**Week-2: PL/SQL Exercise Solutions**

**Initial Setup:**

**Table Creation:**

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Balance INT,

LastModified DATE

);

CREATE TABLE Accounts (

AccountID INT PRIMARY KEY,

CustomerID INT,

AccountType VARCHAR(20),

Balance INT,

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Transactions (

TransactionID INT PRIMARY KEY,

AccountID INT,

TransactionDate DATE,

Amount INT,

TransactionType VARCHAR(10),

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

CREATE TABLE Loans (

LoanID INT PRIMARY KEY,

CustomerID INT,

LoanAmount INT,

InterestRate INT,

StartDate DATE,

EndDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(100),

Position VARCHAR(50),

Salary INT,

Department VARCHAR(50),

HireDate DATE

);

**Record Insertion:**

BEGIN

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (1, 'Ram Kumar', TO\_DATE('1980-01-15', 'YYYY-MM-DD'), 10000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (2, 'Sita Devi', TO\_DATE('1990-03-22', 'YYYY-MM-DD'), 15000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (3, 'Arun Vijay', TO\_DATE('1975-07-10', 'YYYY-MM-DD'), 20000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (4, 'Lakshmi Narayanan', TO\_DATE('1985-06-05', 'YYYY-MM-DD'), 18000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (5, 'Priya Rajesh', TO\_DATE('1992-08-14', 'YYYY-MM-DD'), 25000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified) VALUES (6, 'Vijay Anand', TO\_DATE('1988-12-20', 'YYYY-MM-DD'), 30000, SYSDATE);

END;

/

BEGIN

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (1, 1, 'Savings', 10000, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (2, 2, 'Checking', 15000, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (3, 3, 'Savings', 20000, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (4, 4, 'Checking', 18000, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (5, 5, 'Savings', 25000, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified) VALUES (6, 6, 'Checking', 30000, SYSDATE);

END;

/

BEGIN

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (1, 1, SYSDATE, 500, 'Credit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (2, 2, SYSDATE, 1000, 'Debit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (3, 3, SYSDATE, 1500, 'Credit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (4, 4, SYSDATE, 2000, 'Debit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (5, 5, SYSDATE, 2500, 'Credit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType) VALUES (6, 6, SYSDATE, 3000, 'Debit');

END;

/

BEGIN

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (1, 1, 50000, 5, TO\_DATE('2023-01-01', 'YYYY-MM-DD'), TO\_DATE('2025-01-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (2, 2, 60000, 6, TO\_DATE('2023-02-01', 'YYYY-MM-DD'), TO\_DATE('2025-02-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (3, 3, 70000, 7, TO\_DATE('2023-03-01', 'YYYY-MM-DD'), TO\_DATE('2025-03-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (4, 4, 80000, 8, TO\_DATE('2023-04-01', 'YYYY-MM-DD'), TO\_DATE('2025-04-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (5, 5, 90000, 9, TO\_DATE('2023-05-01', 'YYYY-MM-DD'), TO\_DATE('2025-05-01', 'YYYY-MM-DD'));

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (6, 6, 100000, 10, TO\_DATE('2023-06-01', 'YYYY-MM-DD'), TO\_DATE('2025-06-01', 'YYYY-MM-DD'));

END;

/

BEGIN

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (1, 'Ravi Shankar', 'Manager', 50000, 'Sales', TO\_DATE('2020-01-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (2, 'Kavitha Suresh', 'Analyst', 40000, 'Finance', TO\_DATE('2021-02-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (3, 'Mohan Kumar', 'Developer', 60000, 'IT', TO\_DATE('2019-03-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (4, 'Latha Narayan', 'HR', 45000, 'Human Resources', TO\_DATE('2020-04-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (5, 'Prakash Raj', 'Support', 35000, 'Customer Service', TO\_DATE('2021-05-01', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate) VALUES (6, 'Anitha Devi', 'Designer', 55000, 'Marketing', TO\_DATE('2019-06-01', 'YYYY-MM-DD'));

END;

/

**Main Solutions:**

**Exercise 1: Control Structures**

**Scenario 1:**

DECLARE

CURSOR customer\_cursor IS

SELECT CustomerID, Name, DOB

FROM Customers;

l\_customer\_id Customers.CustomerID%TYPE;

l\_name Customers.Name%TYPE;

l\_dob Customers.DOB%TYPE;

l\_age NUMBER;

BEGIN

FOR customer\_rec IN customer\_cursor LOOP

l\_customer\_id := customer\_rec.CustomerID;

l\_name := customer\_rec.Name;

l\_dob := customer\_rec.DOB;

-- Calculate age

l\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, l\_dob) / 12);

-- Check if age is above 60

IF l\_age > 60 THEN

-- Apply 1% discount to loan interest rates for this customer

UPDATE Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID = l\_customer\_id;

