

# PL/SQL

강사 : 강병준

# PL/SQL 이란 ?

- **PL/SQL(Oracle's Procedural Language extension to SQL)**

- ← **정의**

- ☒ SQL 언어를 사용한 데이터 조작과 질의문 등을 블록 구조 안에 절차적 단 위의 코드로 포함하여 절차적 프로그래밍을 가능하게 한 강력한 트랜잭션 처리 언어

- ← **특징**

- ☒ SQL 문장에서 변수 정의, 조건 처리(IF), 반복 처리(LOOP, WHILE, FOR) 등을 지원
- ☒ 오라클 자체에 내장되어 있는 절차적 언어
- ☒ DECLARE문을 이용하여 정의
- ☒ 블록 구조로 되어 있고, PL/SQL 자신이 컴파일 엔진을 가짐

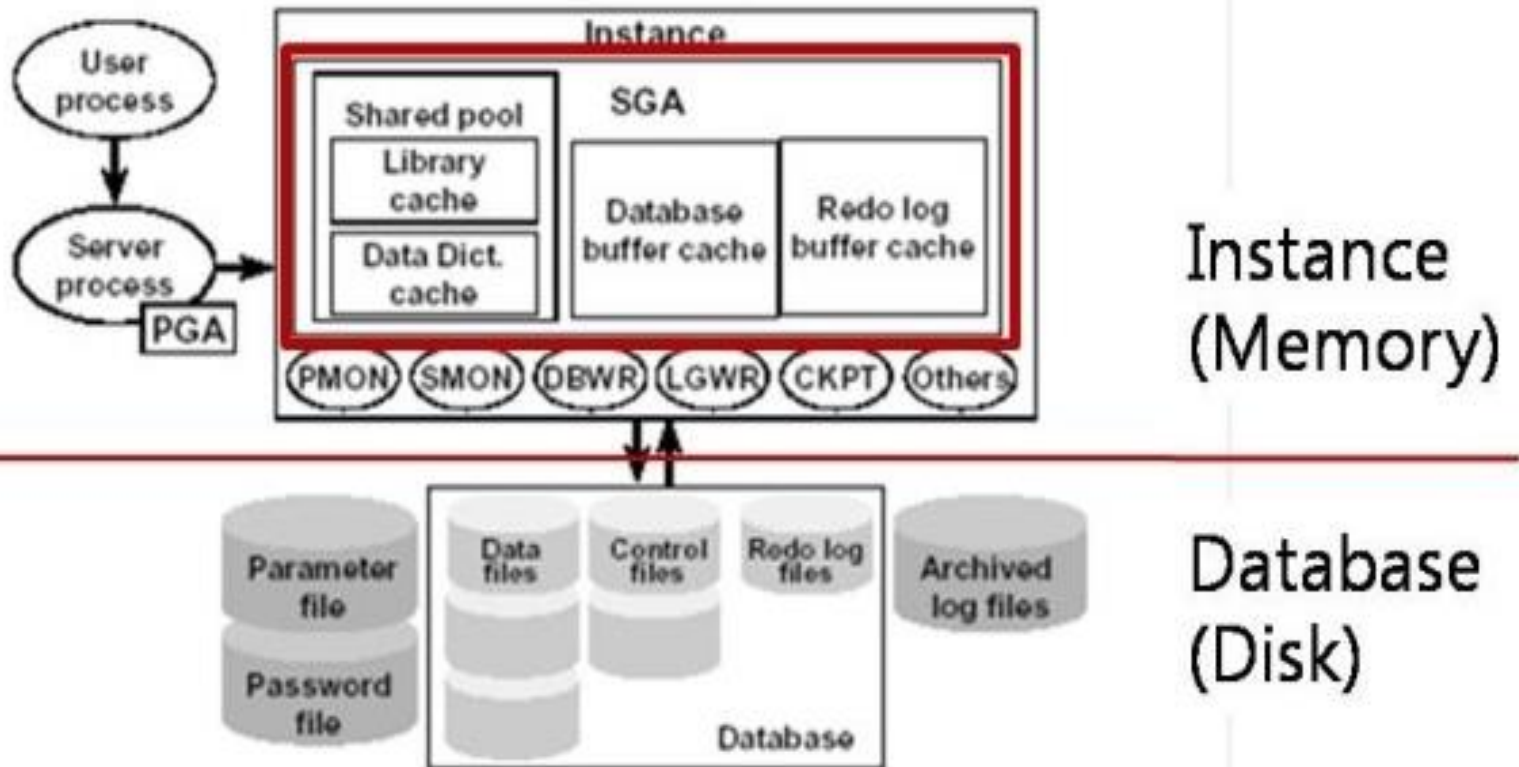
# Oracle SGA란?

