

오라클데이터베이스 PL/SQL

I. PL/SQL 개요

일반 프로그래밍언어 처럼 각 DBMS 벤더(Vendor) 마다 PL(Procedural Language)/SQL(=Structural Query Language), SQL/PL(=DB2), T-SQL(=SQL Server)등의 절차형 SQL 을 제공합니다.

PL/SQL : Procedural Language extension to SQL (SQL을 확장한 순차적/절차적 처리언어) 은 SQL과 일반 프로그래밍 언어의 특성을 결합한 것으로 볼 수 있고, PL/SQL Runtime Engine의 Procedural Statement Executor에 의해서 해석되고 실행됩니다.

Oracle의 PL/SQL은 Block 구조로 되어있고 Block 내에는 DML 문장과 QUERY 문장, 그리고 절차형 언어(IF, LOOP) 등을 사용할 수 있으며, 절차적 프로그래밍을 가능하게 하는 트랜잭션 언어로 이런 PL/SQL을 이용하여 다양한 저장 모듈(Stored Module)을 개발할 수 있습니다.



저장 모듈이란, PL/SQL 문장을 데이터베이스 서버에 저장하여 사용자와 애플리케이션 사이에서 공유할 수 있도록 만든 일종의 SQL 컴포넌트 프로그램이며, 독립적으로 실행되거나 다른 프로그램으로부터 실행될 수 있는 완전한 실행 프로그램이다. Oracle의 저장 모듈에는 Procedure(=프로시저), User Defined Function(=함수), Trigger(=트리거)가 있습니다.

PL/SQL을 사용함으로써 얻는 장점은 간략하게 정리하면 아래와 같습니다.

- BEGIN ~ END와 같은 블록 구조로 여러 SQL 구문이 한꺼번에 서버로 전송되므로 명령 처리속도가 빠르고 통신량도 줄어들게 됩니다.
- 블록 내에 또 다른 명령 블록을 포함하여 모듈화 할 수 있습니다.
- 여러 형태의 변수/동적변, 타입(컬렉션, 레코드, 복합 데이터 타입)을 선언하여 사용할 수 있습니다.
- 조건문, 반복문을 사용하여 절차적인 프로그래밍이 가능하 예외처리(Exception)등을 할 수 있습니다.

II. PL/SQL 기본구조

1. 익명 블록



PL/SQL의 기본 단위는 블록(Block) 입니다. 기본적으로 익명(=Anonymous) 블록으로 작성을 시작할 수 있습니다.

- ① 선언부 (Declarative Part) : 변수나 상수를 선언하는 곳, 필수 아님!
- ② 실행부(Executable Part) : 실제 프로그래밍 로직을 작성하는 곳, BEGIN~END 필수!
- ③ 예외처리부(Exception Part) : 실행부에서 발생할 오류들을 처리하는 곳

```
<< label >> (optional)
DECLARE    -- Declarative part (optional)
           -- Declarations of local types, variables, & subprograms

BEGIN      -- Executable part (required)
           -- Statements (which can use items declared in declarative part)

[EXCEPTION -- Exception-handling part (optional)
           -- Exception handlers for exceptions (errors) raised in executable part]

END;
```

PL/SQL은 오라클이 등장한 이래 시간이 지날수록 DBMS의 역할이 단순히 데이터처리에만 있지 않고 그 기능과 역할이 일반 프로그래밍 언어가 처리하는 수준에 이르게 되면서 추가된 기능으로, 최초 1989년 Oracle 6 부터 추가되어 지금도 계속 기능이 추가되고 있으며, 본 자료는 전체가 아닌 중요한 부분들 위주로 학습을 하는데 활용 하시기 바랍니다.

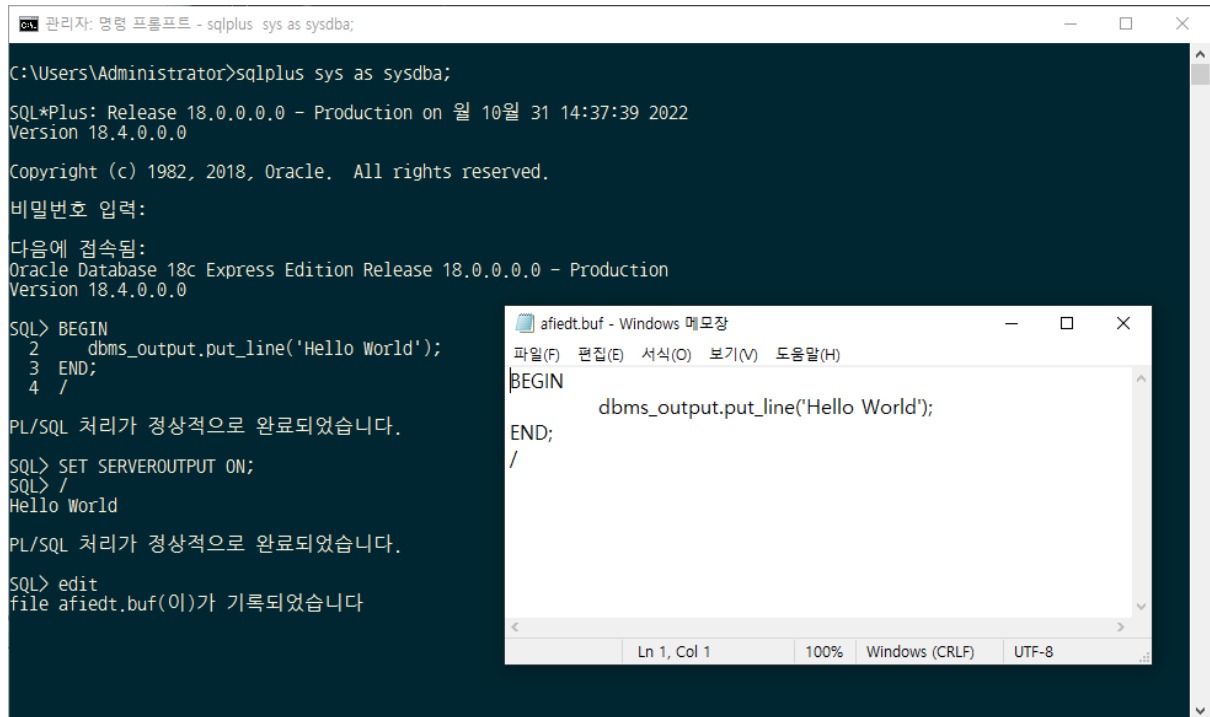
Database PL/SQL Language Reference

Oracle Database Database PL/SQL Language Reference, 21c

<https://docs.oracle.com/en/database/oracle/oracle-database/21/npls/>

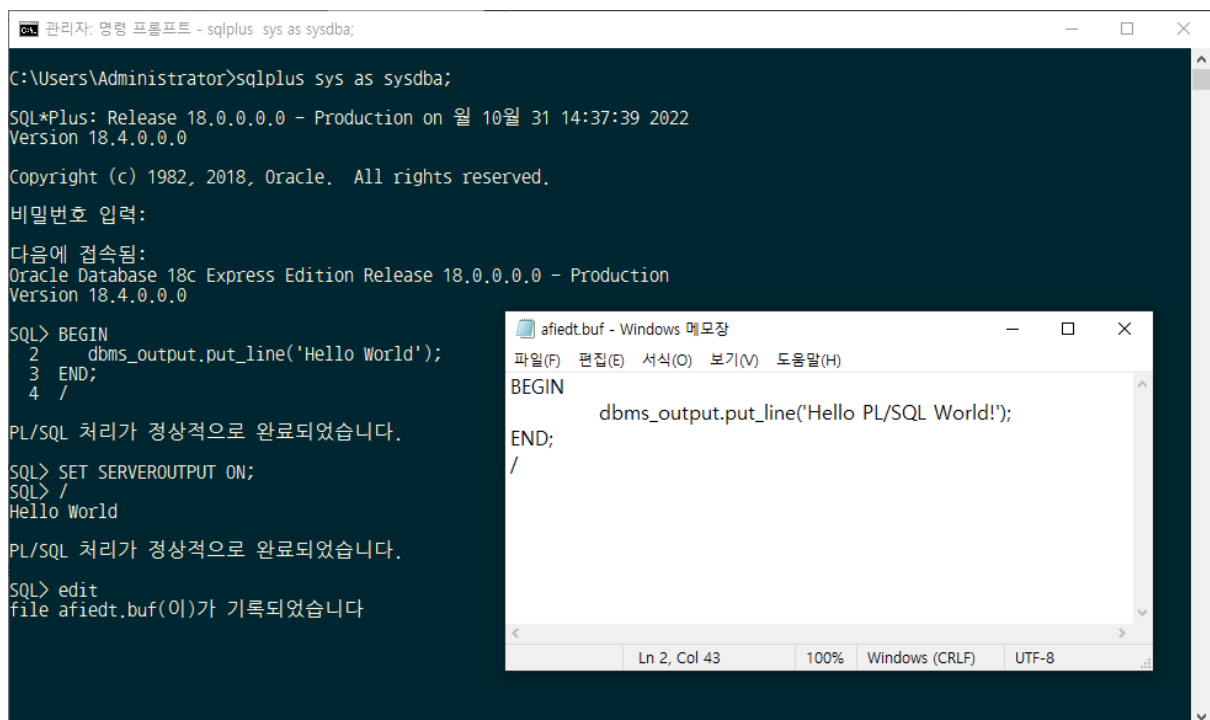
< 오라클 21c PL/SQL 메뉴얼 >

먼저, SQLPLUS로 PL/SQL 기본 문법을 작성해보도록 합니다. 명령 프롬프트로 먼저 실행합니다.



The screenshot shows a Windows command prompt window titled "관리자: 명령 프롬프트 - sqlplus sys as sysdba;". The user has entered the command `C:\Users\Administrator>sqlplus sys as sysdba;`. The output shows the SQL*Plus release information (18.0.0.0.0 - Production on 2022.10.31) and the Oracle Database 18c Express Edition version (18.4.0.0.0). The user is prompted for a password and then connected. The user enters the command `SQL> BEGIN`, followed by `2 dbms_output.put_line('Hello World');`, `3 END;`, and `4 /`. The output shows "PL/SQL 처리가 정상적으로 완료되었습니다." (PL/SQL processing completed successfully). The user then enters `SQL> SET SERVEROUTPUT ON;`, `SQL> /`, and the output shows "Hello World". The user then enters `SQL> edit`, and the output shows "file afiedt.buf(이)가 기록되었습니다" (file afiedt.buf(이) has been saved). A Notepad window titled "afiedt.buf - Windows 메모장" is also shown, containing the PL/SQL script: `BEGIN`, `dbms_output.put_line('Hello World');`, `END;`, and `/`.

< SQLPLUS로 작성한 PL/SQL 예시 >



The screenshot shows the same Windows command prompt window as before. The user has entered the command `C:\Users\Administrator>sqlplus sys as sysdba;`. The output shows the SQL*Plus release information (18.0.0.0.0 - Production on 2022.10.31) and the Oracle Database 18c Express Edition version (18.4.0.0.0). The user is prompted for a password and then connected. The user enters the command `SQL> BEGIN`, followed by `2 dbms_output.put_line('Hello World');`, `3 END;`, and `4 /`. The output shows "PL/SQL 처리가 정상적으로 완료되었습니다." (PL/SQL processing completed successfully). The user then enters `SQL> SET SERVEROUTPUT ON;`, `SQL> /`, and the output shows "Hello World". The user then enters `SQL> edit`, and the output shows "file afiedt.buf(이)가 기록되었습니다" (file afiedt.buf(이) has been saved). A Notepad window titled "afiedt.buf - Windows 메모장" is also shown, containing the modified PL/SQL script: `BEGIN`, `dbms_output.put_line('Hello PL/SQL World!');`, `END;`, and `/`.