-- Print discount application message

DBMS\_OUTPUT.PUT\_LINE('1% discount applied to loan interest rate for Customer ID: ' || l\_customer\_id);

END IF;

END LOOP;

-- Commit the changes

COMMIT;

END;

/

**Output:**

1% discount applied to loan interest rate for Customer ID: 7

**Scenario 2:**

--adding IsVIP column to Customers table

ALTER TABLE Customers ADD IsVIP CHAR(1) DEFAULT 'N';

--performing logic in pl/sql

DECLARE

CURSOR customer\_cursor IS

SELECT CustomerID, Balance

FROM Customers;

l\_customer\_id Customers.CustomerID%TYPE;

l\_balance Customers.Balance%TYPE;

BEGIN

FOR customer\_rec IN customer\_cursor LOOP

l\_customer\_id := customer\_rec.CustomerID;

l\_balance := customer\_rec.Balance;

-- Check if balance is over $10,000

IF l\_balance > 10000 THEN

-- Set IsVIP to 'Y' for this customer

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = l\_customer\_id;

-- Print VIP promotion message

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || l\_customer\_id || ' has been promoted to VIP status.');

END IF;

END LOOP;

-- Commit the changes

COMMIT;

END;

/

**Output:**

**Statement processed.**Customer ID: 2 has been promoted to VIP status.  
Customer ID: 3 has been promoted to VIP status.  
Customer ID: 4 has been promoted to VIP status.  
Customer ID: 5 has been promoted to VIP status.  
Customer ID: 6 has been promoted to VIP status.  
Customer ID: 7 has been promoted to VIP status.

**Scenario 3:**

--inserting a new row to get that output

BEGIN

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate) VALUES (7, 1, 50000, 5, TO\_DATE('2023-01-01', 'YYYY-MM-DD'), TO\_DATE('2024-08-09', 'YYYY-MM-DD'));

END;

/

--pl/sql logic

SET SERVEROUTPUT ON;

DECLARE

CURSOR loan\_cursor IS

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

FROM Loans l

JOIN Customers c ON l.CustomerID = c.CustomerID

WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30;

l\_loan\_id Loans.LoanID%TYPE;

l\_customer\_id Customers.CustomerID%TYPE;

l\_end\_date Loans.EndDate%TYPE;

l\_customer\_name Customers.Name%TYPE;

BEGIN

FOR loan\_rec IN loan\_cursor LOOP

l\_loan\_id := loan\_rec.LoanID;

l\_customer\_id := loan\_rec.CustomerID;

l\_end\_date := loan\_rec.EndDate;

l\_customer\_name := loan\_rec.Name;

-- Print reminder message

DBMS\_OUTPUT.PUT\_LINE('Reminder: Dear ' || l\_customer\_name || ', your loan with ID ' || l\_loan\_id || ' is due on ' || TO\_CHAR(l\_end\_date, 'DD-MON-YYYY') || '. Please make sure to pay it by the due date.');

END LOOP;

END;

/

**Output:**

Reminder: Dear Ram Kumar, your loan with ID 7 is due on 09-AUG-2024. Please make sure to pay it by the due date.

**Exercise 2: Error Handling**

**Scenario 1:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_source\_account\_id IN Accounts.AccountID%TYPE,

p\_target\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN Accounts.Balance%TYPE

)

IS

insufficient\_funds EXCEPTION;

l\_source\_balance Accounts.Balance%TYPE;

l\_target\_balance Accounts.Balance%TYPE;

BEGIN

-- Start the transaction

SAVEPOINT start\_transaction;

-- Fetch the source account balance

SELECT Balance INTO l\_source\_balance

FROM Accounts

WHERE AccountID = p\_source\_account\_id

FOR UPDATE;

-- Check if the source account has sufficient funds

IF l\_source\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

-- Deduct the amount from the source account

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_source\_account\_id;

-- Add the amount to the target account

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_target\_account\_id;

-- Commit the transaction

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful from Account ' || p\_source\_account\_id || ' to Account ' || p\_target\_account\_id || ' for amount ' || p\_amount);

EXCEPTION

WHEN insufficient\_funds THEN

-- Log the error message

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in Account ' || p\_source\_account\_id);

-- Rollback to the savepoint

ROLLBACK TO start\_transaction;

WHEN OTHERS THEN

-- Handle other exceptions

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

-- Rollback to the savepoint

ROLLBACK TO start\_transaction;

END SafeTransferFunds;

/

**Output:**

Procedure created.

