**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.

**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.

**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.

**Scenario 1:**

ALTER TABLE CUSTOMERS ADD loan\_interest\_rate NUMBER(5, 2);

CREATE OR REPLACE PROCEDURE UpdateInterestRateForSeniorCustomers IS

CURSOR c\_customers IS

SELECT customerid, DOB, loan\_interest\_rate

FROM CUSTOMERS;

v\_new\_interest\_rate NUMBER;

v\_age NUMBER;

BEGIN

FOR customer IN c\_customers LOOP

-- Calculate age from DOB

SELECT TRUNC(MONTHS\_BETWEEN(SYSDATE, customer.DOB) / 12) INTO v\_age FROM dual;

IF v\_age > 60 THEN

v\_new\_interest\_rate := customer.loan\_interest\_rate - (customer.loan\_interest\_rate \* 0.01);

UPDATE CUSTOMERS

SET loan\_interest\_rate = v\_new\_interest\_rate

WHERE customerid = customer.customerid;

END IF;

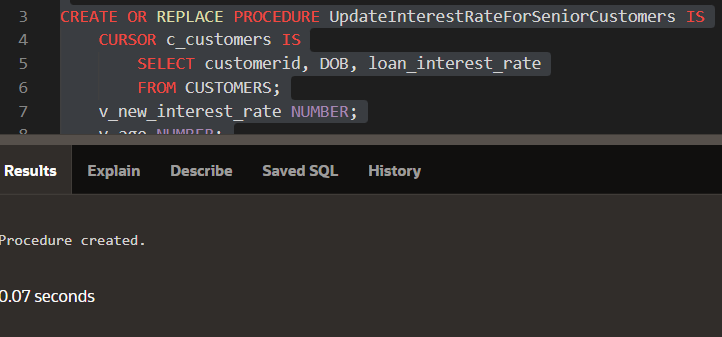
END LOOP;

COMMIT; -- Commit the changes to the database

END UpdateInterestRateForSeniorCustomers;

/

CALL UpdateInterestRateForSeniorCustomers;



**Scenario 2:**

ALTER TABLE CUSTOMERS ADD IsVIP VARCHAR2(3);

BEGIN

FOR rec IN (SELECT customerid, balance FROM accounts WHERE balance > 10000) LOOP

UPDATE customers

SET IsVIP = 'TRUE'

WHERE customerid = rec.customerid;

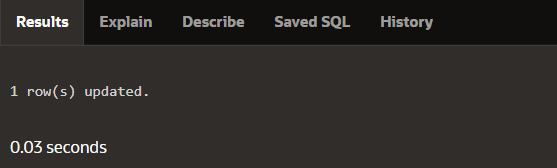
END LOOP;

COMMIT;

END;

/

CALL UpdateInterestRateForSeniorCustomers;



**Scenario 3:**

BEGIN

FOR rec IN (SELECT customerid, balance FROM accounts WHERE balance > 10000) LOOP

DBMS\_OUTPUT.PUT\_LINE('Processing Customer ID: ' || rec.customerid || ' with Balance: ' || rec.balance);

UPDATE customers

SET IsVIP = 'TRUE'

WHERE customerid = rec.customerid;

DBMS\_OUTPUT.PUT\_LINE('Updated Customer ID: ' || rec.customerid || ' to VIP status');

END LOOP;

COMMIT;

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.

**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.

**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.

**SCENARIO 1**:

CREATE PROCEDURE SafeTransferFunds(

IN fromAccountID INT,

IN toAccountID INT,

IN transferAmount DECIMAL(10, 2)

)

BEGIN

DECLARE insufficientFunds BOOL DEFAULT FALSE;

DECLARE accountNotFound BOOL DEFAULT FALSE;

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

-- Rollback transaction and log the error

ROLLBACK;

SELECT 'An error occurred during the transfer. Transaction rolled back.' AS ErrorMessage;

END;

START TRANSACTION;

-- Check if both accounts exist

IF (SELECT COUNT(\*) FROM Accounts WHERE AccountID = fromAccountID) = 0 THEN

SET accountNotFound = TRUE;

END IF;

IF (SELECT COUNT(\*) FROM Accounts WHERE AccountID = toAccountID) = 0 THEN

SET accountNotFound = TRUE;

END IF;

IF accountNotFound THEN

ROLLBACK;

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'One or both accounts do not exist';

END IF;

-- Check if the fromAccount has sufficient balance

IF (SELECT Balance FROM Accounts WHERE AccountID = fromAccountID) < transferAmount THEN

SET insufficientFunds = TRUE;

END IF;

IF insufficientFunds THEN

ROLLBACK;

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Insufficient funds in the from account';

END IF;

-- Perform the transfer

UPDATE Accounts SET Balance = Balance - transferAmount WHERE AccountID = fromAccountID;

