**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.

**Ans)** DO $$

DECLARE

rec RECORD;

BEGIN

FOR rec IN

SELECT l.LoanID

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE AGE(c.DOB) > INTERVAL '60 years'

LOOP

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE LoanID = rec.LoanID;

END LOOP;

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.

**Ans)** DO $$

DECLARE

rec RECORD;

BEGIN

FOR rec IN

SELECT CustomerID, Name, Balance FROM Customers

WHERE Balance > 10000

LOOP

RAISE NOTICE 'Customer % (%), with balance $%, qualifies for VIP status.',

rec.CustomerID, rec.Name, rec.Balance;

END LOOP;

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.

**Ans)** DO $$

DECLARE

rec RECORD;

BEGIN

FOR rec IN

SELECT \* FROM Loans

WHERE EndDate <= CURRENT\_DATE + INTERVAL '30 days'

LOOP

RAISE NOTICE 'Reminder: Loan % is due for Customer %', rec.LoanID, rec.CustomerID;

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.

**Ans)** CREATE OR REPLACE FUNCTION SafeTransferFunds(p\_from INT, p\_to INT, p\_amount NUMERIC) RETURNS VOID AS $$

BEGIN

BEGIN

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

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

COMMIT;

EXCEPTION WHEN OTHERS THEN

ROLLBACK;

RAISE NOTICE 'Transfer failed: %', SQLERRM;

END;

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE OR REPLACE FUNCTION UpdateSalary(p\_empid INT, p\_percent NUMERIC) RETURNS VOID AS $$

BEGIN

UPDATE Employees

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

WHERE EmployeeID = p\_empid;

IF NOT FOUND THEN

RAISE NOTICE 'Employee not found';

END IF;

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE OR REPLACE FUNCTION AddNewCustomer(p\_id INT, p\_name TEXT, p\_dob DATE, p\_balance NUMERIC) RETURNS VOID AS $$

BEGIN

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

VALUES (p\_id, p\_name, p\_dob, p\_balance, CURRENT\_TIMESTAMP);

EXCEPTION

WHEN unique\_violation THEN

RAISE NOTICE 'Customer ID already exists';

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE OR REPLACE FUNCTION ProcessMonthlyInterest() RETURNS VOID AS $$

BEGIN

UPDATE Accounts

SET Balance = Balance + Balance \* 0.01

WHERE AccountType = 'Savings';

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE OR REPLACE FUNCTION UpdateEmployeeBonus(p\_dept TEXT, p\_bonus NUMERIC) RETURNS VOID AS $$

BEGIN

UPDATE Employees

SET Salary = Salary + Salary \* p\_bonus / 100

WHERE Department = p\_dept;

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE OR REPLACE FUNCTION TransferFunds(p\_from INT, p\_to INT, p\_amt NUMERIC) RETURNS VOID AS $$

DECLARE

bal NUMERIC;

BEGIN

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

IF bal >= p\_amt THEN

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

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

ELSE

RAISE NOTICE 'Insufficient balance';

END IF;

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE OR REPLACE FUNCTION CalculateAge(p\_dob DATE) RETURNS INT AS $$

BEGIN

RETURN DATE\_PART('year', AGE(CURRENT\_DATE, p\_dob));

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(p\_amt NUMERIC, p\_rate NUMERIC, p\_years INT)

RETURNS NUMERIC AS $$

DECLARE

r NUMERIC := p\_rate / (12 \* 100);

n INT := p\_years \* 12;

BEGIN

RETURN (p\_amt \* r \* POWER(1 + r, n)) / (POWER(1 + r, n) - 1);

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE OR REPLACE FUNCTION HasSufficientBalance(p\_accid INT, p\_amt NUMERIC) RETURNS BOOLEAN AS $$

DECLARE

bal NUMERIC;

BEGIN

SELECT Balance INTO bal FROM Accounts WHERE AccountID = p\_accid;

RETURN bal >= p\_amt;

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE OR REPLACE FUNCTION UpdateLastModifiedFunc() RETURNS TRIGGER AS $$

BEGIN

NEW.LastModified := CURRENT\_TIMESTAMP;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

EXECUTE FUNCTION UpdateLastModifiedFunc();

**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.

**Ans)** CREATE OR REPLACE FUNCTION LogTransactionFunc() RETURNS TRIGGER AS $$

BEGIN

INSERT INTO AuditLog(TransactionID, LogDate)

VALUES (NEW.TransactionID, CURRENT\_TIMESTAMP);

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

EXECUTE FUNCTION LogTransactionFunc();

**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.

**Ans)** CREATE OR REPLACE FUNCTION CheckTransactionRulesFunc() RETURNS TRIGGER AS $$

DECLARE

bal NUMERIC;

BEGIN

SELECT Balance INTO bal FROM Accounts WHERE AccountID = NEW.AccountID;

IF NEW.TransactionType = 'Withdrawal' AND NEW.Amount > bal THEN

RAISE EXCEPTION 'Withdrawal exceeds balance';

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

RAISE EXCEPTION 'Deposit must be positive';

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

EXECUTE FUNCTION CheckTransactionRulesFunc();

**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.

**Ans)** DO $$

DECLARE

rec RECORD;

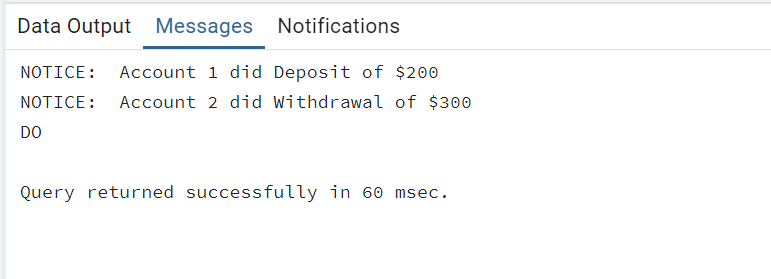
BEGIN

FOR rec IN SELECT \* FROM Transactions WHERE DATE\_PART('month', TransactionDate) = DATE\_PART('month', CURRENT\_DATE) LOOP

RAISE NOTICE 'Account % did % of $%', rec.AccountID, rec.TransactionType, 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.

