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: 1060.6551523907523221827980442955084272980
```

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.
```

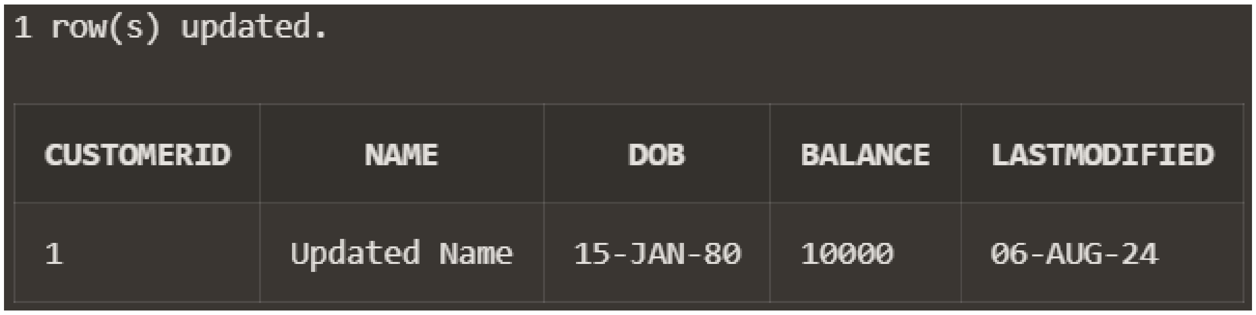
Exercise 5: Triggers

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:



```
1 row(s) updated.
```

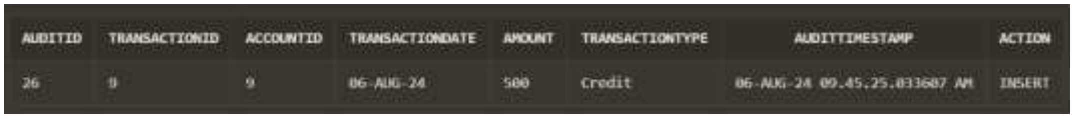
CUSTOMERID	NAME	DOB	BALANCE	LASTMODIFIED
1	Updated Name	15-JAN-80	10000	06-AUG-24

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:



```
AUDITID  TRANSACTIONID  ACCOUNTID  TRANSACTIONDATE  AMOUNT  TRANSACTIONTYPE  AUDITTIMESTAMP                ACTION
26       9              9          06-AUG-24        500     Credit           06-AUG-24 09.45.25.033607 AM  INSERT
```

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
```