

# PL/SQL

## 1. PL/SQL이란?

PL/SQL은 ‘Procedural language extension to Structured Query Language(SQL)’의 약자 SQL을 확장한 순차적 처리 언어

데이터베이스 질의어인 SQL과 일반 프로그래밍 언어의 특성을 결합한 언어. 즉 PL/SQL을 사용하면 조건문이나 반복문, 변수나 상수를 선언해서 사용할 수 있을 뿐만 아니라 SQL도 사용할 수 있음

## 2. PL/SQL의 기본 구조

PL/SQL의 기본 단위는 블록(block)

- 1) 선언부(declarative part) : 사용할 변수나 상수를 선언(선언부에만 변수와 상수 선언 가능)
- 2) 실행부(executable part) : 실제 처리할 로직을 담당하는 부분
- 3) 예외처리부(exception-building part) : 실행부에서 로직을 처리하던 중 발생할 수 있는 각종 오류들에 대해 처리

## 3. 기본 형태 작성 및 출력

### 1) 기본 구조

```
begin
    dbms_output.put_line('Hello World!');
end;
-----
declare
    message VARCHAR2(100);
begin
    -- 실행부에 사용할 변수는 선언부에서 미리 선언되어야 함
    message := 'Hello World!';
    dbms_output.put_line(message);
end;
-----
declare
    message VARCHAR2(100):='HELLO WORLD!';
begin
    dbms_output.put_line(message);
end;
-----
declare
    counter integer;
begin
    counter := counter + 1;
    if counter is null then
```

```
    dbms_output.put_line('Result : counter is null');
end if;
end;
```

## 2) 메시지 출력

\* 결과 메시지 출력 : SQL\*Plus에서 다음의 명령어 입력

```
SQL> SET SERVEROUTPUT ON;
```

\* SQL Developer에서 결과 메시지 출력

```
메뉴>보기>DBMS 출력>접속에 DBMSOUTPUT 사용 선택(+ 아이콘)
```

## 3) 익명 블록(ANONYMOUS BLOCK)

```
declare
  counter integer;
  i integer;
begin
  for i in 1..9 loop
    counter := (2*i);
    dbms_output.put_line(' 2 * ' || i || ' = ' || counter);
  end loop;
end;
```

## 4) 예외처리부

[구문형식]

```
Exception when 예외1 then 예외처리1
  when 예외2 then 예외처리2
  ...
  when others then 예외처리
```

```
declare
  counter integer;
begin
  counter :=10;
  counter :=counter/0;
  dbms_output.put_line(counter);
```

```

exception when others then
  dbms_output.put_line('errors');
end;

```

## 5) EXCEPTION의 구분

예외 내용	예외 번호	SQL CODE	발생 시점
ACCESS_INTO_NULL	ORA-06530	-6530	초기화 되지 않은 오브젝트에 값을 할당하려고 할 경우
CASE_NOT_FOUND	ORA-06592	-6592	CASE 문장에서 ELSE 구문도 없고 WHEN 절에 명시된 조건을 만족하는 것이 하나도 없을 경우
COLLECTION_IS_NULL	ORA-06531	-6531	초기화 되지 않은 중첩 테이블이나 VARRAY같은 콜렉션을 EXISTS 외의 다른 메소드로 접근을 시도할 경우 발생
CURSOR_ALREADY_OPEN	ORA-06511	-6511	이미 오픈된 커서를 다시 오픈하려고 시도하는 경우
DUP_VAL_ON_INDEX	ORA-00001	-1	유일 인덱스가 걸린 컬럼에 중복 데이터를 입력할 경우
INVALID_CURSOR	ORA-01001	-1001	허용되지 않은 커서에 접근할 경우(오픈되지 않은 커서를 닫으려고 시도하는 경우)
INVALID_NUMBER	ORA-01722	-1722	SQL 문장에서 문자형 숫자형으로 변환할 때 제대로 된 숫자로 변환이 되지 않을 경우
LOGIN_DENIED	ORA-01017	-1017	잘못된 사용자나 비밀번호로 로그인을 시도할 때
NO_DATA_FOUND	ORA-01403	+100	SELECT INTO 문장의 결과로 선택된 로우가 하나도 없을 경우
NO_LOGGED_ON	ORA-01012	-1012	오라클에 연결되지 않았을 경우
PROGRAM_ERROR	ORA-06501	-6501	PL/SQL 내부에 문제가 발생했을 경우

