**Exercise 1: Control Structures**

**Scenario 1: The bank wants to apply a discount to loan interest rates for customers above 60 years old.**

**o 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.**

**o Question: Write a PL/SQL block that iterates through all customers and sets a flag Is VIP 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.**

**o 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**

BEGIN

FOR rec IN (SELECT customer\_id, age, loan\_interest\_rate FROM customers) LOOP

IF rec.age > 60 THEN

UPDATE customers

SET loan\_interest\_rate = loan\_interest\_rate \* 0.99

WHERE customer\_id = rec.customer\_id;

END IF;

END LOOP;

COMMIT;

END;

**Scenario 2**

BEGIN

FOR rec IN (SELECT customer\_id, balance FROM customers) LOOP

IF rec.balance > 10000 THEN

UPDATE customers

SET IsVIP = 'TRUE'

WHERE customer\_id = rec.customer\_id;

END IF;

END LOOP;

COMMIT;

END;

**Scenario 3**

BEGIN

FOR rec IN (

SELECT l.loan\_id, l.customer\_id, l.due\_date, c.customer\_name

FROM loans l

JOIN customers c ON l.customer\_id = c.customer\_id

WHERE l.due\_date BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || rec.loan\_id || ' for customer ' || rec.customer\_name || ' (ID: ' || rec.customer\_id || ') is due on ' || TO\_CHAR(rec.due\_date, 'DD-MON-YYYY'));

END LOOP;

END;

**Exercise 2: Error Handling**

**Scenario 1: Handle exceptions during fund transfers between accounts.**

**o 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.**

**o 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.**

**o 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**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

insufficient\_funds EXCEPTION;

PRAGMA EXCEPTION\_INIT(insufficient\_funds, -20900);

BEGIN

DECLARE

v\_balance NUMBER;

BEGIN

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

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

ELSE

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

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

COMMIT;

END IF;

EXCEPTION

WHEN insufficient\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Insufficient funds for transfer.');

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END;

END SafeTransferFunds;

**Scenario 2**

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) AS

employee\_not\_found EXCEPTION;

PRAGMA EXCEPTION\_INIT(employee\_not\_found, -20001);

BEGIN

DECLARE

v\_salary NUMBER;

BEGIN

SELECT salary INTO v\_salary FROM employees WHERE employee\_id = p\_employee\_id;

UPDATE employees SET salary = v\_salary \* (1 + p\_percentage / 100) WHERE employee\_id = p\_employee\_id;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE employee\_not\_found;

WHEN employee\_not\_found THEN

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

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END;

END UpdateSalary;

**Scenario 3**

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER

) AS

customer\_exists EXCEPTION;

PRAGMA EXCEPTION\_INIT(customer\_exists, -20002);

BEGIN

DECLARE

v\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO v\_count FROM customers WHERE customer\_id = p\_customer\_id;

IF v\_count > 0 THEN

RAISE customer\_exists;

ELSE

INSERT INTO customers (customer\_id, name, dob, balance, last\_modified)

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

COMMIT;

END IF;

EXCEPTION

WHEN customer\_exists THEN

DBMS\_OUTPUT.PUT\_LINE('Customer with the same ID already exists.');

ROLLBACK;

WHEN OTHERS THEN

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

ROLLBACK;

END;

END AddNewCustomer;

**Exercise 3: Stored Procedures**

**Scenario 1: The bank needs to process monthly interest for all savings accounts.**

**o 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.**

**o 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.**

**o 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**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

FOR rec IN (SELECT account\_id, balance FROM accounts WHERE account\_type = 'Savings') LOOP

UPDATE accounts

SET balance = balance \* 1.01

WHERE account\_id = rec.account\_id;

END LOOP;

COMMIT;

END ProcessMonthlyInterest;

**Scenario 2**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department IN VARCHAR2,

p\_bonus\_percentage IN NUMBER

) IS

BEGIN

UPDATE employees

SET salary = salary \* (1 + p\_bonus\_percentage / 100)

WHERE department = p\_department;

