**Week-II**

**PL SQL Exercises**

**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.

**Code**:

declare

v\_customer\_age NUMBER;

v\_current\_interest\_rate NUMBER;

begin

for rec in (select CustomerID, DOB from Customers) loop

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

if v\_customer\_age > 60 THEN

select InterestRate into v\_current\_interest\_rate from Loans where

CustomerID = rec.CustomerID;

update Loans set InterestRate = v\_current\_interest\_rate - 1

where CustomerID = rec.CustomerID;

end if;

end loop;

commit;

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.

**Code:**

Step(1): add IsVIP field into the table

alter table Customers ADD (IsVIP char(1) default 'N');

Step(2): setting IsVIP to true

begin

for rec in (select CustomerID, Balance from Customers) loop

if rec.Balance > 10000 then

update Customers

set IsVIP = 'Y'

where CustomerID = rec.CustomerID;

end if;

end loop;

commit;

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.

**Code:**

declare

cursor loan\_cursor IS

SELECT l.CustomerID, l.LoanID, c.Name, l.EndDate

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30;

v\_customer\_name Customers.Name%TYPE;

v\_loan\_id Loans.LoanID%TYPE;

v\_end\_date Loans.EndDate%TYPE;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_customer\_name, v\_loan\_id, v\_end\_date;

EXIT WHEN loan\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || v\_loan\_id || ' for customer ' || v\_customer\_name || ' is due on ' || v\_end\_date);

END LOOP;

CLOSE loan\_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.

**Code:**

create or replace procedure SafeTransferFunds(

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

BEGIN

BEGIN

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Source account not found');

END IF;

IF (SELECT Balance FROM Accounts WHERE AccountID = p\_from\_account) < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Insufficient funds');

END IF;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Destination account not found');

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE(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.

Code:

CREATE OR REPLACE PROCEDURE UpdateSalary(

p\_employee\_id IN NUMBER,

p\_percentage IN NUMBER

) AS

BEGIN

UPDATE Employees

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

WHERE EmployeeID = p\_employee\_id;

IF SQL%ROWCOUNT = 0 THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Employee ID does not exist');

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE(SQLERRM);

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.

**Code:**

CREATE OR REPLACE PROCEDURE AddNewCustomer(

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

p\_dob IN DATE,

p\_balance IN NUMBER,

p\_last\_modified IN DATE

) AS

BEGIN

BEGIN

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

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

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

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

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE(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.

**Code:**

create or replace procedure processmonthlyinterest(

p\_interest\_rate in number

) as

begin

update ACCOUNTS

set balance = balance \* (1 + p\_interest\_rate / 100)

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.

Code:

create or replace procedure updateemployeebonus(

p\_department in varchar2,

p\_bonus\_percentage in number

) as

begin

update EMPLOYEES

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

where department = p\_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.

**Code:**

create or replace procedure transferfunds(

p\_from\_account in number,

p\_to\_account in number,

p\_amount in number

) as

begin

declare

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');

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;

exception

when others then

rollback;

dbms\_output.put\_line(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.

**Code:**

create or replace function calculateage(p\_dob in date) return number as

v\_age number;

begin v\_age := floor(months\_between(sysdate, p\_dob) / 12);

return v\_age;

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.

**Code:**

create or replace function calculatemonthlyinstallment(

p\_loan\_amount in number,

p\_interest\_rate in number,

p\_duration\_years in number

) return number as

v\_monthly\_installment number;

v\_monthly\_rate number;

v\_total\_months number;

begin

v\_monthly\_rate := p\_interest\_rate / 1200;

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

v\_monthly\_installment := p\_loan\_amount \* (v\_monthly\_rate / (1 - power(1 + v\_monthly\_rate, -v\_total\_months)));

return v\_monthly\_installment;

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.

**Code:**

create or replace function hassufficientbalance(

p\_account\_id in number,

p\_amount in number

) return boolean as

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;

/

**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.

**Code:**

create or replace trigger updatecustomerlastmodified

before update on CUSTOMERS

for each row

begin

:new.lastmodified := sysdate;

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.

**Code:**

create or replace trigger logtransaction

after insert on TRANSACTIONS

for each row

begin

insert into AUDITLOG (transactionid, action, actiondate)

values (:new.transactionid, 'INSERT', sysdate);

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.

**Code:**

create or replace trigger checktransactionrules

before insert on TRANSACTIONS

for each row

declare

v\_balance number;

begin

if :new.transactiontype = 'Withdrawal' then

select balance into v\_balance

from ACCOUNTS

where accountid = :new.accountid;

if v\_balance < :new.amount then

raise\_application\_error(-20002, 'Insufficient funds for withdrawal');

end if;

elsif :new.transactiontype = 'Deposit' then

if :new.amount <= 0 then

raise\_application\_error(-20003, 'Deposit amount must be positive');

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.