**Scenario 2:**

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_source\_account\_id IN Accounts.AccountID%TYPE,

p\_target\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN Accounts.Balance%TYPE

)

IS

insufficient\_funds EXCEPTION;

l\_source\_balance Accounts.Balance%TYPE;

l\_target\_balance Accounts.Balance%TYPE;

BEGIN

-- Start the transaction

SAVEPOINT start\_transaction;

-- Fetch the source account balance

SELECT Balance INTO l\_source\_balance

FROM Accounts

WHERE AccountID = p\_source\_account\_id

FOR UPDATE;

-- Check if the source account has sufficient funds

IF l\_source\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

-- Deduct the amount from the source account

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_source\_account\_id;

-- Add the amount to the target account

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_target\_account\_id;

-- Commit the transaction

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful from Account ' || p\_source\_account\_id || ' to Account ' || p\_target\_account\_id || ' for amount ' || p\_amount);

EXCEPTION

WHEN insufficient\_funds THEN

-- Log the error message

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in Account ' || p\_source\_account\_id);

-- Rollback to the savepoint

ROLLBACK TO start\_transaction;

WHEN OTHERS THEN

-- Handle other exceptions

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

-- Rollback to the savepoint

ROLLBACK TO start\_transaction;

END SafeTransferFunds;

/

BEGIN

SafeTransferFunds(1, 2, 500);

END;

/

**Output:**

Procedure created.   
Transfer successful from Account 1 to Account 2 for amount 500

**Scenario 3:**

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id IN Customers.CustomerID%TYPE,

p\_name IN Customers.Name%TYPE,

p\_dob IN Customers.DOB%TYPE,

p\_balance IN Customers.Balance%TYPE

)

IS

customer\_exists EXCEPTION;

PRAGMA EXCEPTION\_INIT(customer\_exists, -00001); -- Initialize exception for duplicate key

BEGIN

-- Attempt to insert a new customer

BEGIN

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

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

DBMS\_OUTPUT.PUT\_LINE('Customer added successfully with ID ' || p\_customer\_id);

-- Commit the transaction

COMMIT;

EXCEPTION

WHEN customer\_exists THEN

-- Handle the case where the customer ID already exists

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

-- Rollback the transaction

ROLLBACK;

END;

EXCEPTION

WHEN OTHERS THEN

-- Handle other exceptions

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

-- Rollback the transaction

ROLLBACK;

END AddNewCustomer;

/

BEGIN

AddNewCustomer(1, 'John', TO\_DATE('1980-01-15', 'YYYY-MM-DD'), 5000);

END;

/

**Output:**

Error: Customer with ID 1 already exists.

**Exercise 3: Stored Procedures**

**Scenario 1:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest

IS

l\_account\_id Accounts.AccountID%TYPE;

l\_current\_balance Accounts.Balance%TYPE;

l\_new\_balance Accounts.Balance%TYPE;

l\_interest\_rate CONSTANT NUMBER := 0.01; -- 1% interest rate

BEGIN

-- Cursor to select all savings accounts

FOR account\_rec IN (SELECT AccountID, Balance FROM Accounts WHERE AccountType = 'Savings' FOR UPDATE)

LOOP

l\_account\_id := account\_rec.AccountID;

l\_current\_balance := account\_rec.Balance;

-- Calculate the new balance with interest

l\_new\_balance := l\_current\_balance + (l\_current\_balance \* l\_interest\_rate);

-- Update the account balance

UPDATE Accounts

SET Balance = l\_new\_balance,

LastModified = SYSDATE

WHERE AccountID = l\_account\_id;

-- Print a message for each account processed

DBMS\_OUTPUT.PUT\_LINE('Account ID ' || l\_account\_id || ' updated. New Balance: ' || l\_new\_balance);

END LOOP;

-- Commit the transaction

COMMIT;

EXCEPTION

WHEN OTHERS THEN

-- Handle other exceptions

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

-- Rollback the transaction

ROLLBACK;

END ProcessMonthlyInterest;

/

BEGIN

ProcessMonthlyInterest;

END;

/

**Output:**

Account ID 1 updated. New Balance: 9595  
Account ID 3 updated. New Balance: 20200  
Account ID 5 updated. New Balance: 25250

**Scenario 2:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department IN Employees.Department%TYPE,

p\_bonus\_percentage IN NUMBER

)

IS

l\_bonus\_amount Employees.Salary%TYPE;

BEGIN

-- Cursor to select all employees in the specified department

FOR employee\_rec IN (SELECT EmployeeID, Salary FROM Employees WHERE Department = p\_department FOR UPDATE)

LOOP

-- Calculate the bonus amount

l\_bonus\_amount := employee\_rec.Salary \* p\_bonus\_percentage / 100;

-- Update the employee's salary with the bonus

UPDATE Employees

SET Salary = Salary + l\_bonus\_amount

WHERE EmployeeID = employee\_rec.EmployeeID;