UPDATE Accounts SET Balance = Balance + transferAmount WHERE AccountID = toAccountID;

COMMIT;

SELECT 'Transfer completed successfully' AS SuccessMessage;

END //

DELIMITER ;

**Scenario 2**:

CREATE PROCEDURE UpdateSalary(

IN empID INT,

IN salaryIncrease DECIMAL(10, 2)

)

BEGIN

DECLARE employeeNotFound BOOL DEFAULT FALSE;

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

-- Log error and rollback

ROLLBACK;

SELECT 'An error occurred while updating the salary. Transaction rolled back.' AS ErrorMessage;

END;

START TRANSACTION;

-- Check if employee exists

IF (SELECT COUNT(\*) FROM Employees WHERE EmployeeID = empID) = 0 THEN

SET employeeNotFound = TRUE;

END IF;

IF employeeNotFound THEN

ROLLBACK;

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Employee ID does not exist';

END IF;

-- Update the salary

UPDATE Employees SET Salary = Salary + salaryIncrease WHERE EmployeeID = empID;

COMMIT;

SELECT 'Salary updated successfully' AS SuccessMessage;

END //

DELIMITER ;

**Scenario 3**:

CREATE PROCEDURE AddNewCustomer(

IN newCustomerID INT,

IN newName VARCHAR(100),

IN newDOB DATE,

IN newBalance DECIMAL(10, 2)

)

BEGIN

DECLARE customerExists BOOL DEFAULT FALSE;

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

-- Log error and rollback

ROLLBACK;

SELECT 'An error occurred while adding the new customer. Transaction rolled back.' AS ErrorMessage;

END;

START TRANSACTION;

-- Check if customer already exists

IF (SELECT COUNT(\*) FROM Customers WHERE CustomerID = newCustomerID) > 0 THEN

SET customerExists = TRUE;

END IF;

IF customerExists THEN

ROLLBACK;

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Customer with this ID already exists';

END IF;

-- Insert new customer

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

VALUES (newCustomerID, newName, newDOB, newBalance, CURRENT\_TIMESTAMP);

COMMIT;

SELECT 'New customer added successfully' AS SuccessMessage;

END //

DELIMITER ;

SELECT COUNT(\*) FROM Employees WHERE EmployeeID = 1;

UPDATE Employees SET Salary = Salary + 1000 WHERE EmployeeID = 1;

SELECT \* FROM Employees WHERE EmployeeID = 1;

SELECT COUNT(\*) FROM Customers WHERE CustomerID = 3;

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

VALUES (3, 'New Customer', '2000-01-01', 5000, CURRENT\_TIMESTAMP);

SELECT \* FROM Customers WHERE CustomerID = 3;

CALL UpdateSalary(1, 1000);

CALL AddNewCustomer(3, 'New Customer', '2000-01-01', 5000);

SELECT \* FROM Employees WHERE EmployeeID = 1;

SELECT \* FROM Customers WHERE CustomerID = 3;

CALL UpdateSalary(1, 1000);

CALL AddNewCustomer(3, 'New Customer', '2000-01-01', 5000);

CALL SafeTransferFunds(1, 2, 100.00);

**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.

**SCENARIO 1:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

DECLARE

CURSOR cur IS SELECT account\_id, balance FROM savings\_accounts;

account\_id savings\_accounts.account\_id%TYPE;

balance savings\_accounts.balance%TYPE;

interest\_rate CONSTANT NUMBER := 0.01;

BEGIN

OPEN cur;

LOOP

FETCH cur INTO account\_id, balance;

EXIT WHEN cur%NOTFOUND;

-- Calculate new balance with interest

balance := balance + (balance \* interest\_rate);

-- Update the balance

UPDATE savings\_accounts

SET balance = balance

WHERE account\_id = account\_id;

END LOOP;

CLOSE cur;

END;

END ProcessMonthlyInterest;

/

EXEC ProcessMonthlyInterest;

**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.

**SCEANRIO 2:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(dept\_id NUMBER, bonus\_percentage NUMBER) IS

BEGIN

-- Update the salary by adding the bonus percentage

UPDATE employees

SET salary = salary + (salary \* (bonus\_percentage / 100))

WHERE department\_id = dept\_id;

END UpdateEmployeeBonus;

/

CALL UpdateEmployeeBonus(5, 10);

**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.

**SCENARIO 3:**

CREATE OR REPLACE PROCEDURE TransferFunds(

source\_account\_id NUMBER,

target\_account\_id NUMBER,

amount NUMBER

) IS

source\_balance NUMBER;

BEGIN

-- Check the balance of the source account

SELECT balance INTO source\_balance FROM accounts WHERE account\_id = source\_account\_id;

IF source\_balance >= amount THEN

-- Deduct amount from source account

UPDATE accounts

SET balance = balance - amount

WHERE account\_id = source\_account\_id;