SELF_IS_NULL	ORA-30625	-30625	OBJECT 타입이 초기화 되지 않은 상태에서 MEMBER 메소드를 사용할 경우
STORAGE_ERROR	ORA-06500	-6500	메모리가 부족한 경우
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533	중첩 테이블이나 VARRAY의 요소값에 접근할 때, 명시한 인덱스 번호가 콜렉션 전체 크기를 넘어설 경우
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532	중첩 테이블이나 VARRAY의 요소값에 접근할 때, 잘못된 인덱스 번호를 사용할 경우(예, 인덱스 번호로 -1 사용시)
SYS_INVALID_ROWID	ORA-01410	-1410	문자열을 ROWID로 변환할 때 변환값에 해당하는 ROWID값이 없을 경우
TIMEOUT_ON_RESOU RCE	ORA-00051	-51	오라클이 리소스를 기다리는 동안 타임아웃이 발생했을 때
TOO_MANY_ROWS	ORA-01422	-1422	SELECT INTO 문장에서 하나 이상의 로우가 반환될 때
VALUE_ERROR	ORA-06502	-6502	문자형 데이터를 숫자형으로 변환하는데 타당한 숫자가 아니거나 값을 할당 시 값의 크기가 선언된 변수의 크기를 넘어서는 경우와 같이 값을 변환하거나 할당할 때 오류가 발생할 경우
ZERO_DIVIDE	ORA-01476	-1476	제수가 0일 때 발생

```

declare
  counter integer;
begin
  counter := 10;
  counter := counter /0;
  dbms_output.put_line(counter);
exception when zero_divide then
  dbms_output.put_line('errors');
end;

```

ZERO\_DIVIDE 예외 발생 시 1로 나누도록 로직 처리

```

declare

```

```
counter integer;
begin
    counter := 10;
    counter := counter /0;
    dbms_output.put_line(counter);
exception when zero_divide then
    counter := counter /1;
    dbms_output.put_line(counter);
end;
```

#### 4. PL/SQL의 구성요소

##### 1) 변수와 상수

- 변수 선언

```
emp_num1 number(9);
grade char(2);
emp_num2 integer := 1;
```

- 상수 선언

```
nYear constant integer := 30; (o)
nYear constant integer; (x)
```

- %TYPE

%TYPE : 참조할 테이블에 있는 컬럼의 데이터 타입을 자동으로 가져옴.

[구문형식]

변수명 테이블명.컬럼명%TYPE

```
nSal emp.sal%type;
```

- %ROWTYPE

%TYPE은 하나의 값에 대해 적용되지만, %ROWTYPE은 하나 이상의 값에 대해 적용.  
로우타입 변수를 선언해서 테이블에 있는 로우를 대입할 수 있다.(콜렉션, OBJECT 타입  
변수에서만 사용 가능)

## 2) 콜렉션

- 콜렉션의 종류

### (1) varray

varray는 variable array의 약자로 고정 길이(fixed number)를 가진 배열

TYPE 타입명 IS { VARRAY | VARYING ARRAY } (크기) OF 요소데이터타입 [NOT NULL];

```
declare
  type varray_test is varray(3) of integer;
  varray1 varray_test; -- 위에서 선언한 varray_test 타입 변수
begin
  varray1 := varray_test(10,20,30);

  dbms_output.put_line(varray1(1));
end;
```

### (2) 중첩 테이블

중첩 테이블은 varray와 흡사하지만 중첩 테이블은 선언 시에 전체 크기를 명시할 필요가 없음

TYPE 타입명 IS TABLE OF 요소데이터 타입 [NOT NULL];

```
declare
  type nested_test is table of varchar2(10);
  nested1 nested_test; -- 위에서 선언한 nested_test 타입 변수
begin
  nested1 := nested_test('A','B','C','D');

  dbms_output.put_line(nested1(2));
end;
```