-- Print a message for each employee processed

DBMS\_OUTPUT.PUT\_LINE('Employee ID ' || employee\_rec.EmployeeID || ' updated. New Salary: ' || (employee\_rec.Salary + l\_bonus\_amount));

END LOOP;

-- Commit the transaction

COMMIT;

EXCEPTION

WHEN OTHERS THEN

-- Handle other exceptions

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

-- Rollback the transaction

ROLLBACK;

END UpdateEmployeeBonus;

/

BEGIN

UpdateEmployeeBonus('Sales', 10); -- Replace 'Sales' with the desired department and 10 with the bonus percentage

END;

/

**Output:**

Employee ID 1 updated. New Salary: 55000

**Scenario 3:**

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_source\_account\_id IN Accounts.AccountID%TYPE,

p\_dest\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

)

IS

l\_source\_balance Accounts.Balance%TYPE;

l\_dest\_balance Accounts.Balance%TYPE;

insufficient\_funds EXCEPTION;

BEGIN

-- Lock the source and destination accounts for update

SELECT Balance INTO l\_source\_balance

FROM Accounts

WHERE AccountID = p\_source\_account\_id

FOR UPDATE;

SELECT Balance INTO l\_dest\_balance

FROM Accounts

WHERE AccountID = p\_dest\_account\_id

FOR UPDATE;

-- Check if the source account has sufficient balance

IF l\_source\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

-- Deduct the amount from the source account

UPDATE Accounts

SET Balance = Balance - p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_source\_account\_id;

-- Add the amount to the destination account

UPDATE Accounts

SET Balance = Balance + p\_amount,

LastModified = SYSDATE

WHERE AccountID = p\_dest\_account\_id;

-- Print a success message

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount || ' from Account ID ' || p\_source\_account\_id || ' to Account ID ' || p\_dest\_account\_id || ' completed successfully.');

-- Commit the transaction

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

-- Handle insufficient funds case

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in Account ID ' || p\_source\_account\_id || '. Transfer aborted.');

-- Rollback the transaction

ROLLBACK;

WHEN NO\_DATA\_FOUND THEN

-- Handle account not found case

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

-- Rollback the transaction

ROLLBACK;

WHEN OTHERS THEN

-- Handle other exceptions

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

-- Rollback the transaction

ROLLBACK;

END TransferFunds;

/

BEGIN

TransferFunds(1, 2, 500); -- Replace 101 and 102 with actual account IDs and 500 with the amount to transfer

END;

/

**Output:**

Transfer of 500 from Account ID 1 to Account ID 2 completed successfully.

**Exercise 4: Functions**

**Scenario 1:**

CREATE OR REPLACE FUNCTION CalculateAge (

p\_dob IN DATE

) RETURN NUMBER

IS

l\_age NUMBER;

BEGIN

-- Calculate the age in years

l\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, p\_dob) / 12);

RETURN l\_age;

EXCEPTION

WHEN OTHERS THEN

-- Handle other exceptions

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

RETURN NULL;

END CalculateAge;

/

DECLARE

v\_dob DATE;

v\_age NUMBER;

BEGIN

v\_dob := TO\_DATE('1985-08-06', 'YYYY-MM-DD');

v\_age := CalculateAge(v\_dob);

DBMS\_OUTPUT.PUT\_LINE('Customer Age: ' || v\_age);

END;

/

**Output:**

Customer Age: 39

**Scenario 2:**

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount IN NUMBER,

p\_annual\_interest\_rate IN NUMBER,

p\_loan\_duration\_years IN NUMBER

) RETURN NUMBER

IS

l\_monthly\_interest\_rate NUMBER;

l\_total\_payments NUMBER;

l\_monthly\_installment NUMBER;

BEGIN

-- Convert annual interest rate to monthly interest rate

l\_monthly\_interest\_rate := p\_annual\_interest\_rate / 12 / 100;

-- Calculate the total number of payments

l\_total\_payments := p\_loan\_duration\_years \* 12;

-- Calculate the monthly installment using the loan amortization formula

l\_monthly\_installment := p\_loan\_amount \* l\_monthly\_interest\_rate /

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

RETURN l\_monthly\_installment;

EXCEPTION

WHEN OTHERS THEN

-- Handle other exceptions

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

RETURN NULL;

END CalculateMonthlyInstallment;

/

DECLARE

v\_loan\_amount NUMBER := 100000; -- Example loan amount

v\_annual\_interest\_rate NUMBER := 5; -- Example annual interest rate (5%)

v\_loan\_duration\_years NUMBER := 10; -- Example loan duration (10 years)

v\_monthly\_installment NUMBER;

BEGIN

v\_monthly\_installment := CalculateMonthlyInstallment(v\_loan\_amount, v\_annual\_interest\_rate, v\_loan\_duration\_years);

DBMS\_OUTPUT.PUT\_LINE('Monthly Installment: ' || v\_monthly\_installment);