- SGA는 오라클서버의 메모리영역 ( 공유 메모리 영역 )
- SGA는 Oracle의 인스턴스에 대한 데이터와 제어 정보를 가지는 공유 메모리 영역의 집합
- 사용 목적의 따라 오라클 파라미터 파일(init[SID].ora)에서 SGA의 각 부분의 크기를 조절 가능 ( 오라클 서버의 종료 없이 SGA의 구성을 SGA\_MAX\_SIZE 파라미터 값 범위 내에서만!! 9i 이상부터 가능 )
- Oracle 서버를 사용하고 있는 사용자는 시스템 글로벌 영역에서 데이터를 공유
- SGA 메모리가 클수록 메모리에 많은 데이터를 넣어 놓을 수 있으므로 성능은 좋아질 수 있음
- SGA는 공유 풀(Shared Pool), 데이터베이스 버퍼캐쉬(DataBase Buffer Cache), 리두로그 버퍼(Redo Log Buffer) 이 세가지 와 LARGE POOL과 JAVA POOL로 구성되어 있습니다.

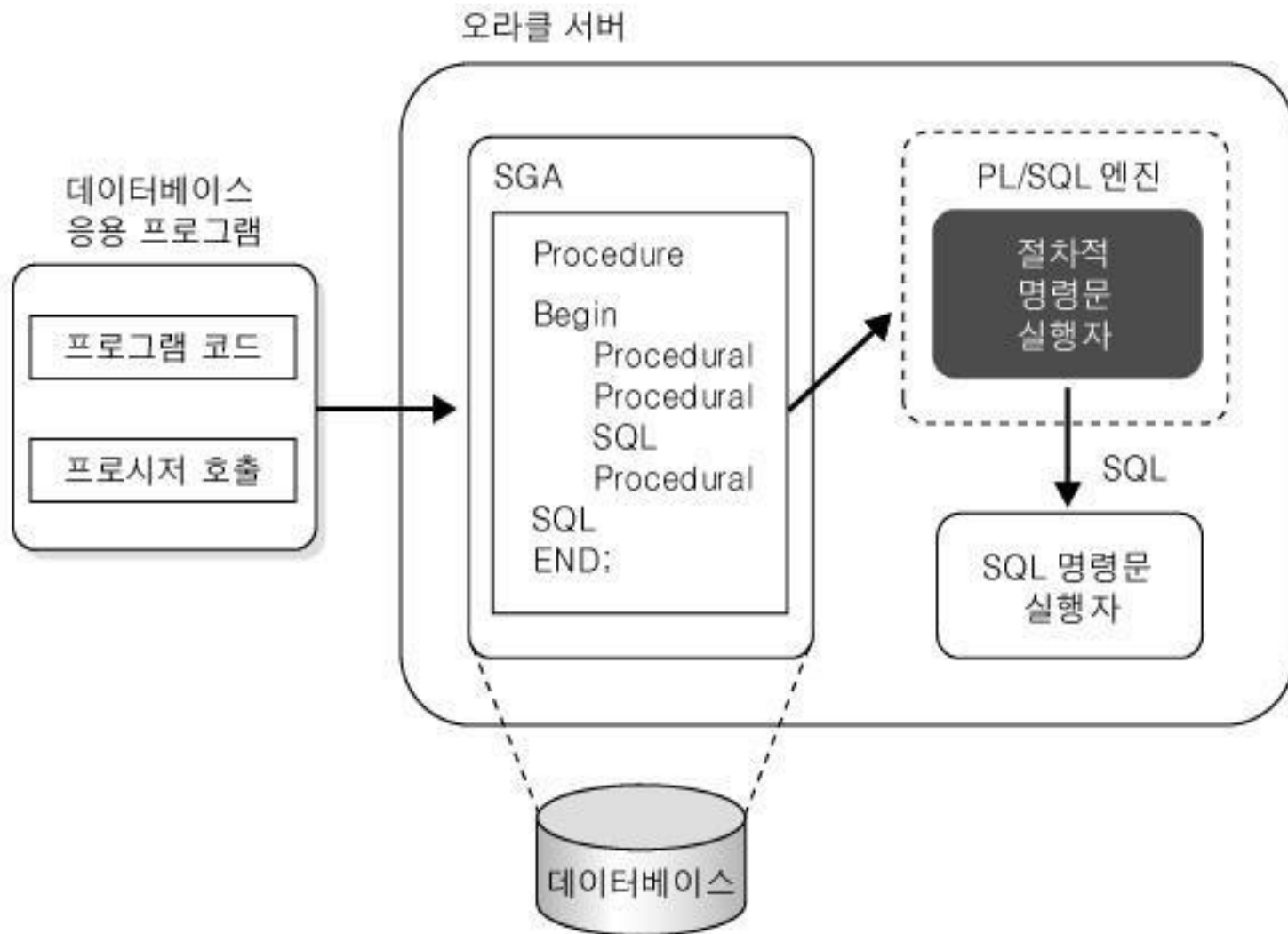
# Oracle SGA란?

