**Exercise 1: Control Structures**

**Scenario 1: Discount for Customers Above 60:**

DECLARE

CURSOR cust\_cursor IS

SELECT customer\_id, age, loan\_interest\_rate

FROM customers;

cust\_rec cust\_cursor%ROWTYPE;

BEGIN

OPEN cust\_cursor;

LOOP

FETCH cust\_cursor INTO cust\_rec;

EXIT WHEN cust\_cursor%NOTFOUND;

IF cust\_rec.age > 60 THEN

UPDATE loans

SET interest\_rate = interest\_rate \* 0.99

WHERE customer\_id = cust\_rec.customer\_id;

END IF;

END LOOP;

CLOSE cust\_cursor;

END;

/

**Scenario 2: VIP Status Based on Balance**

SQL

DECLARE

CURSOR cust\_cursor IS

SELECT customer\_id, balance

FROM customers;

cust\_rec cust\_cursor%ROWTYPE;

BEGIN

OPEN cust\_cursor;

LOOP

FETCH cust\_cursor INTO cust\_rec;

EXIT WHEN cust\_cursor%NOTFOUND;

IF cust\_rec.balance > 10000 THEN

UPDATE customers

SET is\_vip = 'TRUE'

WHERE customer\_id = cust\_rec.customer\_id;

END IF;

END LOOP;

CLOSE cust\_cursor;

END;

/

**Scenario 3: Loan Due Reminders**

SQL

DECLARE

CURSOR loan\_cursor IS

SELECT c.customer\_name, l.due\_date

FROM customers c

JOIN loans l ON c.customer\_id = l.customer\_id

WHERE l.due\_date BETWEEN SYSDATE AND SYSDATE + 30;

loan\_rec loan\_cursor%ROWTYPE;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO loan\_rec;

EXIT WHEN loan\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan due on ' || loan\_rec.due\_date || ' for customer ' || loan\_rec.customer\_name);

END LOOP;

CLOSE loan\_cursor;

END;

/

**Exercise 2: Error Handling**

**Scenario 1: Safe Fund Transfer**

SQL

CREATE OR REPLACE PROCEDURE SafeTransferFunds (

p\_from\_account\_id NUMBER,

p\_to\_account\_id NUMBER,

p\_amount NUMBER

)

IS

BEGIN

DECLARE

v\_from\_balance NUMBER;

v\_to\_balance NUMBER;

BEGIN

-- Check balance before transfer

SELECT balance INTO v\_from\_balance FROM accounts WHERE account\_id = p\_from\_account\_id;

IF v\_from\_balance < p\_amount THEN

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

END IF;

-- Start transaction

SAVEPOINT transfer\_start;

UPDATE accounts SET balance = balance - p\_amount WHERE account\_id = p\_from\_account\_id;

UPDATE accounts SET balance = balance + p\_amount WHERE account\_id = p\_to\_account\_id;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK TO transfer\_start;

-- Log error (replace with your logging mechanism)

INSERT INTO error\_log (error\_message, error\_time)

VALUES (SQLERRM, SYSDATE);

END;

END;

/

Use code [with caution.](/faq#coding)

**Scenario 2: Update Salary with Error Handling**

SQL

CREATE OR REPLACE PROCEDURE UpdateSalary (

p\_employee\_id NUMBER,

p\_percentage NUMBER

)

IS

BEGIN

UPDATE employees

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

WHERE employee\_id = p\_employee\_id;

IF SQL%NOTFOUND THEN

-- Log error (replace with your logging mechanism)

INSERT INTO error\_log (error\_message, error\_time)

VALUES ('Employee ID ' || p\_employee\_id || ' not found', SYSDATE);

END IF;

EXCEPTION

WHEN OTHERS THEN

-- Log error (replace with your logging mechanism)

INSERT INTO error\_log (error\_message, error\_time)

VALUES (SQLERRM, SYSDATE);

END;

/

**Scenario 3: Adding New Customer with Error Handling**

SQL

CREATE OR REPLACE PROCEDURE AddNewCustomer (

p\_customer\_id NUMBER,

p\_name VARCHAR2,

-- other customer details

)

IS

BEGIN

INSERT INTO customers (customer\_id, name, -- other columns)

VALUES (p\_customer\_id, p\_name, -- other values);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

-- Log error (replace with your logging mechanism)

INSERT INTO error\_log (error\_message, error\_time)

VALUES ('Duplicate customer ID: ' || p\_customer\_id, SYSDATE);

WHEN OTHERS THEN

-- Log error (replace with your logging mechanism)

INSERT INTO error\_log (error\_message, error\_time)

VALUES (SQLERRM, SYSDATE);

END;

/

**Exercise 3: Stored Procedures**

Scenario 1: Process Monthly Interest

SQL

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest

IS

BEGIN

UPDATE savings\_accounts

SET balance = balance \* 1.01;

END;

/

Use code [with caution.](/faq#coding)

This procedure simply updates the balance of all savings accounts by multiplying it by 1.01, effectively applying a 1% interest rate.