END;

/

**Output:**

Monthly Installment: 1060.655152390752322182798044295508427298

**Scenario 3:**

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_account\_id IN Accounts.AccountID%TYPE,

p\_amount IN NUMBER

) RETURN BOOLEAN

IS

l\_balance Accounts.Balance%TYPE;

BEGIN

-- Fetch the balance of the specified account

SELECT Balance INTO l\_balance

FROM Accounts

WHERE AccountID = p\_account\_id;

-- Compare the balance with the specified amount

IF l\_balance >= p\_amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

-- Handle account not found case

DBMS\_OUTPUT.PUT\_LINE('Error: Account ID ' || p\_account\_id || ' not found.');

RETURN FALSE;

WHEN OTHERS THEN

-- Handle other exceptions

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

RETURN FALSE;

END HasSufficientBalance;

/

DECLARE

v\_account\_id NUMBER := 1; -- Example account ID

v\_amount NUMBER := 500; -- Example amount

v\_has\_sufficient\_balance BOOLEAN;

BEGIN

v\_has\_sufficient\_balance := HasSufficientBalance(v\_account\_id, v\_amount);

IF v\_has\_sufficient\_balance THEN

DBMS\_OUTPUT.PUT\_LINE('Account has sufficient balance.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Account does not have sufficient balance.');

END IF;

END;

/

**Output:**

Account has sufficient balance.

**Exercise 5: Triggers**

**Scenario 1:**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

AFTER UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END;

/

-- Update a customer's record (assuming a customer with CustomerID 1 exists)

UPDATE Customers

SET Name = 'Updated Name'

WHERE CustomerID = 1;

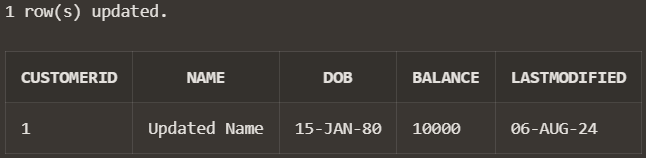
-- Check if the LastModified column has been updated

SELECT CustomerID, Name, DOB, Balance, LastModified

FROM Customers

WHERE CustomerID = 1;

**Output:**



**Scenario 2:**

--creating table

CREATE TABLE AuditLog (

AuditID INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

TransactionID INT,

AccountID INT,

TransactionDate DATE,

Amount INT,

TransactionType VARCHAR(10),

AuditTimestamp TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

Action VARCHAR(10)

);

--creating triggers

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (

TransactionID,

AccountID,

TransactionDate,

Amount,

TransactionType,

Action

)

VALUES (

:NEW.TransactionID,

:NEW.AccountID,

:NEW.TransactionDate,

:NEW.Amount,

:NEW.TransactionType,

'INSERT'

);

END;

/

--checking trigger

-- Insert a new transaction

INSERT INTO Transactions (

TransactionID,

AccountID,

TransactionDate,

Amount,

TransactionType

) VALUES (

9, -- Example TransactionID

9, -- Example AccountID

SYSDATE, -- Example TransactionDate

600, -- Example Amount

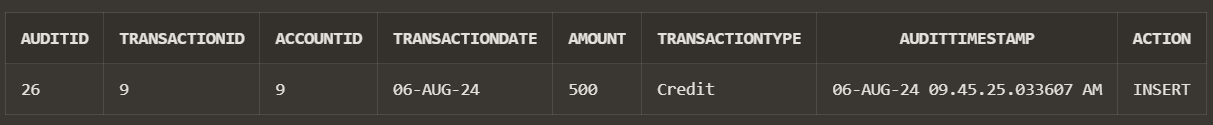
'Debit' -- Example TransactionType

);

-- Check the AuditLog table

SELECT \* FROM AuditLog;

**Output:**

****

**Scenario 3:**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

-- Fetch the current balance of the account

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

-- Check the transaction type and validate accordingly

IF :NEW.TransactionType = 'Withdrawal' THEN

IF :NEW.Amount > v\_balance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Withdrawal amount exceeds the current balance.');

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Deposit amount must be positive.');

END IF;

ELSE

RAISE\_APPLICATION\_ERROR(-20003, 'Invalid transaction type.');

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Account does not exist.');

WHEN OTHERS THEN

RAISE\_APPLICATION\_ERROR(-20005, 'An unexpected error occurred: ' || SQLERRM);

END;

/

-- Insert valid transactions

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

VALUES (1, 1, SYSDATE, 100, 'Deposit');

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

VALUES (2, 2, SYSDATE, 50, 'Withdrawal');

-- Insert invalid transactions

-- This should raise an error: 'Withdrawal amount exceeds the current balance.'