### (3) Associative array(index-by table)

Associative array(연관배열)는 index-by table이라고도 하며 키와 값의 쌍으로 구성된 콜렉션으로, 하나의 키는 하나의 값과 연관되어 있다.

```
TYPE 타입명 IS TABLE OF 요소 데이터 타입 [NOT NULL]
INDEX BY [PLS_INTEGER | BINARY_INTEGER | VARCHAR2(크기)]
INDEX BY 키타입;
```

```
declare
  type assoc_array_num_type is table of number index by pls_integer;
  -- 키는 pls_integer 형이며, 값은 number형인 요소들로 구성
  -- (주의) 키에 pls_integer 형 대신 integer를 사용하면 오류 발생
  assoc1 assoc_array_num_type; -- 위에서 선언한 assoc_array_num_type 타입 변수
begin
  assoc1(3) := 33;--assoc_array_num_type의 키는 3, 값은 33을 넣는다.

  dbms_output.put_line(assoc1(3));
end;

-----
declare
  type assoc_array_str_type is table of varchar2(32) index by pls_integer;
  -- 키는 pls_integer 형이며, 값은 varchar2(32)형인 요소들로 구성
  assoc2 assoc_array_str_type; -- 위에서 선언한 assoc_array_str_type 타입 변수
begin
  assoc2(2) := 'TT';--assoc_array_str_type의 키는 2, 값은 TT를 넣는다.

  dbms_output.put_line(assoc2(2));
end;

-----
declare
  type assoc_array_str_type2 is table of varchar2(32) index by varchar2(64);
  -- 키는 varchar2(64) 형이며, 값은 varchar2(32)형인 요소들로 구성
  assoc3 assoc_array_str_type2; -- 위에서 선언한 assoc_array_str_type2 타입 변수
begin
  assoc3('K') := 'KOREA';--assoc_array_str_type2의 키는 K, 값은 KOREA를 넣는다.

  dbms_output.put_line(assoc3('K'));
end;
```

#### (4) 콜렉션을 데이터베이스 객체로 생성

```
create type alphabet_type as varray(26) of varchar2(2);

-----
declare
  test_alpha alphabet_type;
begin
```

```
test_alpha := alphabet_type('A','B','C','D');
dbms_output.put_line(test_alpha(1));
end;
```

### 3) 레코드

콜렉션에 해당하는 **varray**, 중첩 테이블, **associative array**는 모든 프로그래밍 언어에서 사용하는 배열 형태의 구조를 가진다. 즉 이들은 구성하는 요소들의 데이터 타입은 모두 같아야 한다. 반면 테이블의 컬럼들이 서로 다른 유형의 데이터 타입으로 구성되듯이 레코드 역시 해당 필드(레코드에서는 요소란 말 대신 필드란 용어를 사용한다.)들이 각기 다른 데이터 타입을 가질 수 있다. 프로그래밍 언어와 비교하면 레코드는 구조체에 해당한다고 볼 수 있다.

[구문형식]

```
TYPE 레코드이름 IS RECORD (필드1 데이터타입1, 필드2 데이터타입2, ...);
```

[구문형식]

```
레코드이름 테이블명%ROWTYPE;
```

[구문형식]

```
레코드이름 커서명%ROWTYPE;
```

```
declare
    --TYPE으로 선언한 레코드
    type record1 is record (deptno number not null :=50,
                           dname varchar2(14),
                           loc varchar2(13)
                           );
    --위에서 선언한 record1을 받는 변수 선언
    rec1 record1;
begin
    --record1 레코드 변수 rec1의 dname 필드에 값 할당.
    rec1.dname :='RECORD';
    rec1.loc :='Seoul';

    --rec1 레코드 값을 dept 테이블에 insert
    insert into dept values rec1;

    commit;
exception when others then
    rollback;
end;
```

## 5. PL/SQL 문장과 커서

### 1) IF문

[처리 조건이 한 개일 경우]

IF 조건 THEN

  처리문;

END IF;

[처리 조건이 두 개일 경우]

IF 조건 THEN

  처리문1;

ELSE

  처리문2;

END IF;

[처리 조건이 여러 개일 경우]

IF 조건 THEN

  처리문1;

ELSIF 조건2 THEN

  처리문2;

  ...