Oracle Server = Instance + Database

## Overview of Primary Components



# 오라클 PL/SQL의 물리적 구조



# PL/SQL이 제공하는 기능과 장점

## 기 능

블록 내에서 논리적으로 관련된 문장들의 그룹화하여 모듈화된 프로그램 개발 가능

절차적 언어 구조로 된 프로그램 작성 가능

- 조건에 따라 일련의 문장을 실행(IF)
- 루프에서 반복적으로 일련의 문장을 실행(LOOP)
- 명시적 커서(Explicit Cursor)를 이용한 다중 행(Multi-row) 질의 처리 가능

데이터베이스의 테이블 구조와 컬럼을 기반으로 하는 동적인 변수 선언 가능

예외(Exception) 처리 루틴을 이용하여 에러 처리, 사용자 정의 에러를 선언하고 예외(Exception) 처리 루틴으로 처리 가능

PL/SQL은 블록 구조로 다수의 SQL 문을 한번에 오라클 데이터베이스로 보내서 처리하므로 응용 프로그램의 성능 향상 가능.

PL/SQL은 오라클에 내장되어 있으므로 오라클과 PL/SQL을 지원하는 어떤 호스트에도 프로그램 이식(사용) 가능.

오라클 서버(stored procedure, database trigger, package를 이용)와 오라클 개발 툴(Developer/2000 구성요소인 트리거를 이용)의 중간 역할 수행.

# Block Type(PL/SQL 블록의 유형)

- ◆ Anonymous Block(익명 블록)  
이름이 없는 블록을 의미합니다.  
실행하기 위해 프로그램 안에서 선언되고 실행 시에 실행을 위해 PL/SQL 엔진으로 전달됩니다.  
선행 컴파일러 프로그램과 SQL\*Plus 또는 서버 관리자에서 익명의 블록을 내장할 수 있습니다.
- ◆ Stored Procedure(프로시저)  
특정 작업을 수행할 수 있는 이름이 있는 PL/SQL 블록으로서.  
매개 변수를 받을 수 있고 반복적으로 사용할 수 있습니다.  
보통 연속실행 또는 구현이 복잡한 트랜잭션을 수행하는 PL/SQL 블록을 데이터 베이스에 저장하기 위해 생성합니다.
- ◆ Function(함수)  
보통 값을 계산하고 결과값을 반환하기 위해서 함수를 많이 사용합니다.  
대부분 구성이 프로시저와 유사하지만,  
반드시 반환될 값의 데이터 타입을 RETURN문에 선언해야 합니다.  
또한 PL/SQL 블록 내에서 RETURN문을 통해서 반드시 값을 반환해야 합니다.
- ◆ 패키지(Package)  
관련된 프로시저, 함수, 식별자 등을 모두 모은 이름이 있는 PL/SQL 모듈
- ◆ 데이터베이스 트리거(Database Triggers)  
데이터베이스와 연결되어 자동으로 실행되는 이름있는 PL/SQL 블록