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

VALUES (3, 3, SYSDATE, 10000, 'Withdrawal');

-- This should raise an error: 'Deposit amount must be positive.'

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

VALUES (4, 4, SYSDATE, -50, 'Deposit');

**Output:**

ORA-20005: An unexpected error occurred: ORA-20002: Deposit amount must be positive.

**Exercise 6: Cursors**

**Scenario 1:**

DECLARE

-- Cursor to fetch customer details and their transactions for the current month

CURSOR customer\_cursor IS

SELECT

c.CustomerID,

c.Name,

a.AccountID,

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 >= TRUNC(SYSDATE, 'MM') -- Start of the current month

AND t.TransactionDate < TRUNC(SYSDATE, 'MM') + INTERVAL '1' MONTH; -- End of the current month

-- Record type for the cursor

customer\_record customer\_cursor%ROWTYPE;

BEGIN

-- Open the cursor

OPEN customer\_cursor;

-- Loop through all fetched rows

LOOP

FETCH customer\_cursor INTO customer\_record;

EXIT WHEN customer\_cursor%NOTFOUND;

-- Print statement for each customer

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || customer\_record.CustomerID);

DBMS\_OUTPUT.PUT\_LINE('Customer Name: ' || customer\_record.Name);

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || customer\_record.AccountID);

DBMS\_OUTPUT.PUT\_LINE('Transaction Date: ' || customer\_record.TransactionDate);

DBMS\_OUTPUT.PUT\_LINE('Amount: ' || customer\_record.Amount);

DBMS\_OUTPUT.PUT\_LINE('Transaction Type: ' || customer\_record.TransactionType);

DBMS\_OUTPUT.PUT\_LINE('------------------------------');

END LOOP;

-- Close the cursor

CLOSE customer\_cursor;

EXCEPTION

WHEN OTHERS THEN

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

END;

/

**Output:**

Statement processed.  
Customer ID: 1  
Customer Name: Updated Name  
Account ID: 1  
Transaction Date: 06-AUG-24  
Amount: 100  
Transaction Type: Deposit  
------------------------------  
Customer ID: 2  
Customer Name: Sita Devi  
Account ID: 2  
Transaction Date: 06-AUG-24  
Amount: 50  
Transaction Type: Withdrawal  
------------------------------  
Customer ID: 3  
Customer Name: Arun Vijay  
Account ID: 3  
Transaction Date: 06-AUG-24  
Amount: 10000  
Transaction Type: Withdrawal  
------------------------------

**Scenario 2:**

DECLARE

-- Define the annual fee amount

annual\_fee NUMBER := 50; -- Change this value to the actual annual fee

-- Cursor to fetch all accounts

CURSOR account\_cursor IS

SELECT

AccountID,

Balance

FROM

Accounts;

-- Record type for the cursor

account\_record account\_cursor%ROWTYPE;

BEGIN

-- Open the cursor

OPEN account\_cursor;

-- Loop through all fetched rows

LOOP

FETCH account\_cursor INTO account\_record;

EXIT WHEN account\_cursor%NOTFOUND;

-- Deduct the annual fee from the balance

UPDATE Accounts

SET Balance = Balance - annual\_fee

WHERE AccountID = account\_record.AccountID;

-- Optionally, print the account ID and new balance

DBMS\_OUTPUT.PUT\_LINE('Account ID: ' || account\_record.AccountID);

DBMS\_OUTPUT.PUT\_LINE('New Balance: ' || (account\_record.Balance - annual\_fee));

DBMS\_OUTPUT.PUT\_LINE('------------------------------');

END LOOP;

-- Close the cursor

CLOSE account\_cursor;

-- Commit the changes

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK; -- Rollback changes in case of error

END;

/

**Output:**

Statement processed.  
Account ID: 1  
New Balance: 9045  
------------------------------  
Account ID: 2  
New Balance: 15950  
------------------------------  
Account ID: 3  
New Balance: 20150  
------------------------------  
Account ID: 4  
New Balance: 17950  
------------------------------  
Account ID: 5  
New Balance: 25200  
------------------------------  
Account ID: 6  
New Balance: 29950  
------------------------------  
Account ID: 9  
New Balance: 4950  
------------------------------

**Scenario 3:**

DECLARE

-- Define the percentage increase for the interest rate

percentage\_increase NUMBER := 0.02; -- Example: 2% increase

-- Cursor to fetch all loans

CURSOR loan\_cursor IS

SELECT

LoanID,

InterestRate

FROM

Loans;

-- Record type for the cursor

loan\_record loan\_cursor%ROWTYPE;

BEGIN

-- Open the cursor

OPEN loan\_cursor;

-- Loop through all fetched rows

LOOP

FETCH loan\_cursor INTO loan\_record;

EXIT WHEN loan\_cursor%NOTFOUND;