ELSE

  처리문n;

END IF;

예)

```
declare
  grade char(1);
begin
  grade := 'B';

  if grade = 'A' then
    dbms_output.put_line('Excellent');
  elsif grade = 'B' then
    dbms_output.put_line('Good');
  elsif grade = 'C' then
    dbms_output.put_line('Fair');
  elsif grade = 'D' then
    dbms_output.put_line('Poor');
  end if;
```

```
end;
```

## 2) CASE문

```
declare
grade char(1);
begin
grade := 'B';

case grade
when 'A' then
  dbms_output.put_line('Excellent');
when 'B' then
  dbms_output.put_line('Good');
when 'C' then
  dbms_output.put_line('Fair');
when 'D' then
  dbms_output.put_line('Poor');
else
  dbms_output.put_line('Not Found');
end case;

end;
```

## 3) LOOP문

[구문형식]

```
LOOP
  처리문장들..
END LOOP;
```

예)

```
declare
  test_number integer;
  result_num integer;
begin
  test_number :=1;
```

```
loop
```

```

result_num := 2 * test_number;
if result_num > 20 then
    exit; --블록 종료
else
    dbms_output.put_line(result_num);
end if;
test_number := test_number + 1;
end loop;
end;
-----
```

```

declare
    test_number integer;
    result_num integer;
begin
    test_number :=1;

    loop
        result_num := 2 * test_number;

        exit when result_num > 20;

        dbms_output.put_line(result_num);
        test_number := test_number + 1;
    end loop;
end;
```

#### 4) WHILE-LOOP문

WHILE 조건 LOOP  
 처리문장들;  
 END LOOP;

예)  
 declare
 test\_number integer;
 result\_num integer;
 begin
 test\_number :=1;
 **result\_num :=0;**
**while result\_num < 20 loop**

```
result_num := 2 * test_number;
dbms_output.put_line(result_num);
test_number := test_number + 1;
end loop;
end;
```

## 5) FOR .. LOOP

[구문형식]  
FOR 카운터 IN[REVERSE] 최소값..최대값 LOOP  
처리문장들;  
END LOOP;

예)  
declare  
test\_number integer;  
result\_num integer;  
begin  
test\_number :=1;  
result\_num :=0;  
  
for test\_number in 1..10 loop  
result\_num := 2 \* test\_number;  
dbms\_output.put\_line(result\_num);  
end loop;  
end;

---

-----  
declare  
test\_number integer;  
result\_num integer;  
begin  
  
for test\_number in reverse 1..10 loop  
  
result\_num := 2 \* test\_number;  
dbms\_output.put\_line(result\_num);  
end loop;  
end;

## 6) GOTO문

```

declare
    test_number integer;
    result_num integer;
begin
    test_number :=1;
    result_num :=0;

    goto second;
    dbms_output.put_line('<<first>>');
    <<first>>
    for test_number in 1..10 loop
        result_num := 2 * test_number;
        dbms_output.put_line(result_num);
    end loop;

    dbms_output.put_line('<<second>>');

    result_num :=0;
    <<second>>
    for test_number in reverse 1..10 loop

        result_num := 2 * test_number;
        dbms_output.put_line(result_num);
    end loop;
end;

```

## 7) 커서

SELECT 문장을 실행하면 조건에 따른 결과가 추출된다. 추출되는 결과는 한 건이 될 수도 있고 여러 건이 될 수도 있으므로 이를 결과 셋(result set) 혹은 결과집합이라고 부르기도 한다. 쿼리에 의해 반환되는 결과는 메모리 상에 위치하게 되는데 PL/SQL에서는 바로 커서(cursor)를 사용하여 이 결과집합에 접근할 수 있다. 즉 커서를 사용하면 결과집합의 각 개별 데이터에 접근이 가능하다.

[커서 선언 : 커서에 이름을 주고, 이 커서가 접근하려는 쿼리를 정의]

CURSOR 커서명 IS  
    SELECT 문장;

[커서 열기(open): 커서로 정의된 쿼리를 실행하는 역할을 한다.]

OPEN 커서명;

[패치(fetch) : 쿼리의 결과에 접근한다.]

FETCH 커서명 INTO 변수 ...;

[커서 닫기(close) : 패치 작업이 끝나면 사용된 커서를 닫는다.]