# Block Type(PL/SQL 블록의 유형)

[ DECLARE ]

BEGIN

-- statements

[ EXCEPTION ]

END ;

[ Anonymous ]

PROCEDURE name  
IS

BEGIN

-- statements

[ EXCEPTION ]

END ;

[ Procedure ]

FUNCTION name  
RETURN datatype  
IS

BEGIN

-- statements

RETURN value;

[ EXCEPTION ]

END ;

[ Function ]



# PL/SQL Block Structure

- PL/SQL은 프로그램을 논리적인 블록으로 나누는 구조화된 블록 언어입니다.
  - PL/SQL 블록은 선언부(선택적), 실행부(필수적), 예외 처리부(선택적)로 구성되어 있고, BEGIN과 END 키워드는 반드시 기술해 주어야 합니다.
  - PL/SQL 블록에서 사용하는 변수는 블록에 대해 논리적으로 선언할 수 있고 사용할 수 있습니다.
- 
- ◆ Declarative Section(선언부)
    - 변수, 상수, CURSOR, USER\_DEFINE Exception 선언
  - ◆ Executable Section(실행부)
    - SQL, 반복문, 조건문 실행
    - 실행부는 BEGIN으로 시작하고 END 로 끝납니다.
    - 실행문은 프로그램 내용이 들어가는 부분으로서 필수적으로 사용되어야 합니다.
  - ◆ Exception Handling Section(예외처리)
    - 예외에 대한 처리.
    - 일반적으로 오류를 정의하고 처리하는 부분으로 선택 사항입니다.

# PL/SQL Block Structure

- DECLARE
  - Optional
  - Variables, cursors, user-defined exceptions
- BEGIN
  - Mandatory
  - SQL Statements
  - PL/SQL Statements
- EXCEPTION
  - Actions to perform when errors occur
- END;
  - Mandatory



# PL/SQL 프로그램의 작성 요령

- PL/SQL 블록내에서는 한 문장이 종료할 때마다 세미콜론(;)을 사용합니다.
- END뒤에 ;을 사용하여 하나의 블록이 끝났다는 것을 명시 합니다.
- PL/SQL 블록의 작성은 편집기를 통해 파일로 작성할 수도 있고, SQL프롬프트에서 바로 작성할 수도 있습니다.
- SLQ\*PLUS환경에서는 DELCLARE나 BEGIN이라는 키워드로 PL/SQL블록이 시작하는 것을 알 수 있습니다.
- 단일행 주석 : --
- 여러행 주석 : /\* \*/
- PL/SQL 블록은 행에 / 가 있으면 종결됩니다.

```
FUNCTION sal_evl ( v_emp_no NUMBER )  
    RETURN NUMBER  
IS  
    v_salary    NUMBER;  
BEGIN -- PL/SQL 블록 시작  
    SELECT salary INTO v_salary  
        WHERE v_emp_no =  
    RETURN ( v_salary );  
END sal_evl; -- PL/SQL 블록 끝
```

```
FUNCTION sal_evl ( v_emp_no NUMBER )  
    RETURN NUMBER  
IS  
    v_salary    NUMBER;  
BEGIN  
/* 이 기호는 한 줄 이상의 주석을 작성하는 경우  
   사용하는 기호이다. */  
    SELECT salary INTO v_salary FROM salary  
        WHERE v_emp_no = emp_no;  
    RETURN ( v_salary );  
END sal_evl;
```

# 식별자

## ● 정의

- 식별자(Identifier)는 변수, 커서, 타입 등의 PL/SQL 개체를 명명하기 위해 사용.

## ● 특징

- 식별자의 첫 번째 문자는 알파벳으로 시작한다.
- 식별자의 최대 길이는 30자 이내다.
- 식별자에 &, -, /, 공백을 제외한 특수문자가 사용가능하다.
- 대소문자를 구별하지 않는다.
- 예약어(Reserved Word)를 식별자로 사용할 수 없다.
- 식별자에 따옴표를 사용할 수 있고, 이 경우에는 공백, /, &, - 등의 문자와 예약어를 포함하는 이름을 사용할 수 있다. 또한 따옴표를 사용하는 경우는 대소문자를 구별한다.

