**//SUBHANKAR PANI**

**//SUPERSET\_ID 5005259**

**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.

DECLARE

CURSOR customers\_cursor IS

SELECT CustomerID, DOB, Balance

FROM Customers;

customer customers\_cursor%ROWTYPE;

BEGIN

OPEN customers\_cursor;

LOOP

FETCH customers\_cursor INTO customer;

EXIT WHEN customers\_cursor%NOTFOUND;

IF TRUNC(MONTHS\_BETWEEN(SYSDATE, customer.DOB) / 12) > 60 THEN

UPDATE Customers

SET Balance = Balance \* 0.99

WHERE CustomerID = customer.CustomerID;

END IF;

END LOOP;

CLOSE customers\_cursor;

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.

DECLARE

CURSOR customers\_cursor IS

SELECT CustomerID, Balance

FROM Customers;

customer customers\_cursor%ROWTYPE;

BEGIN

OPEN customers\_cursor;

LOOP

FETCH customers\_cursor INTO customer;

EXIT WHEN customers\_cursor%NOTFOUND;

IF customer.Balance > 10000 THEN

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = customer.CustomerID;

END IF;

END LOOP;

CLOSE customers\_cursor;

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.

DECLARE

CURSOR loans\_cursor IS

SELECT LoanID, CustomerID, EndDate

FROM Loans

WHERE EndDate <= SYSDATE + 30;

loan loans\_cursor%ROWTYPE;

BEGIN

OPEN loans\_cursor;

LOOP

FETCH loans\_cursor INTO loan;

EXIT WHEN loans\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || loan.LoanID || ' for customer ' || loan.CustomerID || ' is due on ' || loan.EndDate);

END LOOP;

CLOSE loans\_cursor;

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.

CREATE OR REPLACE PROCEDURE SafeTransferFunds(

from\_account\_id NUMBER,

to\_account\_id NUMBER,

amount NUMBER

)

AS

BEGIN

IF amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Invalid transfer amount');

END IF;

IF from\_account\_id = to\_account\_id THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Cannot transfer funds to the same account');

END IF;

DECLARE

from\_balance NUMBER;

to\_balance NUMBER;

BEGIN

SELECT Balance INTO from\_balance

FROM Accounts

WHERE AccountID = from\_account\_id;

IF from\_balance < amount THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Insufficient funds in source account');

END IF;

SELECT Balance INTO to\_balance

FROM Accounts

WHERE AccountID = to\_account\_id;

UPDATE Accounts

SET Balance = from\_balance - amount

WHERE AccountID = from\_account\_id;

UPDATE Accounts

SET Balance = to\_balance + amount

WHERE AccountID = to\_account\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

RAISE\_APPLICATION\_ERROR(-20004, 'Error transferring funds: ' || SQLERRM);

END;

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.

CREATE OR REPLACE PROCEDURE UpdateSalary(

employee\_id NUMBER,

percentage NUMBER

)

AS

BEGIN

IF percentage <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20005, 'Invalid salary percentage');

END IF;

DECLARE

current\_salary NUMBER;

BEGIN

SELECT Salary INTO current\_salary

FROM Employees

WHERE EmployeeID = employee\_id;

IF current\_salary IS NULL THEN

RAISE\_APPLICATION\_ERROR(-20006, 'Employee not found');

END IF;

UPDATE Employees

SET Salary = current\_salary \* (1 + percentage / 100)

WHERE EmployeeID = employee\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

RAISE\_APPLICATION\_ERROR(-20007, 'Error updating salary: ' || SQLERRM);

END;

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.

CREATE OR REPLACE PROCEDURE AddNewCustomer(

customer\_id NUMBER,

name VARCHAR2,

dob DATE,

balance NUMBER

)

AS

BEGIN

IF customer\_id IS NULL OR name IS NULL OR dob IS NULL OR balance IS NULL THEN

RAISE\_APPLICATION\_ERROR(-20008, 'Invalid customer data');

END IF;

DECLARE

existing\_customer NUMBER;

BEGIN

SELECT CustomerID INTO existing\_customer

FROM Customers

WHERE CustomerID = customer\_id;

IF existing\_customer IS NOT NULL THEN

RAISE\_APPLICATION\_ERROR(-20009, 'Customer already exists');

END IF;

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

VALUES (customer\_id, name, dob, balance);

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

RAISE\_APPLICATION\_ERROR(-20010, 'Error adding customer: ' || SQLERRM);

END;

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.