CLOSE 커서명;

예)

```
declare
    cursor emp_csr is
        select empno
        from emp
        where deptno = 10;
```

```
    emp_no emp.empno%type;
```

```
begin
```

```
    open emp_csr;
```

```
loop
```

```
    fetch emp_csr into emp_no;
```

--%notfound : 커서에서만 사용 가능한 속성.

--더 이상 패치(할당)할 로우가 없음을 의미

```
exit when emp_csr%notfound;
```

```
    dbms_output.put_line(emp_no);
```

```
end loop;
```

```
close emp_csr;
```

```
end;
```

커서의 속성

속성	정의
%FOUND	PL/SQL 코드가 마지막으로 얻은 커서의 결과 SET에 레코드가 있다면 참
%NOTFOUND	%FOUND 연산자와 반대의 의미
%ROWCOUNT	커서에서 얻은 레코드의 수를 반환
%ISOPEN	커서가 열렸고, 아직 닫히지 않은 상태라면 참

## 6. PL/SQL 서브프로그램

PL/SQL 서브프로그램은 파라미터와 고유의 이름을 가진 PL/SQL 블록을 말하며, 데이터베이스 객체로 존재한다. 즉 필요할 때마다 호출해서 사용할 수 있다는 의미이다. 이러한 서브프로그램에는 내장 프로시저(stored procedure)와 함수(function)가 있다.

### 1) 함수

## [구문형식]

입력 받은 값으로부터 10%의 세율을 얹는 함수  
create or replace function tax (p\_value in number)

return number

is

10

begin  
return

and

end;

```
select tax(100) from dual;
```

급여와 커미션을 합쳐서 세금 계산

```
create or replace function tax2 (
```

p\_sal in emp.sal%type,

p\_comm emp.comm%type

)

return number

is

begin

```
return ((p_sal + nvl(p_comm.0))*0.1);
```

end.

```
select empno,ename,sal,comm, tax2(sal,comm) as tax  
from emp;
```

급여(보너스 포함)에 대한 세율을 다음과 같이 정의함. 급여가 월 \$1,000보다 적으면 세율을

5% 적용하며, \$1,000 이상 \$2,000 이하면 10%, \$2,000을 초과하면 20%를 적용함

```
create or replace function tax3 (
```

p\_sal in emp.sal%type,

p\_comm emp.comm%type

)

```
return number
is
l_sum number;
l_tax number;
begin
l_sum := p_sal + nvl(p_comm,0);

if l_sum < 1000 then
    l_tax := l_sum * 0.05;
elsif l_sum <= 2000 then
    l_tax := l_sum * 0.1;
else
    l_tax := l_sum * 0.2;
end if;

return (l_tax);
end;
```

```
select empno,ename,sal,com, tax3(sal,comm) as tax
from emp;
```

---

```
create or replace function emp_salaries (emp_no number)
    return number is
    nSalaries number(9);
begin
    nSalaries :=0;
    select sal
        into nSalaries
        from emp
        where empno = emp_no;

    return nSalaries;
end;
```

```
select emp_salaries(7839) from dual;
```

```
select empno, ename, emp_salaries(empno) from emp where deptno=10;
```

---

```
create or replace function get_dept_name(dept_no number)
```

```
return varchar2 is  
  
    sDeptName varchar2(30);  
begin  
    select dname  
    into sDeptName  
    from dept  
    where deptno = dept_no;  
  
    return sDeptName;  
end;  
  
select get_dept_name(10) from dual
```

- 생성된 함수 확인하기

데이터 사전(Data Dictionary)을 통해 검색. 데이터 사전에 저장된 모든 값은 대문자로 저장되기 때문에 대문자로 검색

```
select object_name,object_type  
from user_objects  
where object_type='FUNCTION'  
and object_name='TAX';
```

- 작성된 함수의 소스 코드 확인

```
select text  
from user_source  
where type='FUNCTION' and name='TAX';
```