# 변수선언과 대입문

- PL/SQL의 선언부에서는 실행부에서 사용할 변수를 선언합니다. 변수를 선언할 때 변수명 다음에 자료형을 기술해야 합니다.
- PL/SQL에서 변수 선언할 때 사용되는 자료형은 SQL에서 사용하던 자료형과 거의 유사합니다.

*identifier* [CONSTANT] *datatype* [NOT NULL]  
[:= | DEFAULT *expression*];

구문	설명
identifier	변수의 이름
CONSTANT	변수의 값을 변경할 수 없도록 제약합니다.
datatype	자료형을 기술합니다.
NOT NULL	값을 반드시 포함하도록 하기 위해 변수를 제약합니다.
Expression	Literal, 다른 변수, 연산자나 함수를 포함하는 표현식

# 변수선언과 대입문

- 쿼리문을 수행하고 난 후에 얻어진 결과를 컬럼 단위로 변수에 저장할 경우 다음과 같이 선언합니다.

```
VEMPNO NUMBER(4);  
VENAME VARCHAR2(10);
```

- PL/SOL에서 변수를 선언할 때 위와 같이 SQL에서 사용하던 자료형과 유사하게 선언하는 것을 스칼라(SCALAR) 변수라고 합니다.
- 숫자를 저장하기 위해서 VEMPNO 변수는 NUMBER로 선언하고 VENAME 변수는 문자를 저장하려면 VARCHAR2를 사용해서 선언하였습니다.

# 대입문으로 변수에 값 지정하기

- PL/SQL에서는 변수의 값을 지정하거나 재지정하기 위해서 :=를 사용합니다. := 의 좌측에 새 값을 받기 위한 변수를 기술하고 우측에 저장할 값을 기술합니다.

***identifier := expression;***

- 선언부에서 선언한 변수에 값을 할당하기 위해서는 :=를 사용해 봅시다.

```
VEMPNO := 7788;  
VENAME := 'SCOTT';
```



# pl-sql 기초

set serveroutput on

-----

DECLARE

    vi\_num NUMBER;

BEGIN

    vi\_num := 100;

    DBMS\_OUTPUT.PUT\_LINE(vi\_num);

END;

/

# pl-sql 기초

-- 연산자

DECLARE

  a INTEGER := 2\*\*2\*3\*\*2;

BEGIN

  DBMS\_OUTPUT.PUT\_LINE('a = ' || TO\_CHAR(a));

END;

/

# pl-sql 기초

-- 주석

DECLARE

-- 한 줄 주석, 변수선언

a INTEGER := 2\*\*2\*3\*\*2;

BEGIN

/\* 실행부

DBMS\_OUTPUT을 이용한 변수값 출력

\*/

DBMS\_OUTPUT.PUT\_LINE('a = ' || TO\_CHAR(a));

END;

/

# pl-sql 기초

```
set serveroutput on
```

```
-----
```

```
declare
```

```
    t1 char(3);
```

```
begin
```

```
    t1 := '&t2' ;
```

```
    dbms_output.put_line(t1);
```

```
exception
```

```
    when others then
```

```
        dbms_output.put_line('you are wrong');
```

```
end;
```

```
/
```

# pl-sql 기초

-- DML 문

DECLARE

vs\_emp\_name VARCHAR2(80); -- 사원명 변수

vs\_dep\_name VARCHAR2(80); -- 부서명 변수

BEGIN

SELECT a.ename, b.dname

INTO vs\_emp\_name, vs\_dep\_name

FROM emp a,

dept b

WHERE a.deptid = b.deptid

AND a.empid = 100;

DBMS\_OUTPUT.PUT\_LINE( vs\_emp\_name || ' - ' || vs\_dep\_name);

END;

/

# pl-sql 기초

DECLARE

vs\_first\_name employees.emp\_name%TYPE;

vs\_dep\_name departments.department\_name%TYPE;

BEGIN

SELECT a.first\_name , b.department\_name

INTO vs\_first\_name , vs\_dep\_name

FROM employees a,

departments b

WHERE a.department\_id = b.department\_id

AND a.employee\_id = 100;