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest

AS

BEGIN

UPDATE Accounts

SET Balance = Balance \* 1.01

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.

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

department VARCHAR2,

bonus\_percentage NUMBER

)

AS

BEGIN

UPDATE Employees

SET Salary = Salary \* (1 + bonus\_percentage / 100)

WHERE Department = department;

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.

CREATE OR REPLACE PROCEDURE TransferFunds(

from\_account\_id NUMBER,

to\_account\_id NUMBER,

amount NUMBER

)

AS

BEGIN

IF amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20011, 'Invalid transfer amount');

END IF;

IF from\_account\_id = to\_account\_id THEN

RAISE\_APPLICATION\_ERROR(-20012, 'Cannot transfer funds to the same account');

END IF;

DECLARE

from\_balance NUMBER;

to\_balance NUMBER;

BEGIN

SELECT Balance INTO from\_balance

FROM Accounts

WHERE AccountID = from\_account\_id;

IF from\_balance < amount THEN

RAISE\_APPLICATION\_ERROR(-20013, 'Insufficient funds in source account');

END IF;

SELECT Balance INTO to\_balance

FROM Accounts

WHERE AccountID = to\_account\_id;

UPDATE Accounts

SET Balance = from\_balance - amount

WHERE AccountID = from\_account\_id;

UPDATE Accounts

SET Balance = to\_balance + amount

WHERE AccountID = to\_account\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

RAISE\_APPLICATION\_ERROR(-20014, 'Error transferring funds: ' || SQLERRM);

END;

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.

CREATE OR REPLACE FUNCTION CalculateAge(

dob DATE

)

RETURN NUMBER

AS

BEGIN

RETURN TRUNC(MONTHS\_BETWEEN(SYSDATE, 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.

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(

loan\_amount NUMBER,

interest\_rate NUMBER,

loan\_duration NUMBER

)

RETURN NUMBER

AS

BEGIN

RETURN loan\_amount \* (interest\_rate / 100) / loan\_duration;

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.

CREATE OR REPLACE FUNCTION HasSufficientBalance(

account\_id NUMBER,

amount NUMBER

)

RETURN BOOLEAN

AS

BEGIN

DECLARE

balance NUMBER;

BEGIN

SELECT Balance INTO balance

FROM Accounts

WHERE AccountID = account\_id;

RETURN balance >= amount;

END;

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.

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

AFTER UPDATE ON Customers

FOR EACH ROW

BEGIN

UPDATE Customers

SET LastModified = SYSDATE

WHERE CustomerID = :NEW.CustomerID;

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.

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, TransactionDate, Amount, TransactionType)

VALUES (:NEW.TransactionID, :NEW.TransactionDate, :NEW.Amount, :NEW.TransactionType);

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.

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

BEGIN

IF :NEW.TransactionType = 'Withdrawal' THEN

DECLARE

balance NUMBER;

BEGIN

SELECT Balance INTO balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

IF balance < :NEW.Amount THEN

RAISE\_APPLICATION\_ERROR(-20015, 'Insufficient funds for withdrawal');

END IF;

END;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20016, 'Invalid deposit amount');

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.

DECLARE

CURSOR customers\_cursor IS

SELECT CustomerID, Name

FROM Customers;

customer customers\_cursor%ROWTYPE;

CURSOR transactions\_cursor IS

SELECT TransactionDate, Amount, TransactionType

FROM Transactions

WHERE AccountID IN (SELECT AccountID FROM Accounts WHERE CustomerID = :customer\_id)

AND TransactionDate >= TRUNC(SYSDATE, 'MM') - INTERVAL '1 month';

transaction transactions\_cursor%ROWTYPE;

BEGIN

OPEN customers\_cursor;

LOOP

FETCH customers\_cursor INTO customer;

EXIT WHEN customers\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Statement for customer ' || customer.Name);

OPEN transactions\_cursor;

LOOP

FETCH transactions\_cursor INTO transaction;

EXIT WHEN transactions\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(' ' || transaction.TransactionDate || ' - ' || transaction.TransactionType || ' - ' || transaction.Amount);

END LOOP;

CLOSE transactions\_cursor;

END LOOP;

CLOSE customers\_cursor;

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.

DECLARE

CURSOR accounts\_cursor IS

SELECT AccountID, Balance

FROM Accounts;

account accounts\_cursor%ROWTYPE;

BEGIN

OPEN accounts\_cursor;

LOOP

FETCH accounts\_cursor INTO account;

EXIT WHEN accounts\_cursor%NOTFOUND;

UPDATE Accounts

SET Balance = account.Balance - 50

WHERE AccountID = account.AccountID;

END LOOP;

CLOSE accounts\_cursor;

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.