-- Add amount to target account

UPDATE accounts

SET balance = balance + amount

WHERE account\_id = target\_account\_id;

ELSE

-- Handle insufficient funds

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in source account');

END IF;

END TransferFunds;

/

CALL TransferFunds(1, 2, 100);

**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.

**QUERY:**

CREATE OR REPLACE FUNCTION Calculate\_Age(

DOB IN DATE

)

RETURN NUMBER

IS

age NUMBER;

BEGIN

age := TRUNC(MONTHS\_BETWEEN(SYSDATE, DOB) / 12);

RETURN age;

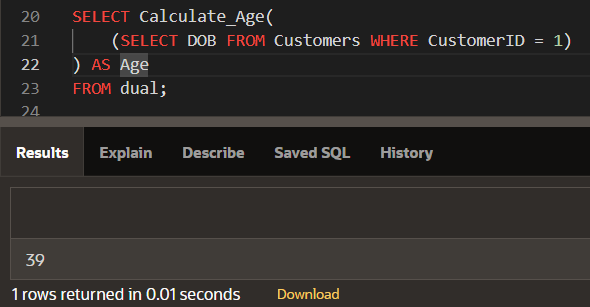
END Calculate\_Age;

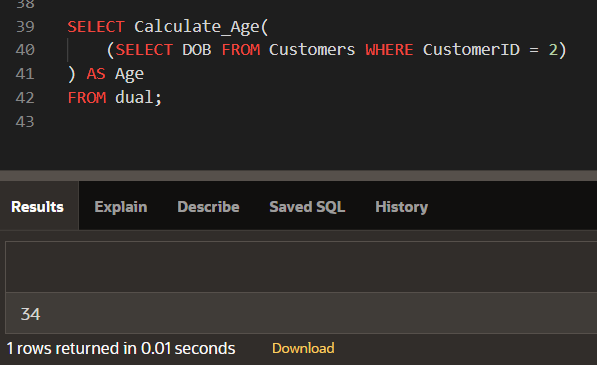
SELECT Calculate\_Age(

(SELECT DOB FROM Customers WHERE CustomerID = 1)

) AS Age

FROM dual;





**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.

**QUERY:**

CREATE OR REPLACE FUNCTION Calculate\_EMI(

loan\_amount IN NUMBER,

annual\_interest\_rate IN NUMBER,

total\_months IN NUMBER

) RETURN NUMBER

IS

monthly\_interest\_rate NUMBER;

emi NUMBER;

BEGIN

monthly\_interest\_rate := annual\_interest\_rate / (12 \* 100);

emi := (loan\_amount \* monthly\_interest\_rate \* POWER((1 + monthly\_interest\_rate), total\_months)) /

(POWER((1 + monthly\_interest\_rate), total\_months) - 1);

RETURN emi;

END Calculate\_EMI;

SELECT ROUND(

Calculate\_EMI(

(SELECT LoanAmount FROM Loans WHERE LoanID = 1),

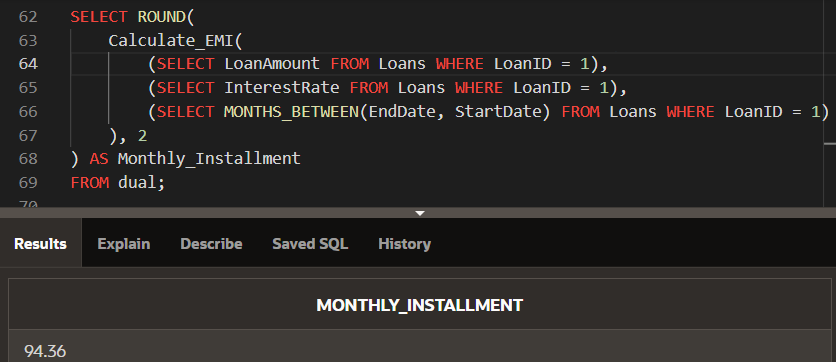
(SELECT InterestRate FROM Loans WHERE LoanID = 1),

(SELECT MONTHS\_BETWEEN(EndDate, StartDate) FROM Loans WHERE LoanID = 1)

), 2

) AS Monthly\_Installment

FROM dual;



**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.

**QUERY:**

CREATE OR REPLACE FUNCTION Check\_Account\_Balance(

p\_AccountID IN NUMBER,

p\_Amount IN NUMBER

) RETURN BOOLEAN

IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance

FROM Accounts

WHERE AccountID = p\_AccountID;

IF v\_Balance >= p\_Amount THEN

RETURN TRUE;

ELSE

RETURN FALSE;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END Check\_Account\_Balance;

DECLARE

v\_Result BOOLEAN;

BEGIN

v\_Result := Check\_Account\_Balance(1, 500);