Scenario 2: Update Employee Bonus

SQL

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_department\_id NUMBER,

p\_bonus\_percentage NUMBER

)

IS

BEGIN

UPDATE employees

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

WHERE department\_id = p\_department\_id;

END;

/

Use code [with caution.](/faq#coding)

This procedure takes a department ID and a bonus percentage as input and updates the salaries of all employees in that department by the specified percentage.

Scenario 3: Transfer Funds

SQL

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_from\_account\_id NUMBER,

p\_to\_account\_id NUMBER,

p\_amount NUMBER

)

IS

v\_from\_balance NUMBER;

BEGIN

-- Check balance before transfer

SELECT balance INTO v\_from\_balance FROM accounts WHERE account\_id = p\_from\_account\_id;

IF v\_from\_balance < p\_amount THEN

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

END IF;

UPDATE accounts

SET balance = balance - p\_amount

WHERE account\_id = p\_from\_account\_id;

UPDATE accounts

SET balance = balance + p\_amount

WHERE account\_id = p\_to\_account\_id;

END;

/

**Exercise 4: Functions**

Scenario 1: Calculate Customer Age

SQL

CREATE OR REPLACE FUNCTION CalculateAge (p\_birthdate DATE)

RETURN NUMBER

IS

v\_age NUMBER;

BEGIN

v\_age := TRUNC(SYSDATE - p\_birthdate) / 365;

RETURN v\_age;

END;

/

Use code [with caution.](/faq#coding)

This function takes a birthdate as input and calculates the age in years by subtracting the birthdate from the current date and dividing by 365.

Scenario 2: Calculate Monthly Installment

SQL

CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment (

p\_loan\_amount NUMBER,

p\_interest\_rate NUMBER,

p\_loan\_duration NUMBER

)

RETURN NUMBER

IS

v\_monthly\_interest\_rate NUMBER;

v\_numerator NUMBER;

v\_denominator NUMBER;

v\_monthly\_installment NUMBER;

BEGIN

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

v\_numerator := v\_monthly\_interest\_rate \* POWER(1 + v\_monthly\_interest\_rate, p\_loan\_duration \* 12);

v\_denominator := POWER(1 + v\_monthly\_interest\_rate, p\_loan\_duration \* 12) - 1;

v\_monthly\_installment := p\_loan\_amount \* (v\_numerator / v\_denominator);

RETURN v\_monthly\_installment;

END;

/

Use code [with caution.](/faq#coding)

This function calculates the monthly installment for a loan based on the provided loan amount, interest rate, and loan duration using the standard loan amortization formula.

Scenario 3: Check Sufficient Balance

SQL

CREATE OR REPLACE FUNCTION HasSufficientBalance (

p\_account\_id NUMBER,

p\_amount NUMBER

)

RETURN BOOLEAN

IS

v\_balance NUMBER;

BEGIN

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

RETURN v\_balance >= p\_amount;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RETURN FALSE;

END;

/

**Exercise 5: Triggers**

Scenario 1: Update Customer Last Modified

SQL

CREATE OR REPLACE TRIGGER UpdateCustomerLastModified

AFTER UPDATE ON Customers

FOR EACH ROW

BEGIN

UPDATE Customers

SET LastModified = SYSDATE

WHERE Customer\_ID = :NEW.Customer\_ID;

END;

/

Use code [with caution.](/faq#coding)

This trigger updates the LastModified column of the Customers table with the current system date whenever a row in the Customers table is updated.

Scenario 2: Log Transactions

SQL

CREATE OR REPLACE TRIGGER LogTransaction

AFTER INSERT ON Transactions

FOR EACH ROW

BEGIN

INSERT INTO AuditLog (TransactionID, Action, Timestamp)

VALUES (:NEW.TransactionID, 'Insert', SYSDATE);

END;

/

Use code [with caution.](/faq#coding)

This trigger inserts a record into the AuditLog table for each transaction inserted into the Transactions table. The AuditLog table should have columns for TransactionID, Action (e.g., 'Insert', 'Update', 'Delete'), and Timestamp.

Scenario 3: Enforce Transaction Rules

SQL

CREATE OR REPLACE TRIGGER CheckTransactionRules

BEFORE INSERT ON Transactions

FOR EACH ROW

BEGIN

IF :NEW.TransactionType = 'Withdrawal' THEN

SELECT balance INTO v\_balance FROM accounts WHERE account\_id = :NEW.AccountID;

IF :NEW.Amount > v\_balance THEN

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

END IF;

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

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

END IF;

END;

/

**Exercise 6: Cursors**

**Scenario 1: Generate Monthly Statements**

SQL

DECLARE

CURSOR cust\_cursor IS

SELECT customer\_id

FROM customers;

cust\_rec cust\_cursor%ROWTYPE;

v\_transaction\_date DATE := TRUNC(SYSDATE, 'MM');

BEGIN

OPEN cust\_cursor;

LOOP

FETCH cust\_cursor INTO cust\_rec;

EXIT WHEN cust\_cursor%NOTFOUND;

-- Logic to generate statement for cust\_rec.customer\_id

-- using transactions for v\_transaction\_date

END LOOP;

CLOSE cust\_cursor;

END;

/

Use code [with caution.](/faq#coding)

**Note:**

* This code outline provides a basic structure. You'll need to implement the logic to fetch transactions for the current month, format the statement, and potentially send it to the customer.
* Consider using dynamic SQL or stored procedures to encapsulate the statement generation logic.