COMMIT;

END UpdateEmployeeBonus;

**Scenario 3**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account\_id IN NUMBER,

p\_to\_account\_id IN NUMBER,

p\_amount IN NUMBER

) AS

BEGIN

DECLARE

v\_balance NUMBER;

BEGIN

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

IF v\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance in source account.');

ELSE

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

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

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK;

END;

END TransferFunds;

**Exercise 4: Functions**

**Scenario 1: Calculate the age of customers for eligibility checks.**

**o 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.**

**o 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.**

**o 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**

CREATE OR REPLACE FUNCTION CalculateAge (

p\_dob IN DATE

) RETURN NUMBER IS

v\_age NUMBER;

BEGIN

v\_age := TRUNC((SYSDATE - p\_dob) / 365.25);

RETURN v\_age;

END CalculateAge;

**Scenario 2**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount IN NUMBER,

p\_interest\_rate IN NUMBER,

p\_duration\_years IN NUMBER

) RETURN NUMBER IS

v\_monthly\_installment NUMBER;

v\_rate\_per\_month NUMBER;

v\_number\_of\_payments NUMBER;

BEGIN

v\_rate\_per\_month := p\_interest\_rate / 12 / 100;

v\_number\_of\_payments := p\_duration\_years \* 12;

v\_monthly\_installment := p\_loan\_amount \* v\_rate\_per\_month / (1 - POWER(1 + v\_rate\_per\_month, -v\_number\_of\_payments));

RETURN v\_monthly\_installment;

END CalculateMonthlyInstallment;

**Scenario 3**

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;

END HasSufficientBalance;

**Exercise 5: Triggers**

**Scenario 1: Automatically update the last modified date when a customer's record is updated.**

**o 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.**

**Scenario 2: Maintain an audit log for all transactions.**

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

**Scenario 3: Enforce business rules on deposits and withdrawals.**

**o 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.**

**Scenario 1**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON customers

FOR EACH ROW

BEGIN

:NEW.last\_modified := SYSDATE;

END UpdateCustomerLastModified;

**Scenario 2**

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (transaction\_id, account\_id, transaction\_date, amount, transaction\_type)

VALUES (:NEW.transaction\_id, :NEW.account\_id, :NEW.transaction\_date, :NEW.amount, :NEW.transaction\_type);

END LogTransaction;

**Scenario 3**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON transactions

FOR EACH ROW

BEGIN

IF :NEW.transaction\_type = 'Withdrawal' THEN

DECLARE

v\_balance NUMBER;

BEGIN

SELECT balance INTO v\_balance

FROM accounts

WHERE account\_id = :NEW.account\_id;

IF :NEW.amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Withdrawal amount exceeds balance.');

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Account not found.');

END;

END IF;

IF :NEW.transaction\_type = 'Deposit' AND :NEW.amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Deposit amount must be positive.');

END IF;

END CheckTransactionRules;

**Exercise 6: Cursors**

**Scenario 1: Generate monthly statements for all customers.**

**o 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.**

**Scenario 2: Apply annual fee to all accounts.**

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

**Scenario 3: Update the interest rate for all loans based on a new policy.**

**o 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.**

**Scenario 1**

DECLARE

CURSOR cur\_transactions IS

SELECT t.transaction\_id, t.account\_id, t.transaction\_date, t.amount, t.transaction\_type, a.customer\_id

FROM transactions t

JOIN accounts a ON t.account\_id = a.account\_id

WHERE EXTRACT(MONTH FROM t.transaction\_date) = EXTRACT(MONTH FROM SYSDATE)

AND EXTRACT(YEAR FROM t.transaction\_date) = EXTRACT(YEAR FROM SYSDATE);

v\_customer\_id accounts.customer\_id%TYPE;

BEGIN

FOR rec IN cur\_transactions LOOP

DBMS\_OUTPUT.PUT\_LINE('Statement for Account ID ' || rec.account\_id || ' - Transaction ID: ' || rec.transaction\_id ||