DECLARE

CURSOR loans\_cursor IS

SELECT LoanID, InterestRate

FROM Loans;

loan loans\_cursor%ROWTYPE;

BEGIN

OPEN loans\_cursor;

LOOP

FETCH loans\_cursor INTO loan;

EXIT WHEN loans\_cursor%NOTFOUND;

UPDATE Loans

SET InterestRate = loan.InterestRate \* 1.05

WHERE LoanID = loan.LoanID;

END LOOP;

CLOSE loans\_cursor;

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.

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddNewCustomer(

customer\_id NUMBER,

name VARCHAR2,

dob DATE,

balance NUMBER

);

PROCEDURE UpdateCustomerDetails(

customer\_id NUMBER,

name VARCHAR2,

dob DATE

);

FUNCTION GetCustomerBalance(

customer\_id NUMBER

) RETURN NUMBER;

END CustomerManagement;

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddNewCustomer(

customer\_id NUMBER,

name VARCHAR2,

dob DATE,

balance NUMBER

)

AS

BEGIN

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

VALUES (customer\_id, name, dob, balance);

END;

PROCEDURE UpdateCustomerDetails(

customer\_id NUMBER,

name VARCHAR2,

dob DATE

)

AS

BEGIN

UPDATE Customers

SET Name = name, DOB = dob

WHERE CustomerID = customer\_id;

END;

FUNCTION GetCustomerBalance(

customer\_id NUMBER

) RETURN NUMBER

AS

balance NUMBER;

BEGIN

SELECT Balance INTO balance

FROM Customers

WHERE CustomerID = customer\_id;

RETURN 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.

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireNewEmployee(

employee\_id NUMBER,

name VARCHAR2,

position VARCHAR2,

salary NUMBER,

department VARCHAR2,

hire\_date DATE

);

PROCEDURE UpdateEmployeeDetails(

employee\_id NUMBER,

name VARCHAR2,

position VARCHAR2,

salary NUMBER,

department VARCHAR2

);

FUNCTION CalculateAnnualSalary(

employee\_id NUMBER

) RETURN NUMBER;

END EmployeeManagement;

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireNewEmployee(

employee\_id NUMBER,

name VARCHAR2,

position VARCHAR2,

salary NUMBER,

department VARCHAR2,

hire\_date DATE

)

AS

BEGIN

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

VALUES (employee\_id, name, position, salary, department, hire\_date);

END;

PROCEDURE UpdateEmployeeDetails(

employee\_id NUMBER,

name VARCHAR2,

position VARCHAR2,

salary NUMBER,

department VARCHAR2

)

AS

BEGIN

UPDATE Employees

SET Name = name, Position = position, Salary = salary, Department = department

WHERE EmployeeID = employee\_id;

END;

FUNCTION CalculateAnnualSalary(

employee\_id NUMBER

) RETURN NUMBER

AS

salary NUMBER;

BEGIN

SELECT Salary INTO salary

FROM Employees

WHERE EmployeeID = employee\_id;

RETURN 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.

CREATE OR REPLACE PACKAGE AccountManagement AS

PROCEDURE OpenNewAccount(

account\_id NUMBER,

customer\_id NUMBER,

account\_type VARCHAR2,

balance NUMBER

);

PROCEDURE UpdateAccountDetails(

account\_id NUMBER,

account\_type VARCHAR2,

balance NUMBER

);

FUNCTION GetAccountBalance(

account\_id NUMBER

) RETURN NUMBER;

END AccountManagement;

CREATE OR REPLACE PACKAGE BODY AccountManagement AS

PROCEDURE OpenNewAccount(

account\_id NUMBER,

customer\_id NUMBER,

account\_type VARCHAR2,

balance NUMBER

)

AS

BEGIN

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

VALUES (account\_id, customer\_id, account\_type, balance);

END;

PROCEDURE UpdateAccountDetails(

account\_id NUMBER,

account\_type VARCHAR2,

balance NUMBER

)

AS

BEGIN

UPDATE Accounts

SET AccountType = account\_type, Balance = balance

WHERE AccountID = account\_id;

END;

FUNCTION GetAccountBalance(

account\_id NUMBER

) RETURN NUMBER

AS

balance NUMBER;

BEGIN

SELECT Balance INTO balance

FROM Accounts

WHERE AccountID = account\_id;

RETURN balance;

END;

END AccountManagement;