**Ans)** DO $$

DECLARE

rec RECORD;

BEGIN

FOR rec IN SELECT \* FROM Accounts LOOP

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

END LOOP;

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.

**Ans)** DO $$

DECLARE

rec RECORD;

BEGIN

FOR rec IN SELECT \* FROM Loans LOOP

UPDATE Loans SET InterestRate = InterestRate \* 0.95 WHERE LoanID = rec.LoanID;

END LOOP;

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.

**Ans)** CREATE SCHEMA IF NOT EXISTS customer\_management;

CREATE OR REPLACE FUNCTION customer\_management.add\_customer(

p\_id INT,

p\_name TEXT,

p\_dob DATE,

p\_balance NUMERIC

) RETURNS VOID AS $$

BEGIN

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

VALUES (p\_id, p\_name, p\_dob, p\_balance, CURRENT\_TIMESTAMP);

EXCEPTION

WHEN unique\_violation THEN

RAISE NOTICE 'Customer already exists';

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION customer\_management.update\_customer\_name(

p\_id INT,

p\_new\_name TEXT

) RETURNS VOID AS $$

BEGIN

UPDATE Customers SET Name = p\_new\_name, LastModified = CURRENT\_TIMESTAMP

WHERE CustomerID = p\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION customer\_management.get\_balance(p\_id INT)

RETURNS NUMERIC AS $$

DECLARE

bal NUMERIC;

BEGIN

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

RETURN bal;

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE SCHEMA IF NOT EXISTS employee\_management;

CREATE OR REPLACE FUNCTION employee\_management.hire\_employee(

p\_id INT,

p\_name TEXT,

p\_position TEXT,

p\_salary NUMERIC,

p\_department TEXT,

p\_hire\_date DATE

) RETURNS VOID AS $$

BEGIN

INSERT INTO Employees VALUES (

p\_id, p\_name, p\_position, p\_salary, p\_department, p\_hire\_date

);

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION employee\_management.update\_employee\_salary(

p\_id INT,

p\_new\_salary NUMERIC

) RETURNS VOID AS $$

BEGIN

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

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION employee\_management.calculate\_annual\_salary(

p\_id INT

) RETURNS NUMERIC AS $$

DECLARE

sal NUMERIC;

BEGIN

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

RETURN sal \* 12;

END;

$$ LANGUAGE plpgsql;

**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.

**Ans)** CREATE SCHEMA IF NOT EXISTS account\_operations;

CREATE OR REPLACE FUNCTION account\_operations.open\_account(

p\_accid INT,

p\_custid INT,

p\_type TEXT,

p\_balance NUMERIC

) RETURNS VOID AS $$

BEGIN

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

VALUES (p\_accid, p\_custid, p\_type, p\_balance, CURRENT\_TIMESTAMP);

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION account\_operations.close\_account(

p\_accid INT

) RETURNS VOID AS $$

BEGIN

DELETE FROM Accounts WHERE AccountID = p\_accid;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION account\_operations.total\_balance(p\_custid INT)

RETURNS NUMERIC AS $$

DECLARE

total NUMERIC;

BEGIN

SELECT SUM(Balance) INTO total FROM Accounts WHERE CustomerID = p\_custid;

RETURN COALESCE(total, 0);

END;

$$ LANGUAGE plpgsql;

**Schema to be Created**

*CREATE TABLE Customers (*

*CustomerID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*DOB DATE,*

*Balance NUMBER,*

*LastModified DATE*

*);*

*CREATE TABLE Accounts (*

*AccountID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*AccountType VARCHAR2(20),*

*Balance NUMBER,*

*LastModified DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Transactions (*

*TransactionID NUMBER PRIMARY KEY,*

*AccountID NUMBER,*

*TransactionDate DATE,*

*Amount NUMBER,*

*TransactionType VARCHAR2(10),*

*FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)*

*);*

*CREATE TABLE Loans (*

*LoanID NUMBER PRIMARY KEY,*

*CustomerID NUMBER,*

*LoanAmount NUMBER,*

*InterestRate NUMBER,*

*StartDate DATE,*

*EndDate DATE,*

*FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)*

*);*

*CREATE TABLE Employees (*

*EmployeeID NUMBER PRIMARY KEY,*

*Name VARCHAR2(100),*

*Position VARCHAR2(50),*

*Salary NUMBER,*

*Department VARCHAR2(50),*

*HireDate DATE*

*);*

**Example Scripts for Sample Data Insertion**

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

*VALUES (1, 'John Doe', TO\_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);*

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

*VALUES (2, 'Jane Smith', TO\_DATE('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);*

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

*VALUES (1, 1, 'Savings', 1000, SYSDATE);*

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

*VALUES (2, 2, 'Checking', 1500, SYSDATE);*

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

*VALUES (1, 1, SYSDATE, 200, 'Deposit');*

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

*VALUES (2, 2, SYSDATE, 300, 'Withdrawal');*

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

*VALUES (1, 1, 5000, 5, SYSDATE, ADD\_MONTHS(SYSDATE, 60));*

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

*VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO\_DATE('2015-06-15', 'YYYY-MM-DD'));*

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

*VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO\_DATE('2017-03-20', 'YYYY-MM-DD'));*