-- Calculate the new interest rate

DECLARE

new\_interest\_rate NUMBER;

BEGIN

new\_interest\_rate := loan\_record.InterestRate \* (1 + percentage\_increase);

-- Update the interest rate in the Loans table

UPDATE Loans

SET InterestRate = new\_interest\_rate

WHERE LoanID = loan\_record.LoanID;

-- Optionally, print the Loan ID and new interest rate

DBMS\_OUTPUT.PUT\_LINE('Loan ID: ' || loan\_record.LoanID);

DBMS\_OUTPUT.PUT\_LINE('New Interest Rate: ' || new\_interest\_rate);

DBMS\_OUTPUT.PUT\_LINE('------------------------------');

END;

END LOOP;

-- Close the cursor

CLOSE loan\_cursor;

-- Commit the changes

COMMIT;

EXCEPTION

WHEN OTHERS THEN

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

ROLLBACK; -- Rollback changes in case of error

END;

/

**Output:**

Loan ID: 1  
New Interest Rate: 5.1  
------------------------------  
Loan ID: 2  
New Interest Rate: 6.12  
------------------------------  
Loan ID: 3  
New Interest Rate: 7.14  
------------------------------  
Loan ID: 4  
New Interest Rate: 8.16  
------------------------------  
Loan ID: 5  
New Interest Rate: 9.18  
------------------------------  
Loan ID: 6  
New Interest Rate: 10.2  
------------------------------  
Loan ID: 7  
New Interest Rate: 5.1  
------------------------------

**Exercise 7: Packages**

**Scenario 1:**

CREATE OR REPLACE PACKAGE CustomerManagement AS

-- Procedure to add a new customer

PROCEDURE AddNewCustomer(

p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2,

p\_DOB IN DATE,

p\_Balance IN NUMBER

);

-- Procedure to update customer details

PROCEDURE UpdateCustomerDetails(

p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2,

p\_DOB IN DATE,

p\_Balance IN NUMBER

);

-- Function to get the balance of a customer

FUNCTION GetCustomerBalance(

p\_CustomerID IN NUMBER

) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

-- Implementation of AddNewCustomer procedure

PROCEDURE AddNewCustomer(

p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2,

p\_DOB IN DATE,

p\_Balance IN NUMBER

) IS

BEGIN

BEGIN

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

VALUES (p\_CustomerID, p\_Name, p\_DOB, p\_Balance, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || p\_CustomerID || ' already exists.');

WHEN OTHERS THEN

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

END;

END AddNewCustomer;

-- Implementation of UpdateCustomerDetails procedure

PROCEDURE UpdateCustomerDetails(

p\_CustomerID IN NUMBER,

p\_Name IN VARCHAR2,

p\_DOB IN DATE,

p\_Balance IN NUMBER

) IS

BEGIN

BEGIN

UPDATE Customers

SET Name = p\_Name,

DOB = p\_DOB,

Balance = p\_Balance,

LastModified = SYSDATE

WHERE CustomerID = p\_CustomerID;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No customer found with ID ' || p\_CustomerID);

ELSE

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END;

END UpdateCustomerDetails;

-- Implementation of GetCustomerBalance function

FUNCTION GetCustomerBalance(

p\_CustomerID IN NUMBER

) RETURN NUMBER IS

v\_Balance NUMBER;

BEGIN

BEGIN

SELECT Balance INTO v\_Balance

FROM Customers

WHERE CustomerID = p\_CustomerID;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Customer ID ' || p\_CustomerID || ' not found.');

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END;

RETURN v\_Balance;

END GetCustomerBalance;

END CustomerManagement;

/

BEGIN

-- Add a new customer

CustomerManagement.AddNewCustomer(101, 'Khaliq', DATE '1980-01-01', 5000);

-- Update customer details

CustomerManagement.UpdateCustomerDetails(101, 'Khaliq', DATE '1980-01-01', 5500);

-- Get customer balance

DBMS\_OUTPUT.PUT\_LINE('Customer Balance: ' || CustomerManagement.GetCustomerBalance(101));

END;

/

**Output:**

Package Body created. Statement processed.  
Customer Balance: 5500

**Scenario 2:**

CREATE OR REPLACE PACKAGE EmployeeManagement AS

-- Procedure to hire a new employee

PROCEDURE HireEmployee(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Salary IN NUMBER,

p\_Department IN VARCHAR2,

p\_HireDate IN DATE

);

-- Procedure to update employee details

PROCEDURE UpdateEmployeeDetails(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Salary IN NUMBER,

p\_Department IN VARCHAR2

);