< 수정하고 다시 실행 예시 >

```
SQL> edit
file afiedt.buf(이)가 기록되었습니다
1 BEGIN
2   dbms_output.put_line('Hello PL/SQL World!');
3* END;

SQL> /
Hello PL/SQL World!

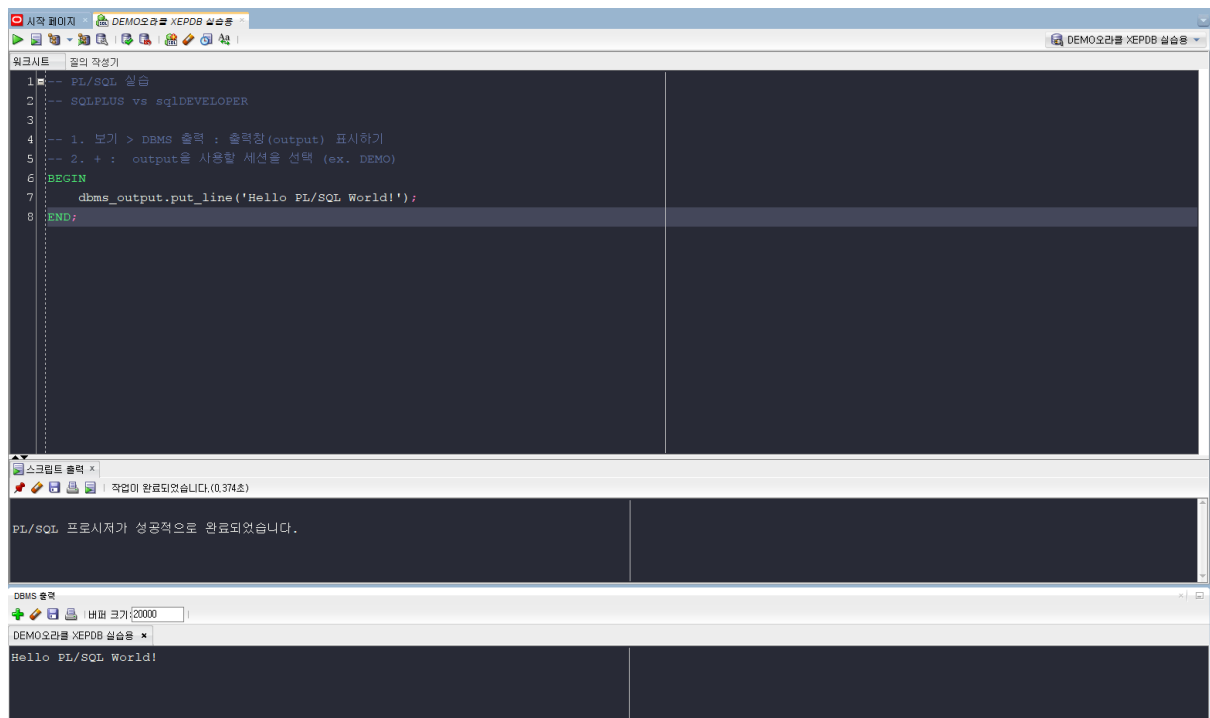
PL/SQL 처리가 정상적으로 완료되었습니다.

SQL>
```

< 수정후 재실행 예시 >

- 1) 시작 - CMD - sqlplus sys as sysdba
- 2) SET SERVEROUTPUT ON;
- 3) PL/SQL 기본문법 작성
- 4) (재)실행 : /

다만, sqlDeveloper를 이용해서 작성할 예정이므로, sqlDeveloper를 실행하고 메뉴중 [보기 > DBMS출력] 창을 실행해서 output을 확인하면서 실습해 나가도록 하겠습니다.



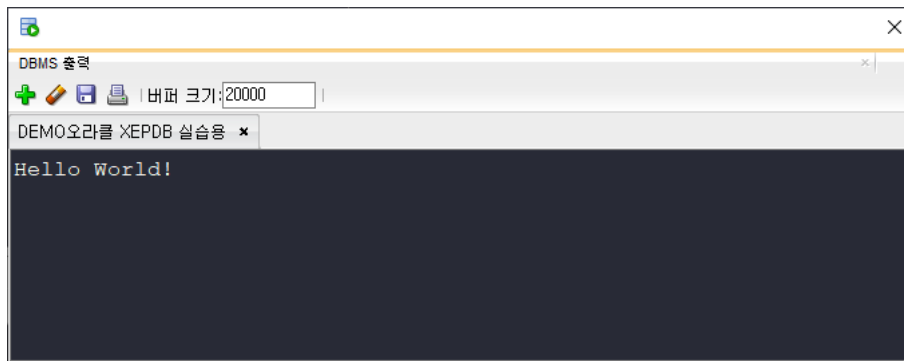
< sqlDeveloper >



dbms_output 패키지는 PL/SQL 블록, 하위 프로그램, 패키지 및 트리거가 출력을 표시하도록 합니다. 특히 PL/SQL 디버깅 정보를 표시하는 데 사용됩니다.

이번에는 선언부(Declarative Part)를 이용해 변수, 상수를 사용한 예제를 작성해봅니다.

```
DECLARE
  l_message VARCHAR2(255) := 'Hello World!'; -- local variable
BEGIN
  dbms_output.put_line(l_message);
END;
```



< output >

PL/SQL은 선언부에서 실행부에서 사용할 변수를 선언하고 초기화할 수 있습니다.



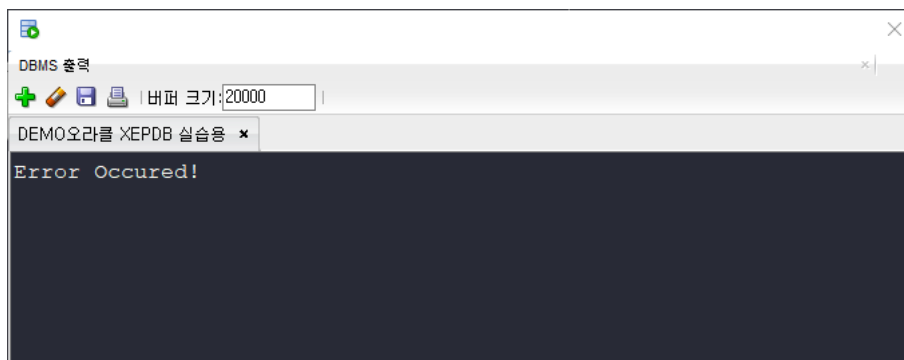
변수의 선언시 일반 프로그래밍언어와 다르게 변수명 := 값 형태로 작성합니다.

```
DECLARE
  part_number      NUMBER(6);      -- SQL data type
  part_name        VARCHAR2(20);   -- SQL data type
  in_stock         BOOLEAN;        -- PL/SQL-only data type
  part_price       NUMBER(6,2);    -- SQL data type
  part_description VARCHAR2(50);   -- SQL data type
BEGIN
  NULL;
END;
-- 실행결과는 아무것도 출력되지 않습니다.
```

이번에는 예외 처리의 예시를 살펴보겠습니다. 기본적인 예외 처리는 PL/SQL 파서가 기본적인 예외처리를 합니다. 다만, 사용자가 별도의 예외처리를 하기 위한 부분으로 EXCEPTION 으로 작성합니다.

예외가 발생할 경우, Error Occured! 라는 메시지를 출력하는 예시입니다.

```
DECLARE
  l_counter NUMBER := 1;
BEGIN
  l_counter := l_counter / 0; -- zero divide
  dbms_output.put_line(l_counter);
  EXCEPTION WHEN OTHERS THEN
    dbms_output.put_line('Error Occured!');
END;
```



< 예외 처리 예시 >

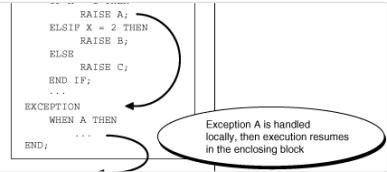
예외 처리 구문은 아래와 같습니다.

```
EXCEPTION WHEN 예외1 THEN 예외처리1
          WHEN 예외2 THEN 예외처리2
          ...
          WHEN OTHERS THEN 예외처리
-- 예외의 종류 : 사용자 정의 또는 사전 정의된(Predefined Exception) 예외가 있습니다.
```

Database PL/SQL Language Reference

The script content on this page is for navigation purposes only and does not alter the content in any way. This chapter explains how to handle PL/SQL compile-time warnings and PL/SQL runtime errors. The latter are called exceptions. Tip: If you have problems creating or running PL/SQL code, check the Oracle

<https://docs.oracle.com/en/database/oracle/oracle-database/21/lnpls/plsql-error-handling.html>

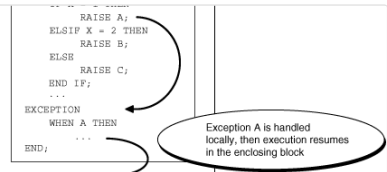


< Exception 에러처리 >

Database PL/SQL Language Reference

The script content on this page is for navigation purposes only and does not alter the content in any way. This chapter explains how to handle PL/SQL compile-time warnings and PL/SQL runtime errors. The latter are called exceptions. Tip: If you have problems creating or running PL/SQL code, check the Oracle

<https://docs.oracle.com/en/database/oracle/oracle-database/21/lnpls/plsql-error-handling.html#GUID-8844A6D8-FE6F-4DFF-B449-59AB076316C1>

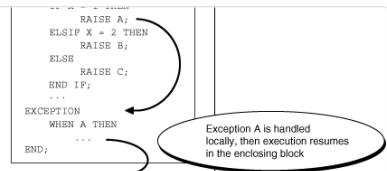


< 오라클 Userdefined Exception >

Database PL/SQL Language Reference

The script content on this page is for navigation purposes only and does not alter the content in any way. This chapter explains how to handle PL/SQL compile-time warnings and PL/SQL runtime errors. The latter are called exceptions. Tip: If you have problems creating or running PL/SQL code, check the Oracle

<https://docs.oracle.com/en/database/oracle/oracle-database/21/lnpls/plsql-error-handling.html#GUID-8C327B4A-71FA-4CFB-8BC9-4550A23734D6>



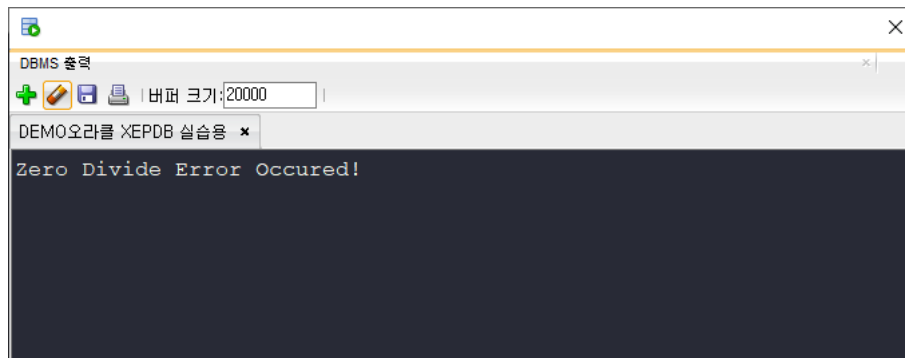
< 오라클 Predefined Exception >



ZERO_DIVIDE (젯수가 0일때 발생) EXCEPTION 처리로 위의 코드를 바꿔보세요!

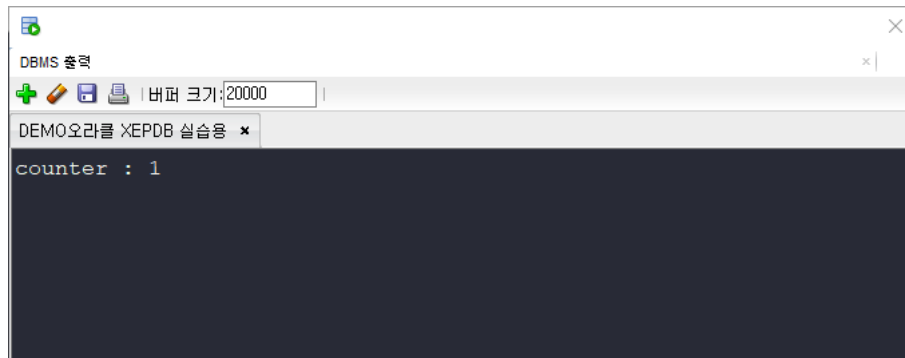
▼ ZERO_DIVIDE EXCEPTION 정답

```
-- 1.ZERO DIVIDE EXCEPTION 처리
DECLARE
  l_counter NUMBER := 1;
BEGIN
  l_counter := l_counter / 0; -- Divide by zero error encountered.
  dbms_output.put_line(l_counter);
  EXCEPTION WHEN ZERO_DIVIDE THEN
    dbms_output.put_line('Zero Divide Error Occured!');
END;
```



< 예외처리 1 >

```
-- 2. (또는) 1로 나누기 처리로 변경
DECLARE
    l_counter NUMBER := 1;
BEGIN
    l_counter := l_counter / 0; -- Divide by zero error encountered.
    dbms_output.put_line(l_counter);
    --EXCEPTION WHEN OTHERS THEN
    EXCEPTION WHEN ZERO_DIVIDE THEN
        l_counter := l_counter / 1;
        dbms_output.put_line(CONCAT('result : ', l_counter));
END;
-- CONCAT(str1, str2) : 문자열 연결 함수
```



< 예외처리 2 >

오라클의 PL/SQL의 예외처리(Exception)는 Java와 같은 프로그래밍 언어에서 사용되는 try..catch..finally 문과 비슷한 형태입니다.