DBMS\_OUTPUT.PUT\_LINE( vs\_emp\_name || ' - ' || vs\_first\_name );

END;

/

# sql문과 변수 사용

```
declare
  s1 number(4);
begin
  s1 := &t2 ;
  select empno into s1
  from emp
  where empno = s1;
  dbms_output.put_line(s1 || '중복 아이디');
exception
  when others then
    dbms_output.put_line('good ID');
end;
/
```



# 변수 값 할당

```
declare
```

```
    s1 number :=50;
```

```
begin
```

```
    s1 :=100;
```

```
    dbms_output.put_line(s1);
```

```
end;
```

```
/
```

# sql로 값할당

```
declare
    s1 salgrade.hisal%type;
    s2 salgrade.losal%type;
begin
    select min(hisal), max(losal)
    into s1,s2
    from salgrade;
    dbms_output.put_line(s1);
    dbms_output.put_line(s2);
end;
/
```

# 상수 값 할당 오류

```
declare
    s1 constant number :=50;
begin
    --s1 := 100;
    dbms_output.put_line(s1);
end;
/
```

# 디폴트 값 할당

```
declare
    s1 number default 50;
begin
    --s1 := 100;
    dbms_output.put_line(s1);
end;
/
```

# 중첩 선언 및 값 할당

```
declare
    s1 number;
begin
    declare
        s2 number;
    begin
        s1 := 1;
        s2 := 2;
        dbms_output.put_line(s1);
        dbms_output.put_line(s2);
    end;
end;
/
```

# null 값 선언

declare

    v1 varchar2(100);

    v2 varchar2(100) := '';

    v3 varchar2(100) := null;

    v4 varchar2(100) default null;

begin

    null;

end;

/

# type형 이용

declare

```
s1 salgrade.grade%type;  s2 salgrade.losal%type;  
s3 salgrade.hisal%type;
```

begin

```
s1 :=6;          s2 :=5000;  s3 :=8000;  
insert into salgrade (grade, losal, hisal) values (s1,  
s2, s3);
```

```
commit;
```

```
dbms_output.put_line(s1);
```

```
dbms_output.put_line(s2);
```

```
dbms_output.put_line(s3);
```

```
end;
```

```
/
```



```
alter table salgrade  
add constraint CK_losal check( losal < 7000)  
declare  
    s1 salgrade.grade%type;           s2 salgrade.losal%type;  
    s3 salgrade.hisal%type;  
  
begin  
    s1 := &그레이드;          s2 := &로우살;   s3 := &하이살;  
    insert into salgrade      (grade, losal, hisal) values (s1, s2, s3);  
    commit;  
    dbms_output.put_line('성공');  
exception  
    when others then         rollback;  
        dbms_output.put_line('실패');  
  
end;  
/
```

# rowtype형 이용

```
declare
```

```
    s1 salgrade%rowtype;
```

```
begin
```

```
    s1.grade :=7; s1.losal :=7000;    s1.hisal :=8000;
```

```
    insert into salgrade ( grade, losal, hisal)
```

```
    values (s1.grade, s1.losal, s1.hisal);
```

```
    commit;
```

```
    dbms_output.put_line(s1.grade);
```

```
    dbms_output.put_line(s1.losal);
```

```
    dbms_output.put_line(s1.hisal);
```

```
end;
```

```
/
```

# 바인드(bind) 변수

바인드변수 : SQL실행 시점에 값을 할당할 수 있는 변수

```
select sal from emp where empno=:en;
```

```
select sal from emp where empno=:en;
```

```
declare
```

```
    v_sal emp.sal%type;
```

```
begin
```

```
    select sal into v_sal from emp where empno = :empno;
```

```
    dbms_output.put_line(v_sal);
```

```
end;
```

```
/
```

# sysdate 와 같이 사용

```
declare
```

```
    v varchar2(100);
```

```
begin
```

```
    v := 'today ' || sysdate;
```

```
    dbms_output.put_line(v);
```

```
end;
```

```
/
```

# 연습문제

1. 구구단 중 3단을 출력하는 익명 블록을 만들어보자.