-- Function to calculate annual salary

FUNCTION CalculateAnnualSalary(

p\_EmployeeID IN NUMBER

) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

-- Implementation of HireEmployee procedure

PROCEDURE HireEmployee(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Salary IN NUMBER,

p\_Department IN VARCHAR2,

p\_HireDate IN DATE

) IS

BEGIN

BEGIN

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

VALUES (p\_EmployeeID, p\_Name, p\_Position, p\_Salary, p\_Department, p\_HireDate);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Employee ID ' || p\_EmployeeID || ' already exists.');

WHEN OTHERS THEN

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

END;

END HireEmployee;

-- Implementation of UpdateEmployeeDetails procedure

PROCEDURE UpdateEmployeeDetails(

p\_EmployeeID IN NUMBER,

p\_Name IN VARCHAR2,

p\_Position IN VARCHAR2,

p\_Salary IN NUMBER,

p\_Department IN VARCHAR2

) IS

BEGIN

BEGIN

UPDATE Employees

SET Name = p\_Name,

Position = p\_Position,

Salary = p\_Salary,

Department = p\_Department

WHERE EmployeeID = p\_EmployeeID;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No employee found with ID ' || p\_EmployeeID);

ELSE

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END;

END UpdateEmployeeDetails;

-- Implementation of CalculateAnnualSalary function

FUNCTION CalculateAnnualSalary(

p\_EmployeeID IN NUMBER

) RETURN NUMBER IS

v\_Salary NUMBER;

BEGIN

BEGIN

SELECT Salary INTO v\_Salary

FROM Employees

WHERE EmployeeID = p\_EmployeeID;

RETURN v\_Salary \* 12; -- Assuming the salary is monthly

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

RETURN NULL;

WHEN OTHERS THEN

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

RETURN NULL;

END;

END CalculateAnnualSalary;

END EmployeeManagement;

/

BEGIN

-- Hire a new employee

EmployeeManagement.HireEmployee(201, 'Kumar', 'Developer', 6000, 'IT', DATE '2024-08-01');

-- Update employee details

EmployeeManagement.UpdateEmployeeDetails(201, 'Kumar', 'Senior Developer', 7000, 'IT');

-- Calculate annual salary

DBMS\_OUTPUT.PUT\_LINE('Annual Salary: ' || EmployeeManagement.CalculateAnnualSalary(201));

END;

/

**Output:**

Package created.

Package Body created.

Statement processed.  
Annual Salary: 84000

**Scenario 3:**

CREATE OR REPLACE PACKAGE AccountOperations AS

-- Procedure to open a new account

PROCEDURE OpenAccount(

p\_AccountID IN NUMBER,

p\_CustomerID IN NUMBER,

p\_AccountType IN VARCHAR2,

p\_Balance IN NUMBER

);

-- Procedure to close an account

PROCEDURE CloseAccount(

p\_AccountID IN NUMBER

);

-- Function to get the total balance of a customer across all accounts

FUNCTION GetTotalBalance(

p\_CustomerID IN NUMBER

) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

-- Implementation of OpenAccount procedure

PROCEDURE OpenAccount(

p\_AccountID IN NUMBER,

p\_CustomerID IN NUMBER,

p\_AccountType IN VARCHAR2,

p\_Balance IN NUMBER

) IS

BEGIN

BEGIN

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

VALUES (p\_AccountID, p\_CustomerID, p\_AccountType, p\_Balance, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('Account ID ' || p\_AccountID || ' already exists.');

WHEN OTHERS THEN

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

END;

END OpenAccount;

-- Implementation of CloseAccount procedure

PROCEDURE CloseAccount(

p\_AccountID IN NUMBER

) IS

BEGIN

BEGIN

DELETE FROM Accounts

WHERE AccountID = p\_AccountID;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No account found with ID ' || p\_AccountID);

ELSE

COMMIT;

END IF;

EXCEPTION

WHEN OTHERS THEN

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

END;

END CloseAccount;

-- Implementation of GetTotalBalance function

FUNCTION GetTotalBalance(

p\_CustomerID IN NUMBER

) RETURN NUMBER IS

v\_TotalBalance NUMBER;

BEGIN

BEGIN

SELECT SUM(Balance) INTO v\_TotalBalance

FROM Accounts

WHERE CustomerID = p\_CustomerID;

IF v\_TotalBalance IS NULL THEN

RETURN 0;

ELSE

RETURN v\_TotalBalance;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN 0;

WHEN OTHERS THEN

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

RETURN NULL;

END;

END GetTotalBalance;

END AccountOperations;

/

BEGIN

-- Open a new account

AccountOperations.OpenAccount(301, 101, 'Savings', 2000);

-- Close an account

AccountOperations.CloseAccount(301);

-- Get total balance for a customer

DBMS\_OUTPUT.PUT\_LINE('Total Balance: ' || AccountOperations.GetTotalBalance(101));

END;

/

**Output:**

Package created.

Package Body created.

Statement processed.  
Total Balance: 0