PL/SQL 예외처리	JAVA 예외처리
DECLARE ...변수나 상수 선언, 초기화 ... BEGIN 실행 부 EXCEPTION WHEN 예러1 THEN 처리1 WHEN 예 러2 THEN 처리2 ...계속... WHEN OTHERS THEN 처리 END;	TRY { 실행부 } CATCH (Exception 예러1) { 처리1} CATCH (Exception 예러2) { 처리2} FINALLY { ... 코 드 ... }

II. PL/SQL 구성요소

PL/SQL은 SQL을 사용하여 일반 프로그래밍 하듯이 순차적인 처리를 할 수 있게 해줍니다. 이에 추가적으로 제공되는 구성요소등에 대해
서 알아보기로 합니다.

1. 변수와 상수(Variables & Constants)

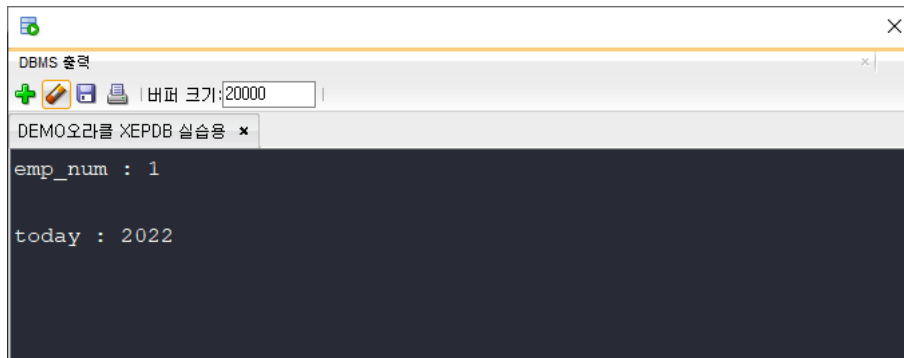
```
-- 변수의 선언
emp_num1 NUMBER(9);    -- 변수의 선언
grade CHAR(2);
emp_num2 INTEGER := 1; -- 변수의 선언 및 초기화

-- 상수의 선언 : 변수처럼 선언하되 상수는 반드시 값을 할당해야 함
nYear CONSTANT INTEGER := 2022;    -- (0)
nYear CONSTANT INTEGER;             -- (X)

-- 작성예시
-- 1-1. 변수
DECLARE
    emp_num1 NUMBER(9);
    grade CHAR(2);
    emp_num2 INTEGER := 1;
BEGIN
    dbms_output.put_line(CONCAT('emp_num : ', emp_num2));
END;

-- 1-2. 상수
DECLARE
```

```
nYear CONSTANT INTEGER := 2022; -- (0)
--nYear CONSTANT INTEGER; -- (X)
BEGIN
  dbms_output.put_line(CONCAT('today : ', nYear));
END;
```



PL/SQL이 일반 프로그래밍 언어와 다른점은 데이터베이스에 있는 테이블이나 뷰에 있는 데이터들을 대상으로 로직(Logic)을 처리한다는 점입니다.

Database PL/SQL Language Reference

Database PL/SQL Language Reference PL/SQL Predefined Data Types This appendix groups by data type family the data types and subtypes that the package STANDARD predefines. Constants This constant defines the maximum name length possible.

<https://docs.oracle.com/en/database/oracle/oracle-database/21/npls/plsql-predefined-data-types.html>

예를들어, PL/SQL에서 선언하는 변수는 대부분 테이블이 있는 컬럼값을 할당하는 용도로 사용되며 따라서 변수의 데이터 타입은 컬럼의 데이터 타입과 일치시킬 필요가 있는데, 만약 데이터 타입이 다르면 변수에 값을 할당할때 **VALUE_ERROR** 라는 예외가 발생합니다.

그런데, 변수를 선언할때 마다 일일이 테이블의 컬럼 데이터 타입을 확인해서 데이터 타입을 선언하는 일은 매우 번거롭고 실수할 수 있는데, 예를들면 HR 스키마의 **EMPLOYEES** 테이블의 **SALARY** 컬럼의 값을 받는 변수를 선언한다고 해봅시다.

```
-- 1. 직접 변수의 타입을 설정하는 경우를 예시
DECLARE
  nSalaries; -- 변수명을 정한다.
BEGIN
  ...로직...
END;

-- 2. 참조할 테이블의 타입을 자동으로 가져오는 예시
DECLARE
  nSalaries EMPLOYEES.SALARY%TYPE; -- EMPLOYEES의 SALARY 컬럼은 NUMBER(8, 2) 타입이다.
BEGIN
  ...로직...
END;
```



%ROWTYPE은 하나 이상의 값에 대해 적용합니다. 정확하게 이야기하면 로우타입 변수를 선언해서 테이블에 있는 로우를 대입할 수 있습니다. 다만, **%ROWTYPE**은 Collection 이나 OBJECT 타입 변수에서 사용할 수 있습니다.

2. 컬렉션

프로그래밍 언어에서는 숫자형, 문자형 등 기본적인 데이터 타입(Primitive Type) 뿐만 아니라, 기본 데이터 타입으로 구성된 배열을 사용할 수 있습니다. 배열(Array)이란 같은 데이터 타입으로 구성된 요소들의 집합으로, 언어마다 약간씩 다르지만 보통 배열은 **[]** 기호를 사용합니다.

PL/SQL 에서도 배열 형태의 데이터 타입을 지원하는데 이것들을 보통 컬렉션(Collection) 이라 부릅니다.



컬렉션은 같은 데이터 타입을 가진 요소들로 구성됩니다.

2-1. 컬렉션의 종류

(1) varray : variable array의 약자로, 고정 길이(fixed number)를 가진 배열을 뜻합니다. 즉, varray 선언시 배열 전체의 크기를 명시해야 합니다. 이 말의 의미는 테이블 하나의 컬럼 타입으로 varray가 사용될 수 있음을 뜻합니다. 요소들의 데이터 참조시 순서를 지켜야 합니다. varray[1], varray[2],...

(2) 중첩 테이블 : 중첩 테이블(nested table) 역시 컬렉션의 한 종류로 varray와 흡사한 구조를 갖습니다. 하지만 varray와 다르게 선언시 전체 크기를 명시할 필요는 없으며, 요소들의 데이터 참조시 순서를 지킬 필요가 없습니다.



중첩 테이블은 데이터의 크기와 갯수에 따라 동적으로 늘어나게 되어 있습니다.

(3) Associative array(index-by table) : 연관배열(associative array)은 키-값 쌍으로 구성된 컬렉션으로, 하나의 키는 하나의 값과 연관되어 있습니다.

JAVA, C#과 같은 언어에서 사용되는 해시테이블(Hash Table)과 동일한 개념으로, varray가 요소의 인덱스를 통해 각 요소의 값에 접근한다면 associative array는 키 값에 의해 접근합니다.

Database PL/SQL Language Reference

Database PL/SQL Language Reference PL/SQL Collections and Records PL/SQL lets you define two kinds of composite data types: collection and record. A composite data type stores values that have internal components. You can pass entire composite variables to subprograms as parameters, and you can access

[https://docs.oracle.com/en/database/oracle/oracle-database/21/npls/plsql-collections-and-records.html#](https://docs.oracle.com/en/database/oracle/oracle-database/21/npls/plsql-collections-and-records.html#GUID-7E9034D5-0D33-43A1-9012-918350FE148C)

GUID-7E9034D5-0D33-43A1-9012-918350FE148C

Integers

17	99	407	83	622	105	19	67	278
----	----	-----	----	-----	-----	----	----	-----

x(2)	x(3)	x(4)	x(5)	x(6)	x(7)	x(8)	x(9)	x(10)
------	------	------	------	------	------	------	------	-------

Fixed Upper Bound

Upper of index type

< 오라클 Collection Type >

2-2. 컬렉션의 사용

컬렉션을 사용하려면 TYPE이라는 키워드를 사용해 종류별 컬렉션을 정의하고 사용합니다. 일반적인 프로그래밍 언어에서도 배열을 선언하고 사용하듯, 오라클에서도 컬렉션을 정의하고 생성한 뒤 이를 변수로 선언해서 사용합니다.

```
-- varray 선언
TYPE 타입변수명 IS varray (크기) OF 요소 데이터 타입 [NOT NULL];

-- nested table 선언
TYPE 타입변수명 IS TABLE OF 요소 데이터 타입 [NOT NULL];

-- associative array 선언
TYPE 타입변수명 IS TABLE OF 요소 데이터 타입 [NOT NULL]
INDEX BY [PLS_INTEGER | BINARY_INTEGER | VARCHAR2(크기)]
INDEX BY 키타입;
```

그러면, 실제 컬렉션 타입의 예시를 살펴봅시다.

```
DECLARE
TYPE varray_test IS VARRAY(3) OF INTEGER; -- INTEGER형 요소 3개로 구성된 varray
TYPE nested_test IS TABLE OF VARCHAR2(10); -- VARCHAR2(10) 형으로 구성된 nested_table
TYPE assoc_num_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER; -- NUMBER형 요소들로 구성
TYPE assoc_array_str_type IS TABLE OF VARCHAR2(32) INDEX BY PLS_INTEGER
```


3. 레코드

컬렉션은 배열 형태의 데이터타입, 레코드는 테이블 형태의 데이터 타입입니다. 특히 레코드는 테이블의 컬럼처럼 여러개의 필드로 구성되어 있고, PL/SQL 블록에서 임시로 사용할 수 있는 데이터 타입중 하나라 할 수 있습니다.

컬렉션은 배열의 형태를 가지는데, 이것을 구성하는 요소들의 데이터 타입은 모두 같아야 하지만 레코드는 각기 다른 유형의 데이터 타입을 가질 수 있습니다.

레코드는 테이블의 형태를 갖으므로 실제 대부분의 경우 테이블의 데이터를 읽어오거나 조작하기 위해 PL/SQL 블록 내에서 임시적인 데이터를 저장하는 역할을 수행합니다.

```
-- 레코드 형식
-- 레코드에서는 요소라는 말 대신 필드라는 용어를 사용함. 여기서 요소란 레코드를 구성하는 각각의 데이터를 뜻함
TYPE 레코드이름 IS RECORD ( 필드1 데이터타입1, 필드2 데이터타입2, ... );

-- 또한 변수 %TYPE을 사용하여 선언할 수 있는 것과 같이 %ROWTYPE을 사용하여 선언과 동시에
-- 특정 값으로 초기화할 수 있다.

레코드이름 테이블명%ROWTYPE;

-- %ROWTYPE을 사용하여 레코드를 선언할 경우 레코드를 구성하는 각각의 필드는 테이블의 컬럼 이름과
-- 데이터 타입을 그대로 사용하게 된다.

-- 또한 PL/SQL 블록에서 CURSOR를 사용할 경우 커서가 반환하는 값들을 레코드에 할당할 수 있는데
-- 이러한 경우에도 %ROWTYPE을 사용한다.

레코드이름 커서명%ROWTYPE;

-- 레코드 예시

DECLARE
    -- TYPE을 선언한 레코드
    TYPE record1 IS RECORD ( dept_id NUMBER NOT NULL := 300,
                             dept_name VARCHAR2(30),
                             man_id NUMBER, loc_id NUMBER );
    -- 위에서 선언한 record1을 받는 변수
    rec1 record1;

    -- 테이블명%ROWTYPE으로 선언한 레코드
    rec2 departments%ROWTYPE;

    CURSOR C1 IS
        SELECT department_id, department_name, location_id
        FROM departments
        WHERE location_id = 1700;

    -- 커서명%ROWTYPE을 이용한 레코드
    rec3 C1%ROWTYPE;

BEGIN
    -- record1 레코드변수 rec1의 dept_name 필드에 값 할당
    rec1.dept_name := '레코드 부서1';

    -- rec2 변수에 값 할당
    rec2.department_id := 400;
    rec2.department_name := '레코드 부서2';
    rec2.location_id := 2700;

    --rec1 레코드값을 departments 테이블에 INSERT
    INSERT INTO departments VALUES rec1;

    --rec2 레코드값을 departments 테이블에 INSERT
    INSERT INTO departments VALUES rec2;

    -- 커서 오픈
    OPEN C1;
    LOOP
        FETCH C1 INTO rec3;
        DBMS_OUTPUT.PUT_LINE(rec3.department_id);
        EXIT WHEN C1%NOTFOUND;
    END LOOP;
    COMMIT;
    EXCEPTION WHEN OTHERS THEN
        ROLLBACK;
END;

SELECT * FROM departments;
```



컬렉션이나 레코드는 그 자체로 사용하기보다 함수나 프로시저, 패키지에서 로직을 처리하기 위해 사용합니다.