2) 프로시저

[구문형식]  
CREATE OR REPLACE PROCEDURE 프로시저명  
 (파라미터1 데이터타입,  
 ....  
 )  
IS [AS]  
 변수 선언부...;  
BEGIN

```
프로시저 본문처리...;

EXCEPTION
    예외처리...;
END;

예)
create or replace procedure hello_world
is
    message VARCHAR2(100);
begin
    message := 'Hello World!';
    dbms_output.put_line(message);
end;
```

프로시저 실행  
[구문형식]  
EXEC 혹은 EXECUTE 프로시저명(파라미터...);

```
exec hello_world;
-----
create or replace procedure hello_world
is
    message VARCHAR2(100) := 'HELLO WORLD';
begin
    dbms_output.put_line(message);
end;
```

```
exec hello_world;
-----
create or replace procedure hello_world(p_message IN VARCHAR2)
is
begin
    dbms_output.put_line(p_message);
end;

exec hello_world('Korea');
```

작성된 Stored Procedure 확인

```
SELECT object_name, object_type
```

```
FROM user_objects  
WHERE object_type = 'PROCEDURE';
```

Stored Procedure의 Source를 데이터 사전을 이용해서 얻음

```
SELECT text  
FROM user_source  
WHERE name = 'HELLO_WORLD';
```

\* 테이블이나 **Stored Procedure**의 이름이 같으면 충돌이 일어난다. 따라서 이름을 생성할 때는 구별할 수 있도록 이름 뒤에 타입을 나타내는 접미사를 붙여두는 것도 좋은 방법.

```
create or replace procedure add_department(
    p_deptno      IN dept.deptno%TYPE,
    p_dname       IN dept.dname%TYPE,
    p_loc         IN dept.loc%TYPE)
is
begin
--parameter 변수에 입력받은 값으로
--부서(dept) 테이블의 각 컬럼에 데이터를 추가
--그리고 정상적으로 Transaction을 종료
insert into dept
values (p_deptno,p_dname,p_loc);
commit;
exception when others then
    dbms_output.put_line(p_dname || ' register is failed!');
    rollback;
end;

exec add_department(60,'IT','BUSAN');
```

```
insert into emp(empno, ename, job, mgr, hiredate, sal, comm, deptno)
values(e_no,e_name, j_work, j_mgr, sysdate,j_sal, j_comm,j_deptno);
```

```
commit;
```

```
exception when others then
dbms_output.put_line(e_name || ' register is failed!');
rollback;
end;
```

프로시저 실행

[구문형식]

EXEC 혹은 EXECUTE 프로시저명(파라미터...);

```
-----  
execute register_emp(9000,'PETER','MANAGER',7788,6000,200,10);  
-----
```

```
create or replace procedure output_department(
    p_dept_no      IN dept.deptno%TYPE)
is
--local 변수 선언 -----
    l_dname dept.dname%TYPE;
    l_loc   dept.loc%TYPE;
begin
--parameter 변수로부터 부서 아이디를 받아 해당 부서의 정보 질의
select dname,loc
into l_dname,l_loc
from dept
where deptno = p_dept_no;
dbms_output.put_line(l_dname || ' ' || l_loc);
end;
```

-----  
실행

```
exec output_department(10);  
-----
```

아래 프로시저는 등록은 되지만 실행해서 값을 넘기면 오류가 발생할 수 있음.

이유는 select문의 실행결과를 저장할 수 있는 변수는 하나의 값만 저장 가능.

그러나 인출되는 즉, 질의의 수행 결과 반환되는 행의 수가 하나보다 많을 때는 선언한 변수에 값을 저장할 수 없다는 메시지를 띄울 수 있음. 로컬 변수를 선언하여 INTO 절에서 처리할 수 있는 행의 수는 하나의 행만이 가능함.

```
create or replace procedure info_Hiredate
```

```

(p_year IN VARCHAR2)
is
--%ROWTYPE으로 데이터 타입이 지정되어 있는 사원 테이블(emp)의 하나의 행이 가지는
모든 컬럼의 데이터 타입을 가져옴
l_emp emp%ROWTYPE;

begin
  select empno, ename, sal
  into l_emp.empno,l_emp.ename,l_emp.sal
  from emp
  where to_char(hiredate, 'YYYY') = p_year;

  dbms_output.put_line(l_emp.empno || ' ' || l_emp.ename || ' ' || l_emp.sal);
end;