**Scenario 2: Apply Annual Fee**

SQL

DECLARE

CURSOR account\_cursor IS

SELECT account\_id, balance

FROM accounts;

account\_rec account\_cursor%ROWTYPE;

v\_annual\_fee NUMBER := 100; -- Replace with actual fee

BEGIN

OPEN account\_cursor;

LOOP

FETCH account\_cursor INTO account\_rec;

EXIT WHEN account\_cursor%NOTFOUND;

UPDATE accounts

SET balance = balance - v\_annual\_fee

WHERE account\_id = account\_rec.account\_id;

END LOOP;

CLOSE account\_cursor;

END;

/

Use code [with caution.](/faq#coding)

**Scenario 3: Update Loan Interest Rates**

SQL

DECLARE

CURSOR loan\_cursor IS

SELECT loan\_id, interest\_rate

FROM loans;

loan\_rec loan\_cursor%ROWTYPE;

v\_new\_interest\_rate NUMBER := 5; -- Replace with new interest rate

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO loan\_rec;

EXIT WHEN loan\_cursor%NOTFOUND;

UPDATE loans

SET interest\_rate = v\_new\_interest\_rate

WHERE loan\_id = loan\_rec.loan\_id;

END LOOP;

CLOSE loan\_cursor;

END;

/

**Exercise 7: Packages**

Scenario 1: Customer Management Package

SQL

CREATE OR REPLACE PACKAGE CustomerManagement AS

PROCEDURE AddCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

-- other customer details

);

PROCEDURE UpdateCustomer (

p\_customer\_id IN NUMBER,

-- updated customer details

);

FUNCTION GetCustomerBalance (

p\_customer\_id IN NUMBER

) RETURN NUMBER;

END CustomerManagement;

/

CREATE OR REPLACE PACKAGE BODY CustomerManagement AS

PROCEDURE AddCustomer (

p\_customer\_id IN NUMBER,

p\_name IN VARCHAR2,

-- other customer details

) IS

BEGIN

-- Insert new customer

END AddCustomer;

PROCEDURE UpdateCustomer (

p\_customer\_id IN NUMBER,

-- updated customer details

) IS

BEGIN

-- Update customer details

END UpdateCustomer;

FUNCTION GetCustomerBalance (

p\_customer\_id IN NUMBER

) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

-- Calculate customer balance

RETURN v\_balance;

END GetCustomerBalance;

END CustomerManagement;

/

Use code [with caution.](/faq#coding)

Scenario 2: Employee Management Package

SQL

CREATE OR REPLACE PACKAGE EmployeeManagement AS

PROCEDURE HireEmployee (

-- employee details

);

PROCEDURE UpdateEmployee (

p\_employee\_id IN NUMBER,

-- updated employee details

);

FUNCTION CalculateAnnualSalary (

p\_employee\_id IN NUMBER

) RETURN NUMBER;

END EmployeeManagement;

/

CREATE OR REPLACE PACKAGE BODY EmployeeManagement AS

PROCEDURE HireEmployee (

-- employee details

) IS

BEGIN

-- Insert new employee

END HireEmployee;

PROCEDURE UpdateEmployee (

p\_employee\_id IN NUMBER,

-- updated employee details

) IS

BEGIN

-- Update employee details

END UpdateEmployee;

FUNCTION CalculateAnnualSalary (

p\_employee\_id IN NUMBER

) RETURN NUMBER IS

v\_annual\_salary NUMBER;

BEGIN

-- Calculate annual salary

RETURN v\_annual\_salary;

END CalculateAnnualSalary;

END EmployeeManagement;

/

Use code [with caution.](/faq#coding)

Scenario 3: Account Operations Package

SQL

CREATE OR REPLACE PACKAGE AccountOperations AS

PROCEDURE OpenAccount (

-- account details

);

PROCEDURE CloseAccount (

p\_account\_id IN NUMBER

);

FUNCTION GetCustomerTotalBalance (

p\_customer\_id IN NUMBER

) RETURN NUMBER;

END AccountOperations;

/

CREATE OR REPLACE PACKAGE BODY AccountOperations AS

PROCEDURE OpenAccount (

-- account details

) IS

BEGIN

-- Insert new account

END OpenAccount;

PROCEDURE CloseAccount (

p\_account\_id IN NUMBER

) IS

BEGIN

-- Close account

END CloseAccount;

FUNCTION GetCustomerTotalBalance (

p\_customer\_id IN NUMBER

) RETURN NUMBER IS

v\_total\_balance NUMBER;

BEGIN

-- Calculate total balance for customer

RETURN v\_total\_balance;

END GetCustomerTotalBalance;

END AccountOperations;

/