<정답>

BEGIN

DBMS\_OUTPUT.PUT\_LINE('3 \* 1 = ' || 3\*1);

DBMS\_OUTPUT.PUT\_LINE('3 \* 2 = ' || 3\*2);

DBMS\_OUTPUT.PUT\_LINE('3 \* 3 = ' || 3\*3);

DBMS\_OUTPUT.PUT\_LINE('3 \* 4 = ' || 3\*4);

DBMS\_OUTPUT.PUT\_LINE('3 \* 5 = ' || 3\*5);

DBMS\_OUTPUT.PUT\_LINE('3 \* 6 = ' || 3\*6);

DBMS\_OUTPUT.PUT\_LINE('3 \* 7 = ' || 3\*7);

DBMS\_OUTPUT.PUT\_LINE('3 \* 8 = ' || 3\*8);

DBMS\_OUTPUT.PUT\_LINE('3 \* 9 = ' || 3\*9);

END;

/

# 연습문제

set serveroutput on

DECLARE

num number:=&num;

BEGIN

DBMS\_OUTPUT.PUT\_LINE(num||' \* 1 = ' || num\*1);

DBMS\_OUTPUT.PUT\_LINE(num||' \* 2 = ' || num\*2);

DBMS\_OUTPUT.PUT\_LINE(num||' \* 3 = ' || num\*3);

DBMS\_OUTPUT.PUT\_LINE(num||' \* 4 = ' || num\*4);

DBMS\_OUTPUT.PUT\_LINE(num||' \* 5 = ' || num\*5);

DBMS\_OUTPUT.PUT\_LINE(num||' \* 6 = ' || num\*6);

DBMS\_OUTPUT.PUT\_LINE(num||' \* 7 = ' || num\*7);

DBMS\_OUTPUT.PUT\_LINE(num||' \* 8 = ' || num\*8);

DBMS\_OUTPUT.PUT\_LINE(num||' \* 9 = ' || num\*9);

END;

/

# 연습문제

2. 사원 테이블에서 201번 사원의 이름과 이메일주소를 출력하는 익명 블록을 만들어 보자.

<정답>

```
DECLARE
```

```
    vs_first_name employees.first_name%TYPE;
```

```
    vs_email      employees.email%TYPE;
```

```
BEGIN
```

```
    SELECT first_name , email
```

```
        INTO vs_first_name , vs_email
```

```
        FROM employees
```

```
        WHERE employee_id = 201;
```

```
    DBMS_OUTPUT.PUT_LINE ( vs_first_name || ' - ' || vs_email);
```

```
END;
```

```
/
```



# 연습문제

3. 사원 테이블에서 사원번호가 제일 큰 사원을 찾아낸 뒤, 이 번호 +1번으로 아래의 사원을 사원테이블에 신규 입력하는 익명 블록을 만들어 보자.

<사원명> : Harrison Ford

<이메일> : HARRIS

<입사일자> : 현재일자

<부서번호> : 50

<업무ID> : AD\_VP

입력후에 입력된 사번 출력하세요

<정답>

DECLARE

vn\_max\_empno employees.employee\_id%TYPE;

vs\_email employees.email%TYPE;

BEGIN

SELECT MAX(employee\_id)

INTO vn\_max\_empno

FROM employees;

INSERT INTO employees ( employee\_id, first\_name,last\_name, email, hire\_date,  
department\_id )

VALUES ( vn\_max\_empno + 1, 'Harrison','Ford', 'HARRIS', SYSDATE, 50);

COMMIT;

END;

/

# if 문

```
declare
```

```
    n number := &num;
```

```
BEGIN
```

```
    if mod(n,2) = 0 then
```

```
        dbms_output.put_line('짝수');
```

```
    else
```

```
        dbms_output.put_line('홀수');
```

```
    end if;
```

```
end;
```

```
/
```

## if문 2

```
declare
```

```
  d varchar2(10) := '&요일';
```

```
begin
```

```
  if d='월' then dbms_output.put_line('월요일이네');
```

```
  elsif d='화' then dbms_output.put_line('화요일이네');
```

```
  elsif d='수' then dbms_output.put_line('수요일이네');
```

```
  elsif d='목' then dbms_output.put_line('목요일이네');
```

```
  elsif d='금' then dbms_output.put_line('금요일이네');
```

```
  elsif d='토' then dbms_output.put_line('토요일이네');
```

```
  else dbms_output.put_line('일요일이네');
```

```
  end if;
```

```
end;
```

```
/
```