III. PL/SQL 문장과 커서

PL/SQL에서는 IF문, CASE문, FOR문들을 사용할 수 있습니다.

1. IF문

일반 프로그래밍 언어처럼 IF문을 사용할 수 있습니다. 약간의 차이는 있는데, 기본 형식은 아래와 같습니다.

```
-- 처리 조건이 하나일때
IF 조건 THEN
    처리할 구문;
END IF;

-- 처리 조건이 둘일때
IF 조건 THEN
    처리할 구문1;
ELSE
    처리할 구문2;
END IF;

-- 처리 조건이 여러개일때
IF 조건1 THEN
    처리할 구문1;
ELSIF 조건2 THEN
    처리할 구문2;
...
ELSE
    처리문n
END IF;

-- IF문 예시
DECLARE
    grade CHAR(1);
BEGIN
    grade := 'B';
    IF grade = 'A' THEN
        dbms_output.put_line('Your grade is A');
    ELSIF grade = 'B' THEN
        dbms_output.put_line('Your grade is B');
    ELSIF grade = 'C' THEN
        dbms_output.put_line('Your grade is C');
    ELSE
        dbms_output.put_line('Your grade is not A, B, C');
    END IF;
END;
```

2. CASE 문

SQL의 CASE와 비슷한데, 마지막 END 대신 END CASE를 작성합니다.

```
-- CASE문 예시 : simple CASE
DECLARE
    grade CHAR(1);
BEGIN
    grade := 'B';

    CASE grade
        WHEN 'A' THEN
            dbms_output.put_line('A is Excellent!');
        WHEN 'B' THEN
            dbms_output.put_line('B is Good!');
        WHEN 'C' THEN
            dbms_output.put_line('C is Fair!');
        ELSE
            dbms_output.put_line('Not Found!');
    END CASE;
```

```
END;

-- CASE문 예시 : Searched CASE
```

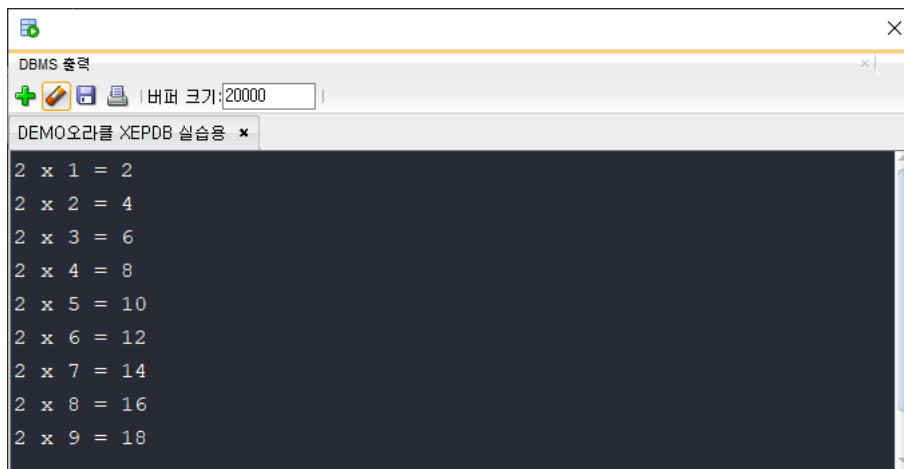
3. LOOP

PL/SQL의 반복문은 3가지가 존재하는데, 그중 간단한 형태인 LOOP를 살펴봅시다.

```
-- 1.LOOP문 기본형식
LOOP
  처리할 문장들...
END LOOP;

-- 처리할 문장만 작성하면 무한 루프!
-- LOOP를 중지하고 빠져 나오는 문장에 EXIT;를 기술하여 처리
-- 구구단을 출력해봅시다.

-- 2.LOOP문 예시(2단 구구단)
DECLARE
  limit_number INTEGER;
  result_number INTEGER;
BEGIN
  limit_number := 1;
  LOOP
    result_number := 2 * limit_number;
    IF limit_number > 9 THEN
      EXIT;
    ELSE
      dbms_output.put_line('2 x ' || limit_number || ' = ' || result_number);
    END IF;
    limit_number := limit_number + 1;
  END LOOP;
END;
```



4. WHILE-LOOP문

WHILE LOOP는 다른 프로그래밍 언어에서도 지원되는 반복문입니다.

```
-- WHILE-LOOP문
WHILE 조건 LOOP
  처리할 문장들..
END LOOP;

-- 조건이 만족하는 동안만 반복처리
-- EXIT WHEN 으로 루프를 빠져나갑니다.
-- 2단 구구단을 변경해봅시다.

DECLARE
  limit_number INTEGER;
  result_number INTEGER;
BEGIN
  limit_number := 1;
  WHILE limit_number <= 9 LOOP
    result_number := 2 * limit_number;
    dbms_output.put_line('2 x ' || limit_number || ' = ' || result_number);
  END LOOP;
```

```

        limit_number := limit_number + 1;
    END LOOP;
END;

```

```

DEMO오라클 XEPDB 실험용 x
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18

```

4. FOR-LOOP문

FOR LOOP는 아래와 같습니다.

```

-- FOR-LOOP 형식
FOR 카운터 IN [REVERSE] 최소값..최대값 LOOP
    처리 문장들;
END LOOP;

-- LOOP or REVERSE-LOOP

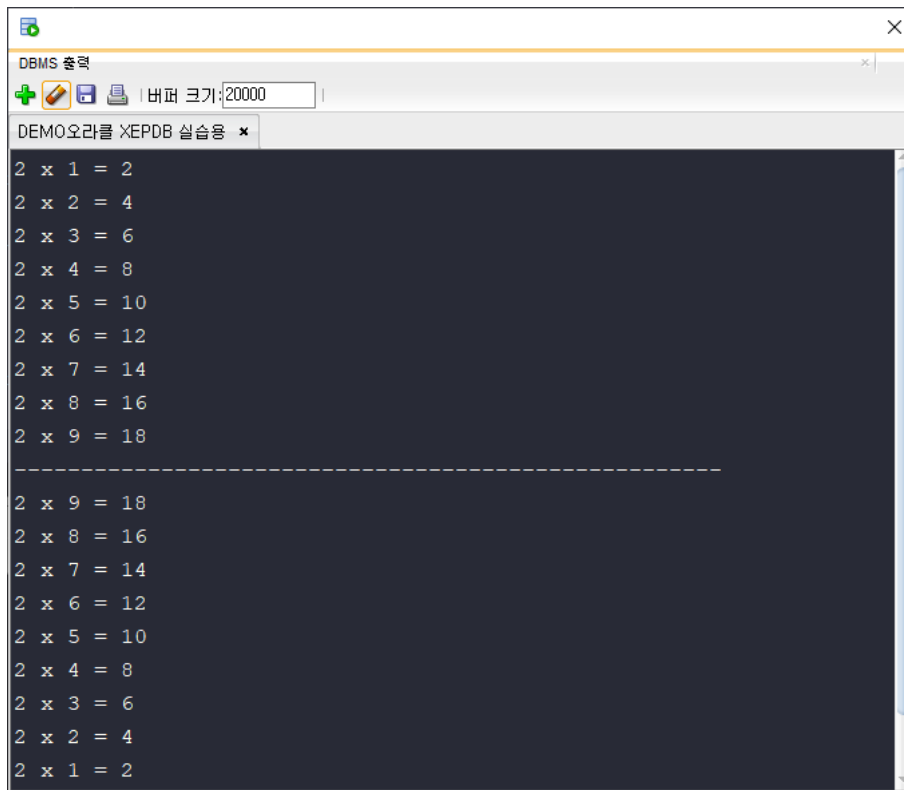
DECLARE
    limit_number INTEGER;
    result_number INTEGER;
BEGIN
    limit_number := 1;
    result_number := 0;
    << FIRST >>
    FOR limit_number IN 1..9 LOOP
        result_number := 2 * limit_number;
        dbms_output.put_line('2 x ' || limit_number || ' = ' || result_number);
    END LOOP;
    dbms_output.put_line('-----');
    << SECOND >>
    FOR limit_number IN REVERSE 1..9 LOOP
        result_number := 2 * limit_number;
        dbms_output.put_line('2 x ' || limit_number || ' = ' || result_number);
    END LOOP;
END;

-- 만약, 사용자가 정의한 변수와 FOR LOOP의 인덱스 변수가 같다면?
DECLARE
    i NUMBER := 5;
BEGIN
    FOR i IN 1..3 LOOP
        DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
    END LOOP;

    DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
END;

-- FOR LOOP 내부에서는 선언부의 i를 무시하고, LOOP 밖에서는 선언부 i를 참조함
--Inside loop, i is 1
--Inside loop, i is 2
--Inside loop, i is 3
--Outside loop, i is 5

```



5. GOTO 문

요즘은 GOTO 문이 없으므로, 생략합니다. 아래 북마크를 참고하세요! GOTO문은 오래전 COBOL, BASIC 같은 언어에서 사용했고 특정 라벨로 이동하는 구문입니다.

Database PL/SQL Language Reference

Database PL/SQL Language Reference PL/SQL Control Statements The IF THEN statement either runs or skips a sequence of one or more statements, depending on a condition. The IF THEN statement has this structure: IF condition THEN statements END IF; If the is true, the statements run; otherwise, the IF statement does nothing.

<https://docs.oracle.com/en/database/oracle/oracle-database/21/npls/plsql-control-statements.html#GUID-7C14CDED-B86F-4496-A1F6-8C3B67E3B032>

< 오라클 GOTO 분기문 >

6. NULL문

보통 NULL은 컬럼이나 변수의 값으로 사용되어 해당 컬럼값이나 변수값이 없음을 나타냅니다. PL/SQL에서 NULL도 이와 비슷하고, 아무런 처리를 하지 않음을 뜻합니다.

```
-- 이전의 IF문을 참고하세요
DECLARE
  message VARCHAR2(20) := 'Hello World!';
BEGIN
  IF LENGTH(message) > 0 THEN
    NULL;
  END IF;
END;

-- 주로 예외처리시 NULL을 사용합니다.
-- 즉, EXCEPTION 절에서 명시한 예외 이외의 오류 발생시 NULL 처리하기도 합니다.
```

7. CURSOR (커서)

SELECT 문장을 실행하면 조건에 따른 결과가 추출되는데, 이 결과는 한 건이 될수도 있고 여러건이 될수 있습니다. 이를 결과셋(Result Set) 혹은 결과집합 이라고도 합니다.