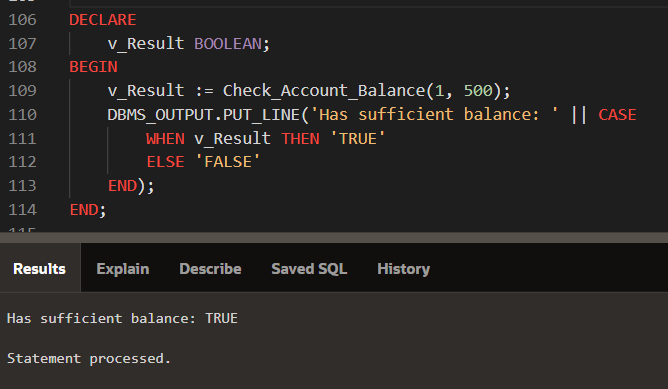
DBMS\_OUTPUT.PUT\_LINE('Has sufficient balance: ' || CASE

WHEN v\_Result THEN 'TRUE'

ELSE 'FALSE'

END);

END;



**Exercise 5: Triggers**

Scenario 1: Automatically update the last modified date when a customer's record is updated.

* Question: Write a trigger UpdateCustomerLastModified that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.

Scenario 2: Maintain an audit log for all transactions.

o Question: Write a trigger LogTransaction that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

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

**Scenario 1**

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

BEFORE UPDATE ON Customers

FOR EACH ROW

BEGIN

:NEW.LastModified := SYSDATE;

END UpdateCustomerLastModified;

/

**Scenario 2**

CREATE TABLE AuditLog (

AuditID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

TransactionID NUMBER,

AccountID NUMBER,

TransactionDate DATE,

Amount NUMBER,

TransactionType VARCHAR2(10),

LogDate DATE

);

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (

TransactionID,

AccountID,

TransactionDate,

Amount,

TransactionType,

LogDate

) VALUES (

:NEW.TransactionID,

:NEW.AccountID,

:NEW.TransactionDate,

:NEW.Amount,

:NEW.TransactionType,

SYSDATE

);

END LogTransaction;

/

**Scenario 3**

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_current\_balance NUMBER;

BEGIN

IF :NEW.TransactionType = 'Withdrawal' THEN

SELECT Balance INTO v\_current\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

IF :NEW.Amount > v\_current\_balance THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds for withdrawal.');

END IF;

ELSIF :NEW.TransactionType = 'Deposit' THEN

IF :NEW.Amount <= 0 THEN

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

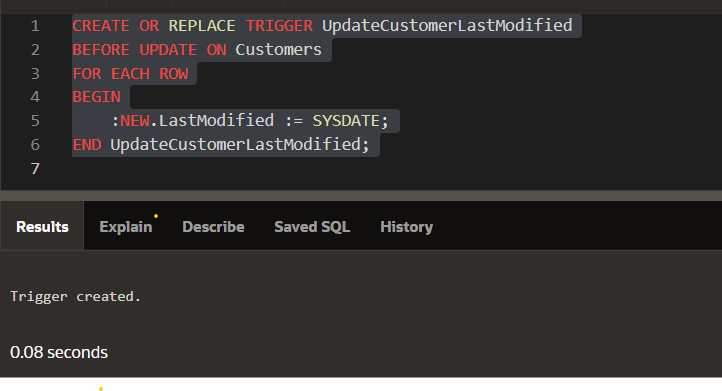
END IF;

ELSE

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

END IF;

END CheckTransactionRules;



**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.

**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.

**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.

**Scenario 1: Generate Monthly Statements**

DECLARE

CURSOR trans\_cursor IS

SELECT CustomerID, TransactionDate, Amount, TransactionType

FROM Transactions

WHERE TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND LAST\_DAY(SYSDATE);

BEGIN

FOR rec IN trans\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('Customer ' || rec.CustomerID || ': ' || rec.TransactionDate || ', ' || rec.Amount || ', ' || rec.TransactionType);

END LOOP;

END;

/

**Scenario 2: Apply Annual Fee**

DECLARE

CURSOR acc\_cursor IS

SELECT AccountID, Balance

FROM Accounts;

v\_fee NUMBER := 50;

BEGIN

FOR rec IN acc\_cursor LOOP

UPDATE Accounts

SET Balance = Balance - v\_fee

WHERE AccountID = rec.AccountID;

END LOOP;

END;

/

**Scenario 3: Update Loan Interest Rates**

DECLARE

CURSOR loan\_cursor IS

SELECT LoanID, InterestRate

FROM Loans;

v\_new\_rate NUMBER := 5;

BEGIN

FOR rec IN loan\_cursor LOOP

UPDATE Loans

SET InterestRate = v\_new\_rate

WHERE LoanID = rec.LoanID;

END LOOP;

END;

/