', Date: ' || TO\_CHAR(rec.transaction\_date, 'DD-MON-YYYY') ||

', Amount: ' || rec.amount || ', Type: ' || rec.transaction\_type);

END LOOP;

END;

**Scenario 2**

DECLARE

CURSOR cur\_accounts IS

SELECT account\_id, balance

FROM accounts;

BEGIN

FOR rec IN cur\_accounts LOOP

UPDATE accounts

SET balance = balance - 50

WHERE account\_id = rec.account\_id;

END LOOP;

COMMIT;

END;

**Scenario 3**

DECLARE

CURSOR cur\_loans IS

SELECT loan\_id, interest\_rate

FROM loans;

BEGIN

FOR rec IN cur\_loans LOOP

UPDATE loans

SET interest\_rate = rec.interest\_rate + 0.5

WHERE loan\_id = rec.loan\_id;

END LOOP;

COMMIT;

END;

**Exercise 7: Packages**

**Scenario 1: Group all customer-related procedures and functions into a package.**

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

**Scenario 2: Create a package to manage employee data.**

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

**Scenario 3: Group all account-related operations into a package.**

**o 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.**

**Scenario 1**

CREATE OR REPLACE PACKAGE CustomerManagement AS

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

PROCEDURE UpdateCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_dob IN DATE);

FUNCTION GetCustomerBalance(p\_customer\_id IN NUMBER) RETURN NUMBER;

END CustomerManagement;

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

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

BEGIN

INSERT INTO customers (customer\_id, name, dob, balance, last\_modified)

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

END AddCustomer;

PROCEDURE UpdateCustomer(p\_customer\_id IN NUMBER, p\_name IN VARCHAR2, p\_dob IN DATE) IS

BEGIN

UPDATE customers

SET name = p\_name, dob = p\_dob, last\_modified = SYSDATE

WHERE customer\_id = p\_customer\_id;

END UpdateCustomer;

FUNCTION GetCustomerBalance(p\_customer\_id IN NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT balance INTO v\_balance

FROM customers

WHERE customer\_id = p\_customer\_id;

RETURN v\_balance;

END GetCustomerBalance;

END CustomerManagement;

**Scenario 2**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

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

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

FUNCTION CalculateAnnualSalary(p\_employee\_id IN NUMBER) RETURN NUMBER;

END EmployeeManagement;

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

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

BEGIN

INSERT INTO employees (employee\_id, name, position, salary, hire\_date)

VALUES (p\_employee\_id, p\_name, p\_position, p\_salary, SYSDATE);

END HireEmployee;

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

BEGIN

UPDATE employees

SET name = p\_name, position = p\_position

WHERE employee\_id = p\_employee\_id;

END UpdateEmployee;

FUNCTION CalculateAnnualSalary(p\_employee\_id IN NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

SELECT salary INTO v\_salary

FROM employees

WHERE employee\_id = p\_employee\_id;

RETURN v\_salary \* 12;

END CalculateAnnualSalary;

END EmployeeManagement;

**Scenario 3**

CREATE OR REPLACE PACKAGE AccountOperations AS

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

PROCEDURE CloseAccount(p\_account\_id IN NUMBER);

FUNCTION GetTotalBalance(p\_customer\_id IN NUMBER) RETURN NUMBER;

END AccountOperations;

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

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

BEGIN

INSERT INTO accounts (account\_id, customer\_id, account\_type, balance, last\_modified)

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

END OpenAccount;

PROCEDURE CloseAccount(p\_account\_id IN NUMBER) IS

BEGIN

DELETE FROM accounts

WHERE account\_id = p\_account\_id;

END CloseAccount;

FUNCTION GetTotalBalance(p\_customer\_id IN NUMBER) RETURN NUMBER IS

v\_total\_balance NUMBER;

BEGIN

SELECT SUM(balance) INTO v\_total\_balance

FROM accounts

WHERE customer\_id = p\_customer\_id;

RETURN v\_total\_balance;

END GetTotalBalance;

END AccountOperations;