AS

BEGIN

UPDATE employees

SET salary = salary + (salary \* p\_bonus\_percent / 100)

WHERE department\_id = p\_department\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Bonus updated for department ' || p\_department\_id);

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

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

END UpdateEmployeeBonus;

/

**Scenario 3**

**TransferFunds** — Transfer amount between accounts with balance check

CREATE OR REPLACE PROCEDURE TransferFunds (

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 account\_id = p\_from\_account;

IF v\_balance < p\_amount THEN

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

END IF;

UPDATE accounts SET balance = balance - p\_amount WHERE account\_id = p\_from\_account;

UPDATE accounts SET balance = balance + p\_amount WHERE account\_id = p\_to\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount || ' from account ' || p\_from\_account ||

' to account ' || p\_to\_account || ' successful.');

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('One of the accounts not found.');

WHEN OTHERS THEN

ROLLBACK;

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

END TransferFunds;

/

**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.

**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.

**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.

**Scenario 1**

**CalculateAge** — Returns age in years from date of birth

CREATE OR REPLACE FUNCTION CalculateAge (

p\_dob IN DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

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

INTO v\_age

FROM dual;

RETURN v\_age;

END CalculateAge;

/

**Scenario 2**

**CalculateMonthlyInstallment** — Returns monthly installment based on loan details

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount IN NUMBER,

p\_interest\_rate IN NUMBER, -- annual interest rate in percent (e.g., 7.5)

p\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_rate NUMBER;

v\_total\_payments NUMBER;

v\_installment NUMBER;

BEGIN

v\_monthly\_rate := (p\_interest\_rate / 100) / 12;

v\_total\_payments := p\_years \* 12;

IF v\_monthly\_rate = 0 THEN

v\_installment := p\_loan\_amount / v\_total\_payments;

ELSE

v\_installment := p\_loan\_amount \* v\_monthly\_rate /

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

END IF;

RETURN ROUND(v\_installment, 2);

END CalculateMonthlyInstallment;

/

**Scenario 3**

**HasSufficientBalance** — Returns TRUE if account balance ≥ amount, else FALSE

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 account\_id = p\_account\_id;

RETURN (v\_balance >= p\_amount);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END HasSufficientBalance;

/

   
 