쿼리에 의해 반환되는 결과집합은 메모리상에 위치하는데 PL/SQL 에서는 커서(Cursor)를 사용하여 이 결과 집합에 접근할 수 있고, 커서를 사용해 결과셋의 각 개별 데이터에 접근할 수 있습니다.

```
-- 7-1. 명시적 커서
CURSOR 커서명 IS
    SELECT 문장;

OPEN 커서명; -- 커서 열기(=커서로 정의된 쿼리를 실행할 때)

FETCH 커서명 INTO 변수..; -- 쿼리의 결과에 접근, 보통 한개이상의 결과를 반환하므로 LOOP를 돌며 개별 값들에 접근하여 임의로 처리합니다.

CLOSE 커서명 : -- 사용된 커서를 닫습니다(=메모리에서 소멸)

-- BOARD 에서 제목을 가져오기 (DEMO 스키마)
-- BOARD 테이블과 샘플 데이터 확인!
DECLARE
    CURSOR BOARD_CURSOR IS
        SELECT TITLE
        FROM BOARD;

    BOARD_SEQ BOARD.TITLE%TYPE;
BEGIN
    OPEN BOARD_CURSOR;
    LOOP
        FETCH BOARD_CURSOR INTO BOARD_SEQ;
        EXIT WHEN BOARD_CURSOR%NOTFOUND;
        dbms_output.put_line(BOARD_SEQ);
    END LOOP;
    CLOSE BOARD_CURSOR;
END;

-- EMPLOYEES 에서 사번 가져오기 (HR 스키마)
-- EMPLOYEES 테이블과 샘플 데이터 확인!
DECLARE
    CURSOR emp_csr IS
        SELECT employee_id
        FROM employees
        WHERE department_id = 100; -- 6rows

    emp_id employees.employee_id%TYPE;
BEGIN
    OPEN emp_csr;
    LOOP
        FETCH emp_csr INTO emp_id;
        EXIT WHEN emp_csr%NOTFOUND;
        dbms_output.put_line('employee_id : ' || emp_id);
    END LOOP;
    CLOSE emp_csr;
END;

-- 7-2. 묵시적 커서
-- 오라클 내부에서 각각의 쿼리결과에 접근하여 사용하기 위한 커서로, 모든 쿼리가 실행될때마다
-- 오픈됩니다. 내부적 접근을 위한 커서로, 별도의 선언과 오픈을 할 필요가 없습니다.
-- 별도의 커서명이 없지만, 가장 최근 실행된 커서를 SQL 커서라고 하며, 이 이름으로 속성에
-- 접근합니다.

DECLARE
    count1 NUMBER;
    count2 NUMBER;
BEGIN
    SELECT COUNT(*)
    INTO count1
    FROM employees
    WHERE department_id = 100;

    count2 := SQL%ROWCOUNT;
    dbms_output.put_line('SELECT COUNT IS ' || count1);
    dbms_output.put_line('ROW COUNT IS ' || count2);
END;

DECLARE
    CURSOR dept_csr IS
        SELECT *
        FROM departments;

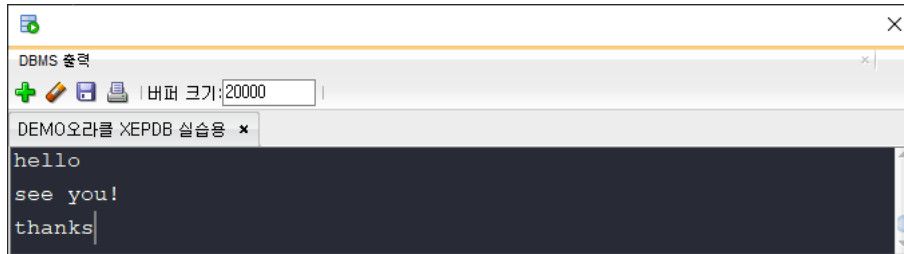
    dept_row departments%ROWTYPE;
BEGIN
    OPEN dept_csr;
    dbms_output.put_line('department_id | department_name | manager_id | location_id');
    LOOP
```

```

    FETCH dept_csr INTO dept_row;
    EXIT WHEN dept_csr%NOTFOUND;
    dbms_output.put_line(dept_row.department_id || ' | ' || dept_row.department_name || ' | ' || dept_row.manager_id || ' | ' || dept_row
    -- tab 추가하려면? 빈 공백문자나..LPAD(), RPAD() 사용
END LOOP;

CLOSE dept_csr;
END;

```



< Cursor 사용결과 >

%FOUND : 커서가 막 오픈된 상태에서는 NULL값을 반환합니다. OPEN 되고 첫 패치가 발생한 이후부터 TRUE 값을 반환하며 더이상 패치할 ROW가 없을 경우 FALSE를 반환합니다.

%ISOPEN : 커서가 오픈될 경우 TRUE를 반환합니다.

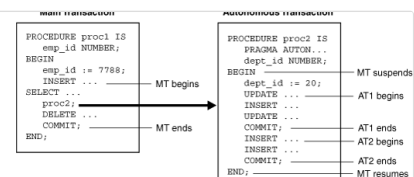
%NOTFOUND: %FOUND의 반대개념 입니다.

%ROWCOUNT : 카운터 역할을 하며, 커서가 막 오픈되면 0을 패치될때마다 1씩 증가합니다.

Database PL/SQL Language Reference

Static SQL is a PL/SQL feature that allows SQL syntax directly in a PL/SQL statement. This chapter describes static SQL and explains how to use it. Static SQL has the same syntax as SQL, except as noted.

<https://docs.oracle.com/en/database/oracle/oracle-database/21/lnpls/static-sql.html#GUID-F1FE15F9-5C96-4C4E-B240-B7363D25A8F1>



IV. PL/SQL 서브 프로그램

이전까지 학습한 PL/SQL은 익명 블록 즉 Anonymous block 입니다. 이것은 항상 컴파일을 한 뒤에 실행할 수 있습니다. 다만, 지금까지는 컴파일과 동시에 코드가 실행되어 그 결과를 반환했을 뿐입니다.

재사용을 위해서는 다시 컴파일과 수행을 해야하는 불편함이 있고, 이에 따라 오라클에서는 데이터베이스 객체로서 저장해놓고 필요할때 마다 호출하여 사용할 수 있도록 파라미터와 고유의 이름을 가진 PL/SQL 블록을 제공하는데 이것을 '서브프로그램'이라고 부릅니다.

1. 함수(function) : SQL에서의 함수는 보통 내장함수를 뜻하지만 PL/SQL에서는 사용자 정의 함수를 뜻합니다. 오라클에서 제공하는 함수는 SQL 함수, 사용자가 직접 작성하면 사용자 정의 함수라 할 수라고 할 수 있습니다.

```

-- (사용자 정의) 함수 선언
-- 별도로 DECLARE가 없고 CREATE OR REPLACE FUNCTION이 그것을 대신함
CREATE OR REPLACE FUNCTION 함수명 (파라미터1 데이터타입, 파라미터2 데이터타입, ...)
RETURN 데이터 타입 IS [AS]
변수 선언..;
BEGIN
처리내용..;

RETURN 리턴값;
END;

-- 예시 : 사원을 입력받아 급여를 반환하는 사용자 정의 함수
-- emp_salaries(사번)
CREATE OR REPLACE FUNCTION emp_salaries (emp_id NUMBER)
RETURN NUMBER IS
nSalaries NUMBER(9);

```

```

BEGIN
    nSalaries := 0;
    SELECT salary
    INTO nSalaries
    FROM EMPLOYEES
    WHERE employee_id = emp_id;
    RETURN nSalaries;
END;

-- Function EMP_SALARIES이(가) 컴파일되었습니다 라는 메시지만 나올뿐...

-- 사용자 정의 함수를 활용해서, 테이블의 결과를 조회
-- 1. 사장인 Steven의 월급 조회
SELECT emp_salaries(100)
FROM dual;

-- 2. 부서번호 100번에 속하는 사원들의 정보와 급여 조회
SELECT employee_id, first_name, emp_salaries(employee_id) salary
FROM employees
WHERE department_id = 100;

-- Q. EMPLOYEES 테이블의 부서코드만 있는 경우 부서명을 추가로 알고자 할때?
-- Q1. JOIN을 사용한다 (또는 스칼라 서브쿼리)
SELECT e.employee_id, e.first_name, emp_salaries(employee_id) salary,
( SELECT department_name
  FROM departments d
  WHERE e.department_id = d.department_id ) dept_names
FROM employees e
WHERE department_id = 100;

-- Q2. 함수를 만들어서 사용한다 (직접 해보세요)
CREATE OR REPLACE FUNCTION get_dept_name(dept_id NUMBER)
RETURN VARCHAR2 IS
    sDeptName VARCHAR2(30);
BEGIN
    SELECT department_name
    INTO sDeptName
    FROM departments
    WHERE department_id = dept_id;

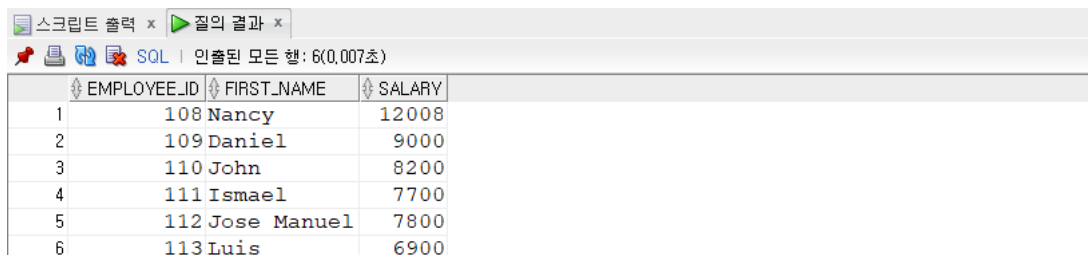
    RETURN sDeptName;
END;

-- Function GET_DEPT_NAME이(가) 컴파일되었습니다
-- 결과는 동일하지만, 함수를 사용하면 조인이 발생하지 않음
-- 부서 번호를 파라미터로 전달해 부서명을 리턴해줄 뿐
-- 비교하는 데이터 양이 적기때문에 성능차이가 거의 나지 않지만, 조인을 하는 경우 시스템에
-- 부하를 줄 수 있고, 이런경우 서브쿼리보다 함수가 낫다고 볼 수 있음
-- 그렇다고 꼭 함수가 좋다는것은 아니지만.. 필요한 만큼의 함수를 만들어 사용하는것이 좋다.
SELECT employee_id, first_name, get_dept_name(department_id) dept_names
FROM employees
WHERE department_id = 100;

CREATE OR REPLACE FUNCTION get_bonus(emp_id NUMBER)
RETURN NUMBER IS
    nBonus NUMBER;
BEGIN
    SELECT salary * (1 + NVL(commission_pct, 0))
    INTO nBonus
    FROM employees
    WHERE employee_id = emp_id;
    RETURN nBonus;
END;

-- employees 148번 사원, 11000 salary, 0.3 commission_pct ==> 14300 bonus

```



EMPLOYEE_ID	FIRST_NAME	SALARY
108	Nancy	12008
109	Daniel	9000
110	John	8200
111	Ismael	7700
112	Jose Manuel	7800
113	Luis	6900

< 사용자 정의 함수 실행 및 결과조회 >

스크립트 출력 x 질의 결과 x			
SQL 인출된 모든 행: 6(0,004초)			
	EMPLOYEE_ID	FIRST_NAME	DEPT_NAME
1	108	Nancy	Finance
2	109	Daniel	Finance
3	110	John	Finance
4	111	Ismael	Finance
5	112	Jose Manuel	Finance
6	113	Luis	Finance

익명 블록(Anonymous Block)에서는 단순한 로직을 처리했으나, emp_salaries 함수를 정의하고 그 결과로 리턴값을 반환받아 출력해 보았습니다.

INTO 키워드는 SELECT 리스트에 명시된 임의의 컬럼값을 선택해서 그 값을 변수에 할당하기 위한 것으로 지금까지 보지 못했던 키워드입니다.

2. 프로시저(또는 내장프로시저, stored procedure)



함수와 프로시저의 큰 차이는 반환값 유무입니다. 함수와 달리 프로시저는 값을 반환하지 않습니다.

```
-- 프로시저 선언
-- 별도로 DECLARE가 없고 CREATE OR REPLACE PROCEDURE가 그것을 대신함
-- 파라미터를 명시할때 구체적인 자릿수를 표기하지 않아도 됨 (되려, 표기하면 오류가 발생)
CREATE OR REPLACE PROCEDURE 프로시저명 (파라미터1 데이터타입, 파라미터2 데이터타입, ...)
IS [AS]
변수 선언부...;
BEGIN
프로시저 본문처리...;
EXCEPTION
예외처리부...;
END;
```

-- 파라미터는 IN, OUT, INOUT 파라미터가 있음
-- IN: 호출하는 곳에서 함수/프로시저로 값 전달 (=기본값)
-- OUT : 함수/프로시저에서 호출한 곳으로 값 전달(=마치 함수의 RETURN 처럼)
-- IN OUT : IN + OUT 합친 것과 같음
-- * 별도로 표기하지 않으면 기본값 IN 적용됨.