```

실행

여러개의 행이 반환되어 에러발생  
 exec info\_Hiredate('1981');  
 하나의 행이 반환되어 에러가 발생하지 않음  
 exec info\_Hiredate('1980');

커서 이용하기

커서(Cursor)를 이용하여 질의 수행 결과 반환되는 여러 행을 처리

```

create or replace procedure info_Hiredate
(p_year IN VARCHAR2)
is
--%ROWTYPE으로 데이터 타입이 지정되어 있는 사원 테이블(emp)의 하나의 행이 가지는
모든 컬럼의 데이터 타입을 가져옴
l_emp emp%ROWTYPE;
--커서의 선언
--CURSOR cursor_name is select statement;
CURSOR emp_cur is
  select empno,ename,sal
  from emp
  where to_char(hiredate,'YYYY') = p_year;
begin
  -- 커서 열기
  -- OPEN cursor_name;
  OPEN emp_cur;

```

```

--커서로부터 데이터 읽기
--FETCH cursor_name INTO local variables;
LOOP
  FETCH emp_cur INTO l_emp.empno,l_emp.ename,l_emp.sal;
  EXIT WHEN emp_cur%NOTFOUND;
  dbms_output.put_line(l_emp.empno || ' ' || l_emp.ename || ' ' || l_emp.sal);
END LOOP;
--커서 닫기
--CLOSE cursor_name;
CLOSE emp_cur;
end;

exec info_Hiredate('1981');

-----
'SALES' 부서에 속한 사원의 정보 보기
create or replace procedure emp_info
  (p_dept dept.dname%type)
is
cursor emp_cur is
  select empno,ename
  from emp e join dept d
  on e.deptno = d.deptno
  where dname = upper(p_dept);

l_emp_no emp.empno%type;
l_emp_name emp.ename%type;
begin
  open emp_cur;
  loop
    fetch emp_cur into l_emp_no,l_emp_name;
    exit when emp_cur%notfound;
    dbms_output.put_line(l_emp_no || ' ' || l_emp_name);
  end loop;
  close emp_cur;
end emp_info;

exec emp_info('SALES');

```

### 3) 패키지(Package)

패키지는 업무와 관련된 Stored Procedure 및 Stored Function을 관리하고, 이를 패키지 단위로 배포할 때 유용하게 사용됨.

패키지는 선언부와 본문으로 구분. 선언부는 패키지에 포함될 Stored Procedure 또는 Stored Function을 포함.

패키지 선언부

```
create or replace package emp_pack is
procedure empno;
procedure emp_salary;
end emp_pack;
```

패키지 본문

```
create or replace package body emp_pack is
procedure empno
is
begin
Statement; ← 코드 넣기
end empno;

procedure emp_salary is

begin
Statement;← 코드 넣기
end emp_salary;

end emp_pack;

exec emp_pack.emp_salary;
```

예)

선언부 생성

```
create or replace
package employee_pkg as
procedure print_ename(p_empno number);
procedure print_sal(p_empno number);
end employee_pkg;
```

본문 생성

```

create or replace
package body employee_pkg as

procedure print_ename(p_empno number) is
  l_ename emp.ename%type;
begin
  select ename
  into l_ename
  from emp
  where empno = p_empno;
  dbms_output.put_line(l_ename);
exception
  when NO_DATA_FOUND then
    dbms_output.put_line('Invalid employee number');
end print_ename;

procedure print_sal(p_empno number) is
  l_sal emp.sal%type;
begin
  select sal
  into l_sal
  from emp
  where empno = p_empno;
  dbms_output.put_line(l_sal);
exception
  when NO_DATA_FOUND then
    dbms_output.put_line('Invalid employee number');
end print_sal;
end employee_pkg;

```

실행

```

execute employee_pkg.print_ename(7369);
SMITH

execute employee_pkg.print_sal(7369);
800

```

#### 4) 트리거(Trigger)

트리거는 데이터의 변경(INSERT,DELETE,UPDATE)문이 실행될 때 자동으로 같이 실행되는 프로시저를 말한다. 오라클은 기본적으로 실행 전 BEFORE 과 실행 후 AFTER 트리거를 지원한다.