Code:

declare

cursor generate\_monthly\_statements is

select c.customerid, c.name, t.transactionid, t.transactiondate, t.amount, t.transactiontype

from CUSTOMERS c

join ACCOUNTS a on c.customerid = a.customerid

join TRANSACTIONS t on a.accountid = t.accountid

where t.transactiondate between trunc(sysdate, 'MM') and last\_day(sysdate);

v\_customerid CUSTOMERS.customerid%type;

v\_name CUSTOMERS.name%type;

v\_transactionid TRANSACTIONS.transactionid%type;

v\_transactiondate TRANSACTIONS.transactiondate%type;

v\_amount TRANSACTIONS.amount%type;

v\_transactiontype TRANSACTIONS.transactiontype%type;

begin

open generate\_monthly\_statements;

loop

fetch generate\_monthly\_statements into v\_customerid, v\_name, v\_transactionid, v\_transactiondate, v\_amount, v\_transactiontype;

exit when generate\_monthly\_statements%notfound;

dbms\_output.put\_line('Customer: ' || v\_name || ' - Transaction ID: ' || v\_transactionid || ' - Date: ' || v\_transactiondate || ' - Amount: ' || v\_amount || ' - Type: ' || v\_transactiontype);

end loop;

close generate\_monthly\_statements;

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.

Code:

declare

cursor apply\_annual\_fee is

select accountid, balance

from ACCOUNTS;

v\_accountid ACCOUNTS.accountid%type;

v\_balance ACCOUNTS.balance%type;

v\_fee number := 50; -- Example annual fee

begin

open apply\_annual\_fee;

loop

fetch apply\_annual\_fee into v\_accountid, v\_balance;

exit when apply\_annual\_fee%notfound;

update ACCOUNTS

set balance = v\_balance - v\_fee

where accountid = v\_accountid;

end loop;

close apply\_annual\_fee;

commit;

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.

Code:

declare

cursor update\_loan\_interest\_rates is

select loanid, interestrate

from LOANS;

v\_loanid LOANS.loanid%type;

v\_interestrate LOANS.interestrate%type;

v\_new\_rate number := 4; -- Example new interest rate

begin

open update\_loan\_interest\_rates;

loop

fetch update\_loan\_interest\_rates into v\_loanid, v\_interestrate;

exit when update\_loan\_interest\_rates%notfound;

update LOANS

set interestrate = v\_new\_rate

where loanid = v\_loanid;

end loop;

close update\_loan\_interest\_rates;

commit;

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.

**Code:**

**Package specification:**

create or replace package customermanagement as

procedure addnewcustomer(

p\_customer\_id in number,

p\_name in varchar2,

p\_dob in date,

p\_balance in number,

p\_last\_modified in date

);

procedure updatecustomerdetails(

p\_customer\_id in number,

p\_name in varchar2,

p\_balance in number

);

function getcustomerbalance(

p\_customer\_id in number

) return number;

end;

/

**Package body:**

create or replace package body customermanagement as

procedure addnewcustomer(

p\_customer\_id in number,

p\_name in varchar2,

p\_dob in date,

p\_balance in number,

p\_last\_modified in date

) is

begin

insert into CUSTOMERS (CustomerID, Name, DOB, Balance, LastModified)

values (p\_customer\_id, p\_name, p\_dob, p\_balance, p\_last\_modified);

commit;

end;

procedure updatecustomerdetails(

p\_customer\_id in number,

p\_name in varchar2,

p\_balance in number

) is

begin

update CUSTOMERS

set Name = p\_name,

Balance = p\_balance

where CustomerID = p\_customer\_id;

commit;

end;

function getcustomerbalance(

p\_customer\_id in number

) return number is

v\_balance number;

begin

select Balance into v\_balance

from CUSTOMERS

where CustomerID = p\_customer\_id;

return v\_balance;

exception

when no\_data\_found then

return 0;

end;

end;

/

**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.

Code:

**Package specification:**

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,

p\_department in varchar2,

p\_hire\_date in date

);

procedure updateemployeedetails(

p\_employee\_id in number,

p\_name in varchar2,

p\_salary in number

);

function calculateannualsalary(

p\_employee\_id in number

) return number;

end;

/

**Package Body:**

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,

p\_department in varchar2,

p\_hire\_date in date

) is

begin

insert into EMPLOYEES (EmployeeID, Name, Position, Salary, Department, HireDate)

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

commit;

end;

procedure updateemployeedetails(

p\_employee\_id in number,

p\_name in varchar2,

p\_salary in number

) is

begin

update EMPLOYEES

set Name = p\_name, Salary = p\_salary where EmployeeID = p\_employee\_id;

commit;

end;

function calculateannualsalary(

p\_employee\_id in number

) return number is

v\_salary number;

begin

select Salary into v\_salary

from EMPLOYEES

where EmployeeID = p\_employee\_id;

return v\_salary \* 12;

exception

when no\_data\_found then

return 0;

end;

end;

/

**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.

**Code:**

**Package specifications:**

create or replace package accountoperations as

procedure opennewaccount(

p\_account\_id in number,

p\_customer\_id in number,

p\_account\_type in varchar2,

p\_balance in number,

p\_last\_modified in date

);

procedure closeaccount(

p\_account\_id in number

);

function gettotalbalance(

p\_customer\_id in number

) return number;

end;

/

**Package code:**

create or replace package body accountoperations as

procedure opennewaccount(

p\_account\_id in number,

p\_customer\_id in number,

p\_account\_type in varchar2,

p\_balance in number,

p\_last\_modified in date

) is

begin

insert into ACCOUNTS (AccountID, CustomerID, AccountType, Balance, LastModified)

values (p\_account\_id, p\_customer\_id, p\_account\_type, p\_balance, p\_last\_modified);

commit;

end;

procedure closeaccount(

p\_account\_id in number

) is

begin

delete from ACCOUNTS

where AccountID = p\_account\_id;

commit;

end;

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 CustomerID = p\_customer\_id;

return v\_total\_balance;

exception

when no\_data\_found then

return 0;

end;

end;

/