매번 신입사원이 입사하면, 일반 회사에서는 인사부에서 직원들의 정보를 DB에 등록하게 되는데, SQL을 이용해 직접 작성하면 번거롭고 귀찮아 집니다. 직원정보를 등록, 업데이트, 삭제하는 프로시저를 작성해보도록 합시다.

```
-- =====
-- EMPLOYEES 테이블에 데이터 입력시 자동으로 JOB_HISTORY를 업데이트하는 트리거가 있습니다.
-- 삭제 프로시저를 사용할때 주의해야합니다. 즉, 등록된 직원정보중 employee_id PK를
-- JOB_HISTORY는 FK로 참조하고 있고 이럴경우, 제대로 삭제되지 않습니다.
-- =====
-- 1. 신입사원 등록 프로시저
CREATE OR REPLACE PROCEDURE register_emp ( f_name VARCHAR2, l_name VARCHAR2, e_acct VARCHAR2, j_id VARCHAR2 )
IS
BEGIN
INSERT INTO employees (employee_id, first_name, last_name, email, hire_date, job_id)
VALUES (EMPLOYEES_SEQ.NEXTVAL, f_name, l_name, e_acct, SYSDATE, j_id);
COMMIT;
EXCEPTION WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE(f_name || ' ' || l_name || 'registration Failed!');
ROLLBACK;
END;
```

-- Procedure REGISTER_EMP이(가) 컴파일되었습니다.
-- 프로시저 실행 : EXEC 혹은 EXECUTE 프로시저명 (파라미터,...);
EXEC register_emp('길동', '홍', 'gildong', 'IT_PROG');

```

SELECT *
FROM employees
WHERE last_name='홍';

-- 2. 신규 사원의 부서, 전화번호, 급여, 매니저번호를 업데이트 하는 프로시저
CREATE OR REPLACE PROCEDURE emp_setting (emp_id NUMBER, dept_id NUMBER, phone VARCHAR2, salaries NUMBER)
IS
    sManager_ID employees.manager_id%TYPE;
BEGIN
    -- 해당 부서의 매니저값을 받아온다
    SELECT manager_id
    INTO sManager_id
    FROM employees
    WHERE department_id = dept_id;

    -- 테이블에 데이터를 업데이트 한다.
    UPDATE employees
    SET department_id = dept_id,
        phone_number = phone,
        salary = salaries,
        manager_id = sManager_id
    WHERE employee_id = emp_id;

    COMMIT;

    EXCEPTION WHEN OTHERS THEN
        dbms_output.put_line('employees update is failed');
        ROLLBACK;
END;

-- 확인하자
SELECT *
FROM employees
WHERE last_name = '홍';

-- 3. 인사이동 프로시저 (부서이동, 급여 상승)
-- 사번, 이동할 부서번호, 직무, 급여
CREATE OR REPLACE PROCEDURE emp_move
(
    emp_id NUMBER,
    trans_dept_id NUMBER,
    trans_job_id VARCHAR2,
    up_salary NUMBER
)
IS
    -- 이동할 부서번호
    new_dept_id employees.department_id%TYPE;
    -- 새로운 직급번호
    new_job_id employees.job_id%TYPE;
    -- 직급에 따른 최대 급여
    max_salaries jobs.max_salary%TYPE;
    -- 직급에 따른 최소 급여
    min_salaries jobs.min_salary%TYPE;

    -- 급여가 너무 높을때 처리할 EXCEPTION 정의
    salary_too_high EXCEPTION;
    -- 급여가 너무 낮을때 처리할 EXCEPTION 정의
    salary_too_low EXCEPTION;
BEGIN
    -- 부서 이동이 있다면
    IF trans_dept_id IS NOT NULL THEN
        new_dept_id := trans_dept_id;
    END IF;

    -- 직급 이동이 있다면
    IF trans_job_id IS NOT NULL THEN
        -- 새로운 직급ID, 최대 급여액, 최소 급여액을 가져오기
        SELECT job_id, max_salary, min_salary
        INTO new_job_id, max_salaries, min_salaries
        FROM jobs
        WHERE job_id = trans_job_id;

        -- 입력한 금액이 최대 급여액 보다 크면
        IF up_salary > max_salaries THEN
            -- 예외발생
            RAISE salary_too_high;
        ELSIF up_salary < min_salaries THEN
            RAISE salary_too_low;
        END IF;
    END IF;

    -- 부서, 직급, 급여 내역을 업데이트한다.
    UPDATE employees
    SET department_id = NVL(new_dept_id, department_id),
        job_id = NVL(new_job_id, job_id),
        salary = NVL(up_salary, salary)
    WHERE employee_id = emp_id;

```

```

COMMIT;
EXCEPTION WHEN salary_too_high THEN
    DBMS_OUTPUT.PUT_LINE('Salary exceed');
    ROLLBACK;
WHEN salary_too_low THEN
    DBMS_OUTPUT.PUT_LINE('salry too low');
    ROLLBACK;
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
    ROLLBACK;
END;

-- 인사이동
EXEC emp_move(198, 60, 'IT_PROG', 4400);

SELECT *
FROM employees
WHERE employee_id = 198; -- salary 2600 up to 4400

-- 198번 Donald OConnell은 배송부서(Shipping) 인데, 60번 부서 IT로 IT_PROG로 인사이동

SELECT *
FROM departments
WHERE department_id = 60;

```

스크립트 출력 x

질의 결과 x

인출된 모든 행: 1(0.001초)

SQL

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	
1	207	yeonghun	seon	sprax	062.362.7797	22/11/02	IT_PROG	5500	(null)	103	60

< 업데이트 프로시저 실행결과 >

신규사원 등록, 사원정보 업데이트 그리고 인사이동 프로시저의 내용을 작성해보니 복잡해 보이는 것도 사실이나, 실제 프로젝트에서 사용하는 프로시저에 비하면 간단한 편입니다. 왜냐하면, 실제 업무는 employees 테이블에서만 데이터 작업을 하는게 아니라 다수의 테이블에서 데이터를 입력하거나 업데이트 하기 때문이죠.

예를들어 사원 테이블, 사원 가족 테이블, 학력 테이블, 자격증 테이블, 인사평가 테이블등에 신규 사원을 등록할때를 생각해보면 거의 동시에 여러 테이블의 데이터를 입력하거나 업데이트 해야 합니다.

예외 발생시 기존에 처리한 모든 변경사항을 ROLLBACK 시키고, 정상적으로 쿼리가 실행된다면 COMMIT으로 트랜잭션을 처리하였습니다.



일련의 작업을 꼭 프로시저로 만들 필요는 없으나 성능이나 수행속도, 서버 부하등을 고려했을때 권장할만 합니다.

▼ ON DELETE CASCADE : HR 스키마의 EMPLOYEES 테이블에 데이터 입력 후 삭제하고자 할때

foreign key 로 연결된 데이터들이 일관성을 유지할 수 있도록 하기 위해서, foreign key constraints 라는 것이 있습니다. 공식 문서는 CASCADE 를 foreign key constraints 에서 옵션으로 사용할 수 있는 Referential Actions 이라고 설명합니다.

기존 제약조건을 삭제 후 업데이트 하지 않고 작업하다보면 HR 스키마의 경우, 샘플로 제공된 트리거에 의해 사원테이블에 사원정보를 추가하면, 자동으로 JOB_HISTORY 테이블에 사원정보를 입력하게 설계되어 있습니다. 이후 삭제를 시도할때 문제가 되었던 바 이 섹션을 기록을 위해 남겨둡니다.

```

-- 기존 JOB_HISTORY에 있는 FK 삭제
ALTER TABLE JOB_HISTORY
DROP CONSTRAINT JHIST_EMP_FK;

```

```
-- JOB_HISTORY에 다시 FK 제약조건 추가 + ON DELETE CASCADE 설정
ALTER TABLE JOB_HISTORY
ADD CONSTRAINT JHIST_EMP_FK FOREIGN KEY (employee_id) REFERENCES employees (employee_id)
ON DELETE CASCADE;
```

스크립트 출력 x | 질의 결과 x | 질의 결과 1 x | 질의 결과 2 x

SQL | 인출된 모든 행: 1(0,004초)

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	208	minsu	choi	minsu-choi	82.062.362.7797	22/11/02 IT_PROG	4000	(null)	121	50

스크립트 출력 x | 질의 결과 x | 질의 결과 1 x | 질의 결과 2 x | 질의 결과 3 x

SQL | 인출된 모든 행: 11(0,005초)

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
5	122 07/01/01	07/12/31	ST_CLERK	50
6	176 06/03/24	06/12/31	SA_REP	80
7	176 07/01/01	07/12/31	SA_MAN	80
8	200 95/09/17	01/06/17	AD_ASST	90
9	200 02/07/01	06/12/31	AC_ACCOUNT	90
10	201 04/02/17	07/12/19	MK_REP	20
11	208 22/11/02	22/11/02	IT_PROG	(null)

3. 트리거

트리거는 프로시저의 한 종류로 개발자가 호출하는게 아닌 특정 이벤트에 반응해 자동으로 실행되는 프로시저라고 볼 수 있습니다.

예를들어, 입고 테이블에 상품이 입고되면 (insert 이벤트 발생) 재고 테이블에 자동으로 재고가 증가(update)하게 만드는 것과 같은 작업을 자동으로 처리합니다.

▼ 일반 계정에서 트리거 생성 권한이 없을때 조치

기본적으로 일반 계정에서 트리거를 생성하는 권한은 부여되어 있지 않습니다. 이럴 경우, 별도 권한을 요청 하여야 합니다.

```
-- 트리거 관련 권한
-- 별도 권한 작업 없이 계정을 생성했을 경우, 아래의 3가지 권한만 추가해 주면,
-- 이용 가능하다.

GRANT CREATE TRIGGER TO userid;
GRANT CREATE ANY TRIGGER TO userid;
GRANT ALTER ANY TRIGGER TO userid;

-- 현재 계정에 할당된 권한 확인하기
SELECT GRANTEE, TABLE_NAME, GRANTOR, PRIVILEGE
FROM USER_TAB_PRIVS;

-- 접속 후 계정 전환
-- SYSDBA는 관리자 권한을 뜻함
CONN {유저명}/{비번} {AS SYSDBA}

-- 현재 접속 사용자 확인
SHOW USER;

-- 계정 잠금 상태 조회
SELECT USERNAME, ACCOUNT_STATUS, LOCK_DATE FROM DBA_USERS WHERE USERNAME = '{유저명}';

-- 계정 잠금 해제
ALTER USER {유저명} ACCOUNT UNLOCK;

-- 새 계정 만들기
ALTER SESSION SET "_ORACLE_SCRIPT"=true; -- c## 으로 시작하지 않는 계정 만들려면
alter system set sec_case_sensitive_logon = false; -- 오라클 11g 이후 대소문자구분 안하게
CREATE USER {유저명} IDENTIFIED BY '{비번}'; -- 오라클 12c 부터 C## 붙여야 함

-- 사용자 계정명 변경하기
SELECT NAME FROM USER$ WHERE NAME = '{유저명}'; -- 있는지 조회
```

```

UPDATE USERS$ SET NAME = '{바꿀유저명}' WHERE NAME = '{대상유저명}'; -- on update cascade

-- 특정 사용자의 비번 변경
ALTER USER {유저명} IDENTIFIED BY "{비번}";

-- 특정 사용자 삭제
DROP USER {유저명};

-- 권한관리
GRANT CONNECT, RESOURCE, DBA TO {유저명}; --권한부여
REVOKE [DBA 등] FROM {유저명}; --권한 제거

SELECT * FROM DBA_ROLE_PRIVS WHERE GRANTEE = '{유저명}';
SELECT * FROM DBA_SYS_PRIVS WHERE GRANTEE = '{유저명}';

-- PDB 부터는 조금 다름
-- multitenant connect 링크 참조
-- https://oracle-base.com/articles/12c/multitenant-connecting-to-cdb-and-pdb-12cr1

-- pdb 뷰 조회
SELECT name FROM v$pdbs;

-- pdb 계정 전환
alter session set container = XEPDB1;
ALTER SESSION SET CONTAINER = '

-- 현재 접속 컨테이너 조회
show con_name;

-- 일반 사용자는 PDB 에서 SYSKM, SYSOPER, SYSRAC 권한을 제외하도록!

```

스크립트 출력 x | 결과 x

SQL | 인출된 모든 행: 2(1,26초)

	GRANTEE	TABLE_NAME	GRANTOR	PRIVILEGE
1	HR	DBMS_STATS	SYS	EXECUTE
2	PUBLIC	HR	HR	INHERIT PRIVILEGES



트리거는 잘 사용하면 아주 편리한 기능이지만, 잘못 생성 하거나 너무 많이 생성하게 되면 관련 오브젝트끼리 복잡한 종속 관계가 되어 **성능 저하**가 생길 수 있습니다.

또한, 데이터베이스 전체의 트리거 조작은 administer database trigger 시스템 권한이 필요합니다.

※ 트리거 생성, 수정, 삭제 시 create trigger, alter trigger, drop trigger의 권한이 필요함.

```

-- 트리거 생성
CREATE OR REPLACE TRIGGER 트리거명
BEFORE OR AFTER
INSERT OR UPDATE OR DELETE ON테이블명
-- [FOR EACH ROW]

BEGIN
-- 실행부
-- INSERT, UPDATE, DELETE
EXCEPTION
-- 예외처리부
END;

-- 트리거 예시 : HR.departments
create or replace trigger tr_dept_insert
after insert on dept
begin
dbms_output.put_line('정상적으로 입력되었습니다.');
```

```

end;

-- 트리거 실행확인
INSERT INTO departments
VALUES (280, 'Counsel', 1800);
-- 정상적으로 입력되었습니다.

```

```

-- 트리거 예시 : 신규 테이블 생성
-- 테이블에 값 입력하는 시간을 지정하기
create table t_order(
no number,
ord_code varchar2(10),
ord_date date
);

-- 시간
create or replace trigger tr_check_time
before insert on t_order
begin
if to_char(sysdate, 'HH24:mi') not between '12:30' and '12:50' then
raise_application_error(-20009, '12:30 ~ 12:50 일 경우만 입력가능');
end if;
end;

-- 트리거 예시 : 특정 레코드만 삽입, 나머지 막기
-- 제품 코드가 'C100'인 제품이 입력될 경우 입력을 허용하고, 나머지 제품은
--- 모두 에러를 발생시키는 트리거 작성
create or replace trigger tr_code_check
before insert on t_order
for each row -- (1) 행 레벨 트리거 : (예를들어 10번 업데이트했다 , 그럼 10번을 수행함) 즉 , 행의 갯수만큼 영향 받은 행의 갯수만큼 진행된다
begin
if :new.ord_code != 'C100' then --(2) : 새로 입력하는 ord_code가 C100이 아니면 error가 뜨게 한다.
raise_application_error(-20010, '제품코드가 C100인 제품만 입력 가능!');
end if;
end;

-- [ 기본 예제 1]
-- 트리거 예제에서 사용할 테이블, 시퀀스 생성
CREATE TABLE tblLog (
seq NUMBER PRIMARY KEY,
num NUMBER NOT NULL REFERENCES tblInsa(num),
regdate DATE DEFAULT SYSDATE NOT NULL
)

CREATE SEQUENCE seqLog;

CREATE TABLE tblBoard (
seq NUMBER PRIMARY KEY,
num NUMBER NOT NULL REFERENCES tblInsa(num),
subject VARCHAR2(1000) NOT NULL
)

CREATE SEQUENCE seqBoard;

-- 트리거 생성
-- 직원들 > tblBoard 글 작성 > 관리자 확인 + 모니터링 + tblLog
CREATE OR REPLACE TRIGGER trgBoard
AFTER -- 사전 전/후 (BEFORE/AFTER)
INSERT -- 감시 사건 (INSERT, UPDATE, DELETE)
ON tblBoard -- 감시 대상 (테이블)
DECLARE
BEGIN
DBMS_OUTPUT.PUT_LINE('직원이 글을 작성 했습니다.');
```

```
END;
```

```
-- 실행
-- > 트리거 객체 생성 + 트리거 작동시작
```

```
-- 값 추가해서 확인하기
INSERT INTO tblBoard (seq, num, subject)
VALUES (seqBoard.nextVal, (SELECT NUM FROM TBLINSA WHERE NAME = '홍길동'), '테스트 입니다.');
```

```
-- 실행
-- > 직원이 글을 작성 했습니다.
-- > 1 행 이(가) 삽입되었습니다.
```

```
-- [ 트리거 로그 기록 예제 2 ]
```

```
-- 트리거 생성
CREATE OR REPLACE TRIGGER trgBoard
AFTER
INSERT
ON tblBoard
DECLARE
vnum NUMBER;
BEGIN
-- 누가 글을 작성 했는지 tblLog 테이블에 기록하기
SELECT num INTO vnum FROM tblBoard WHERE seq = (SELECT MAX(seq) FROM tblBoard);
```

```
-- 로그 기록
INSERT INTO tblLog (seq, num, regdate) VALUES (seqLog.nextVal, vnum, default);
END;

-- 데이터 추가해서 확인하기
INSERT INTO tblBoard (seq, num, subject)
VALUES (seqBoard.nextVal, (SELECT NUM FROM TBLINSA WHERE NAME = '홍길동'), '테스트 입니다.');
```

```
INSERT INTO tblBoard (seq, num, subject)
VALUES (seqBoard.nextVal, (SELECT NUM FROM TBLINSA WHERE NAME = '이순신'), '테스트 입니다.');
```

```
INSERT INTO tblBoard (seq, num, subject)
VALUES (seqBoard.nextVal, (SELECT NUM FROM TBLINSA WHERE NAME = '유관순'), '테스트 입니다.');
```

```
SELECT * FROM tblLog;
--> tblLog에 로그기록(누가 글을 작성 했는지) 저장확인
```

```
-- [ IF를 이용한 트리거 예제 3 ]
-- tblInsa, 직원 퇴사
-- 특정 요일(목요일)에는 퇴사를 할 수 없다.
CREATE OR REPLACE TRIGGER trgInsa
BEFORE
DELETE
ON tblInsa
BEGIN
IF TO_CHAR(SYSDATE, 'dy') = '목' THEN
-- 현재 실행 되려는 DELETE 작업을 중단결로 만들기 -> 강제로 예외 발생
RAISE_APPLICATION_ERROR(-20001, '목요일에는 퇴사가 불가능합니다.');
```

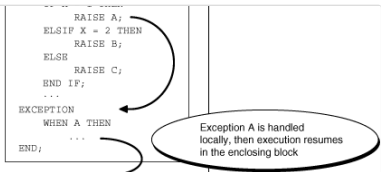
```
END IF;
END trgInsa;

-- 정상 작동하는지 테스트를 위한 DELETE
DELETE FROM tblInsa WHERE NUM = 1001; --> 현재 오늘 날짜는 2021-06-03 (목요일)이므로 삭제 X -> '목요일에는 퇴사가 불가능합니다. 로그 출력
```

Database PL/SQL Language Reference

The script content on this page is for navigation purposes only and does not alter the content in any way. This chapter explains how to handle PL/SQL compile-time warnings and PL/SQL runtime errors. The latter are called exceptions. Tip: If you have problems creating or running PL/SQL code, check the Oracle

<https://docs.oracle.com/en/database/oracle/oracle-database/21/lnpls/plsql-error-handling.html#GUID-48F88C61-8CE9-4821-91CB-48A8F1BC09E1>



< 트리거에 사용하는 명시적 예외처리 >

3. 패키지

패키지는 '꾸러미, 묶음'이라는 뜻을 가진 단어이고, PL/SQL에서도 비슷하게 무언가를 묶는데 사용합니다. 패키지는 함수나 프로시저를 묶는 것을 말하는데, 처리하는 작업의 성격이 비슷한 함수나 프로시저를 하나로 묶어놓은 오라클 객체를 의미합니다.

예를들어, 이전의 프로시저에서 수행했던 직원등록, 직원정보 업데이트, 인사이동등의 모든 인사관련 업무를 수행하는 프로시저를 만들어 봤지만 어떤 회사의 업무 전체를 놓고 봤을때 인사에 관련된 부분만 있는 것은 아닙니다.

물건을 판매하는데 주문을 받는 프로시저가 있을 수 있고 고객 정보를 등록하는 프로시저가 있을수도 있습니다. 회사의 규모가 크거나 수행하는 업무가 많다면, 사용되는 프로시저나 함수가 꽤 많을텐데, 그만큼 어떤 함수가 어떤 프로시저가 어떤 업무들을 처리하는지 구별하기 힘들어 지므로 비슷한 유형의 작업을 수행하는 함수나 프로시저를 패키지로 묶어서 사용하게 합니다.

예를들어, 아래와 같이 함수나 프로시저가 있다고 가정해봅시다.

함수목록(Function)

부서명 반환 : get_dept_name

직원정보 반환 : get_employee

재고상품 조회 : get_item_qty

주문상세조회 : get_order_detail

...

프로시저 목록(Procedure)

사원등록 : register_emp

사원정보 업데이트 : update_emp

상품등록 : register_item

인사이동 : emp_move

주문등록 : register_order

주문상품배송 : shipping_order_item

...

다수의 함수와 프로시저를 아래와 같이 분류해 하나의 묶음, 즉 패키지로 만들어봅시다.

인사업무 패키지(HR_PKG)

부서명 반환 : get_dept_name

사원정보 반환 : get_employee

사원등록 : register_emp

사원정보 업데이트 : update_emp

인사이동 : emp_move

주문업무 패키지(ORDER_PKG)

주문등록 : register_order

주문상세조회 : get_order_detail

주문상품배송 : shipping_order_item

이런식으로 패키지로 묶어두면 인사업무 관련된 서브프로그램은 HR_PKG를 참조하고, 주문업무 관련된 것은 ORDER_PKG를 참조하면 구별이 쉬워집니다.

위와 같은 방식으로 패키지를 정의하고 구현하면, 아래와 같이 함수나 프로시저를 사용할 수 있습니다.

```
EXECUTE HR_PKG.get_dept_name(100);
```

```
EXECUTE HR_PKG.get_employee(100);
```



패키지란, 오라클 데이터베이스에 저장된 프로시저, 함수 뿐만 아니라 변수, 상수, 커서, EXCEPTION을 하나로 묶은 캡슐화된 객체를 뜻합니다.

▼ 패키지(PACKAGE)의 장점

- 1) 애플리케이션을 효율적으로 개발할 수 있게 도와줍니다.
- 2) 관련된 스키마 오브젝트들을 재 컴파일할 필요 없이 수정 가능합니다. 서브프로그램들은 종속성이 있기 때문입니다. 어떤 함수의 내용이 변경된다면, 오라클 패키지내에서 변경된 함수나 프로시저들을 다시 컴파일 하지 않아도 됩니다.
- 3) 한번에 여러개의 패키지 오브젝트들을 메모리로 로드할 수 있습니다.

4) 프로시저나 함수들의 오버로딩이 가능합니다.

5) 패키지 내의 모든 타입, 항목, 서브프로그램들을 PUBLIC 이나 PRIVATE으로 선언해서 사용할 수 있습니다.

그럼, 패키지의 구조와 사용방법에 대해서 알아보겠습니다. 패키지는 함수나 프로시저와 달리 크게 2 부분으로 구성되는데, 바로 (1) 패키지 명세부 와 (2) 패키지 구현부 입니다.

(1) 명세부 : 변수, 상수, EXCEPTION, 커서, 함수, 프로시저를 선언하는 부분으로 이러한 객체들은 모두 PUBLIC 속성을 갖게 됩니다. 즉, 패키지 외부에서 접근이 가능해집니다.

(2) 구현부 : 패키지 명세부에서 선언한 내용을 실제 구현하는 부분으로, 어느 하나의 패키지가 데이터 타입, 변수, 상수, EXCEPTION만 선언했다면 이 패키지는 구현부가 필요하지 않게 됩니다. 즉, 패키지는 선언부만으로도 구성될 수 있다는 뜻입니다. 또한, 구현부에서만 객체를 선언해서 사용할 수 있는데, 이러한 객체는 패키지 내부에서만 참조 가능한 PRIVATE 속성을 갖게 됩니다. 생각가능!



패키지 선언부에서 커서, 함수, 프로시저를 선언하였다면 구현부에서는 구체적인 처리내용을 기술해주어야 합니다.

```
-- 패키지 예시

-- 1. 패키지 명세[필수]
CREATE OR REPLACE PACKAGE hello IS[AS]
    PROCEDURE p_test(p_name VARCHAR2);
END;

CREATE OR REPLACE PACKAGE employee_process AS
    -- PUBLIC 속성을 가
    -- 타입, 커서, EXCEPTION 선언
    TYPE EmpRecord IS RECORD (emp_id INT, salary INT);
    TYPE DeptRecord IS RECORD (dept_id INT, loc_id INT);
    CURSOR salaries RETURN EmpRecord;
    -- 예외 선언 :
    invalid_salary EXCEPTION;
    -- 프로시저 선언
    PROCEDURE hire_emp(
        first_name VARCHAR2,
        last_name VARCHAR2,
        emails VARCHAR2,
        job_id VARCHAR2,
        salary INT,
        commission REAL, --floating point
        dept_id REAL);
    PROCEDURE fire_emp(emp_id INT);
END;

-- 2. 패키지 바디 [필수 아님]
CREATE OR REPLACE PACKAGE BODY hello IS
    PROCEDURE p_test(p_name VARCHAR2) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('hello ' || p_name);
    END;
END;

CREATE OR REPLACE PACKAGE BODY employees_process AS
    -- PRIVATE 속성을 가짐
    number_hired INT;
    -- 명세부에서 정의한 CURSOR를 정의
    CURSOR salaries RETURN EmpRecord IS
    SELECT employee_id, salary
    FROM employees
    ORDER BY salary DESC;

    -- 신규 사원등록 프로시저
    PROCEDURE hire_emp(
        first_name VARCHAR2,
        last_name VARCHAR2,
        emails VARCHAR2,
        job_id VARCHAR2,
        mgr_id INT,
        salary INT,
        commission REAL, --floating point
        dept_id REAL) IS
    new_emp_id INT;
    BEGIN
        -- 신규사원 등록을 위한 employee_id 값의 시퀀스 알아오기
```

```

SELECT employees_seq.NEXTVAL
INTO new_emp_id
FROM dual;

-- 신규사원 등록처리
INSERT INTO employees (employee_id, first_name, last_name, email, job_id, manager_id, hire_date, salary, commission_pct, department_id)
VALUES (new_emp_id, first_name, last_name, emails, job_id, mgr_id, SYSDATE, salary, commission, dept_id);
number_hired := number_hired + 1;
END;

-- 퇴사처리
PROCEDURE fire_emp(emp_id INT) IS
BEGIN
    -- 퇴사할 사원을 테이블에서 삭제
    DELETE employees
    WHERE employee_id = emp_id;
    COMMIT;

    EXCEPTION WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('DELETE ERROR');
        ROLLBACK;
END;

BEGIN
    number_hired := 0;
END;

-- 실행
EXEC hello.p_test('kim');
EXEC employee_process.hire_emp('길동', '홍', 'gildong', 'IT_PROG', 210, 5000, 0, 60);

-- 트랜잭션 처리
COMMIT;

```

▼ 시스템 패키지

I. 패키지 개요

오라클에서 제공하는 유용한 패키지를 시스템 패키지라고 합니다. 오라클에서 제공되는 각종 API라고 볼수 있겠네요

API는 Application Programming Interface(=애플리케이션 프로그램 인터페이스)는 구체적인 구현 사항은 알지 못해도 사용방법만 알면 시스템 패키지의 기능을 사용할수 있습니다

오라클에서 제공하는 시스템패키지는 DBMS와 UTL을 접두어로 구분하고 추가적인 접두어가 몇개 더 있습니다.



데이터 덱서너리는 DBA_, USER_, ALL_을 접두어로 해서 구분합니다.

DBMS 접두어가 붙은 패키지는 오라클이 제공하는 시스템 패키지중 가장 많고 그 기능들이 각기 다릅니다.

```

-- SYSTEM이나 SYS 계정으로 확인!
SELECT COUNT(*) pkg_qty
FROM dba_objects
WHERE object_name LIKE 'DBMS%'
AND object_type = 'PACKAGE'; -- quantity: 518

```

이것중 가장 자주 사용한 패키지인 DBMS_OUTPUT 패키지에 대해서 알아보시다. OUTPUT 이란 패키지 이름 그대로 출력과 관련된 기능을 수행합니다.

DBMS_OUTPUT 패키지를 이용해 출력결과를 보려면 SET SERVEROUTPUT ON 이라는 명령어를 실행해야 하며, 이것은 세션이 오픈된 상태에서는 계속 설정이 유지 됩니다.

II. DBMS_OUTPUT 패키지

앞서 패키지는 비슷한 기능을 수행하는 함수, 프로시저, 타입등의 묶음이라고 했습니다. DBMS_OUTPUT 패키지도 출력관련 함수나 프로시저등이 포함되어 있습니다. 아래, 북마크를 참고하여 함께 보시기 바랍니다.

PL/SQL Packages and Types Reference

PL/SQL Packages and Types Reference Introduction to Oracle Supplied PL/SQL Packages & Types Oracle supplies many PL/SQL packages with the Oracle server to extend database functionality and provide PL/SQL access to SQL features. You can use the supplied packages when creating your applications or for ideas in creating your own stored procedures.

<https://docs.oracle.com/en/database/oracle/oracle-database/21/arpls/introduction-to-oracle-supplied-plsql-packages-and-types.html#GUID-5A731E74-DD0A-4FBA-A862-AA9DEA9E3793>

< 오라클 제공 PL/SQL 패키지 목록 >

DBMS_OUTPUT 패키지는 오라클 내부의 출력 버퍼에 메시지를 쓰는 패키지입니다.

PL/SQL Packages and Types Reference

The DBMS_OUTPUT package enables you to send messages from stored procedures, packages, and triggers. The package is especially useful for displaying PL/SQL debugging information. This chapter contains the following topics: The package is typically used for debugging, or for displaying messages and reports to SQL*DBA or SQL*Plus (such as are produced by applying the SQL command DESCRIBE to procedures).

https://docs.oracle.com/en/database/oracle/oracle-database/21/arpls/DBMS_OUTPUT.html#GUID-C1400094-18D5-4F36-A2C9-D28B0E12FD8C

- PUT_LINE 프로시저 : PUT과 같은 기능이나 END_OF_LINE 값을 버퍼에 함께 추가합니다.

```
DBMS_OUTPUT.PUT_LINE('Hello PL/SQL');  
DBMS_OUTPUT.PUT_LINE('Procedural Language extension to SQL');
```

DBMS_JOB 패키지는 특정 시간에 어떠한 작업을 하게 하는 스케줄링 기능을 제공하는 패키지입니다. 이와 유사하게는 UNIX의 CRON , 맥은 CRONTAB, 윈도우는 작업스케줄러(Task Scheduler)등과 같습니다.

PL/SQL Packages and Types Reference

The DBMS_JOB package schedules and manages jobs in the job queue. Note: The DBMS_JOB package has been superseded by the DBMS_SCHEDULER package, and support for DBMS_JOB might be removed in future releases of Oracle Database. In particular, if you are administering jobs to manage system load, you are encouraged to disable DBMS_JOB by revoking the package execution privilege for users.

https://docs.oracle.com/en/database/oracle/oracle-database/21/arpls/DBMS_JOB.html#GUID-8C62D808-D7A3-4D21-B87F-A229B7CE1956

다음의 DBMS_JOB에 포함된 대표적인 프로시저를 사용해 스케줄을 생성하고, 수정하고, 실행할 수 있습니다.

- DBMS_JOB.SUBMIT : 스케줄 등록
- DBMS_JOB.INSTANCE : 작업을 실행할 인스턴스 할당
- DBMS_JOB.CHANGE : 작업관련 사용자 정의 매개변수를 수정(=alter)
- DBMS_JOB.RUN : 스케줄 실행

그리고 UTL_FILE 패키지등이 있습니다. 서버에 있는 파일을 조작하는데 사용하는 패키지라고 볼 수 있습니다.

PL/SQL Packages and Types Reference

With the UTL_FILE package, PL/SQL programs can read and write operating system text files. UTL_FILE provides a restricted version of operating system stream file I/O. This chapter contains the following topics: The set of files and directories that are accessible to the user through UTL_FILE is controlled by a number of factors and database parameters.

https://docs.oracle.com/en/database/oracle/oracle-database/21/arpls/UTL_FILE.html#GUID-EBC42A36-EB72-4AA1-B75F-8CF4BC6E29B4

```
-- DBMS_JOB : 스케줄링 패키지
-- Job Queue에 일정 시간에 실행될 Job(작업)을 등록
DBMS_JOB.SUBMIT (
  job OUT BINARY_INTEGER,
  what IN VARCHAR2,
  next_date IN DATE DEFAULT SYSDATE,
  interval IN VARCHAR2 DEFAULT 'NULL',
  no_parse IN BOOLEAN DEFAULT FALSE,
```

<오라클데이터베이스 + JAVASCRIPT, XML 다루기 vs 오라클 클라우드>

모든 과정을 학습했습니다. 이제 실습을 통해 그 결과를 확인합니다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/caae43ae-779f-40fe-8563-aaa690898757/ORACLE%E8%B%A4%EC%8A%B5%EA%B3%BC%EC%A0%9C_%EC%A2%85%ED%95%A9%EB%AC%B8%EC%A0%9C.pdf



종합문제에 포함된 주제와 관련해서 이미 개발되어있는 시스템의 일부를 문제로 만든것이므로, 학사관리 시스템에 대한 전반적인 업무파악을 위한 검색을 해보신 다음에 문제를 해결하시길 권합니다.

종합문제 문제풀이

▼ 데이터 모델링

첨부된 모델을 참고하세요!

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/0ad955a7-29fa-436d-a539-1d0e4b2dff80/%EC%98%A4%EB%9D%BC%ED%81%B4%EC%8B%A4%EC%8A%B5%EA%B3%BC%EC%A0%9C-%ED%95%99%EC%82%AC%EA%B4%80%EB%A6%AC%EC%8B%9C%EC%8A%A4%ED%85%9C.drawio>

< draw.io에서 열어보세요 >

▼ 문제풀이(과제 2)

문제 2-1 ~ 문제 2~17까지 진행

▼ PL/SQL (함수, 프로시저, 트리거, 패키지) Named Block 이해하기

* PL/SQL 이해하기

1. PL/SQL 명령처리는 블록단위 : 익명 블록(Anonymous Block) vs 이름있는 블록(Named Block)
2. 학사관리 시스템에서 반복되는 업무를 파악해보기 <----> 프로시저나 함수단위로 구현할 부분
 - 학생 등록 업무
 - 과정 등록 업무
 - 개설과정 조회업무
 - 등록된 학생조회 업무
 - 과정 삭제 업무 (주기적으로 과정종료일을 체크해서 자동으로 삭제하려면, 스케줄러를 만들어야 함)
 - 학생 삭제 업무
 - 과정 정보 업데이트 업무
 - 학생 정보 업데이트 업무
 -
3. 업무를 좀 더 세분화 해보기 <-----> 패키지로 관리하기 (ex. 학원 학사관리 시스템)

- 등록 업무

- 조회업무

- 삭제 업무

4. 패키지 만들어 보기

- 패키지는 크게 명세부와 구현부로 나눔

- (1) 명세부는 변수, 상수, (타입형 변수), 커서, 레코드와 예외, 함수, 프로시저등을 선언하는 곳

- (2) 구현부는 실제 처리로직을 기술하는 곳



명세부는 패키지의 선언 구간으로 패키지의 헤더(header)라고 볼 수 있으나, 구현부는 패키지의 실제 처리 로직을 기술하는 곳으로 CREATE OR REPLACE PACKAGE BODY 라고 정의 해 주어야 합니다.

※ header는 따로 표기하지 않습니다. 생략~!

```
-- (1) 패키지 명세부
-- 패키지의 헤더 : header라고 따로 작성하지 않음!

CREATE OR REPLACE PACKAGE academy_job IS [AS]
    nStudent_Count NUMBER;
BEGIN

    PROCEDURE add_student(id NUMBER, name VARCHAR2, phone CHAR, dept_id NUMBER);
    PROCEDURE del_student(id NUMBER);
    FUNCTION list_student_registration(id NUMBER);
    .... 계속...

END [academy_job]; -- 패키지, 프로시저, 함수명을 작성(옵션)

-- (2) 패키지의 구현부
-- 패키지의 바디 : body라고 명시해야 함!

CREATE OR REPLACE PACKAGE BODY academy_job IS[AS]

BEGIN
    nStudent_Count := 0; -- 변수 초기화, 등록된 학생 수 카운트용
    -- 패키지 블록 내에 또다른 프로시저 블록인 BEGIN ~ END가 나올
    PROCEDURE add_student(id NUMBER, name VARCHAR2, phone CHAR, dept_id NUMBER) IS
    BEGIN
        NULL; -- 코드 길이상 생략, 실제 처리 로직을 입력하세요
    END;

    PROCEDURE del_student(id NUMBER) IS
    BEGIN
        NULL; -- 코드 길이상 생략, 실제 처리 로직을 입력하세요
    END;

    FUNCTION list_student_registration(id NUMBER) RETURN VARCHAR2 IS
    BEGIN
        NULL; -- 함수는 RETURN 값이 반드시 있음
    END;
    -- 함수나 프로시저를 비슷한 기능끼리 묶어야 할 필요가 있을때 패키지를 생성
EXCEPTION WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Exception 발생!');
    ROLLBACK; -- 예외 발생시, 모든 트랜잭션을 취소! (프로시저나 함수블록에 둘수도있음)
END;

-- 패키지 내의 함수나 프로시저 실행하기
EXEC academy_job.add_student('등록학생명', '학생연락처', '학과번호');
EXEC academy_job.del_student('학번');

--EXEC 패키지를 프로시저명(파라미터값);
--EXECUTE 패키지를 함수명(파라미터값);
```