# case 문

```
declare
```

```
    d varchar2(10) := '&요일';
```

```
BEGIN
```

```
    case d
```

```
    when '월' then dbms_output.put_line('월요일입니다.');
```

```
    when '화' then dbms_output.put_line('화요일입니다.');
```

```
    when '수' then dbms_output.put_line('수요일입니다.');
```

```
    when '목' then dbms_output.put_line('목요일입니다.');
```

```
    when '금' then dbms_output.put_line('금요일입니다.');
```

```
    when '토' then dbms_output.put_line('토요일입니다.');
```

```
    else dbms_output.put_line('일요일입니다.');
```

```
    end case;
```

```
end;
```

```
/
```

# loop 문

declare

L number:=0;

BEGIN

dbms\_output.put\_line('loop 시작');

loop

if L > 100 then exit;

end if;

dbms\_output.put\_line('loop ' || L || ' time');

L := L + 1;

end loop;

dbms\_output.put\_line('loop 끝');

end;

/

# loop 문

set serveroutput on

DECLARE

num number:=&num;

l number := 1;

BEGIN

dbms\_output.put\_line('구구단 '||num||'단');

loop

if l > 9 then exit;

end if;

DBMS\_OUTPUT.PUT\_LINE(num||' \* '||l||' = ' || num\*l);

l := l+1;

end loop;

END;

/

# EXIT WHEN LOOP문

declare

L number:=0;

BEGIN

dbms\_output.put\_line('loop 시작');

loop

exit when L > 100;

dbms\_output.put\_line('loop ' || L || ' time');

L := L + 1;

end loop;

dbms\_output.put\_line('loop 끝');

end;

/

# While loop문

declare

    L number:=0;

BEGIN

    dbms\_output.put\_line('loop 시작');

    while L < 5 loop

        dbms\_output.put\_line('loop ' || L || ' time');

        L := L + 1;

    end loop;

    dbms\_output.put\_line('loop 끝');

end;

/



# for문

```
declare
```

```
    i number:=1;
```

```
    n number:=&num ;
```

```
BEGIN
```

```
    dbms_output.put_line('for 시작');
```

```
    for idx in i .. n loop
```

```
        dbms_output.put_line('for ' || n || ' time');
```

```
    end loop;
```

```
    dbms_output.put_line('for 끝');
```

```
end;
```

```
/
```

# for문

set serveroutput on

DECLARE

num number:=&num;

BEGIN

dbms\_output.put\_line('구구단 ' || num || '단');

for l in 1..9 loop

DBMS\_OUTPUT.PUT\_LINE(num || ' \* ' || l || ' = ' || num \* l);

end loop;

END;

/

## FOR문 2

```
declare
begin
  for i in 1..3 loop
    for j in 1..3 loop
      dbms_output.put_line('i = '||i||', j = '||j);
    end loop;
  end loop;
end;
/
```

## for문 3

declare

n DBMS\_SQL.VARCHAR2\_TABLE;

BEGIN

n(1) := 'JINO';

n(2) := 'ABC';

n(3) := 'HAHAH';

-- for idx in 1 .. 3 loop

for idx in 1 .. N.COUNT loop

dbms\_output.put\_line( n(idx) );

end loop;

end;

/

## for문 reverse

declare

n DBMS\_SQL.VARCHAR2\_TABLE;

BEGIN

n(1) := 'JINO';

n(2) := 'ABC';

n(3) := 'HAHAH';

for idx in reverse 1 .. n.COUNT loop

dbms\_output.put\_line( n(idx) );

end loop;

end;

/

# 사용자 정의 예러

declare

t number(10) := &온도; hot exception; cold exception;

begin

case

when t < 20 then raise cold;

when t >= 40 then raise hot;

else dbms\_output.put\_line('쭈아 온도');

end case;

exception

when hot then dbms\_output.put\_line('앗 뜨거');

when cold then dbms\_output.put\_line('웃 차가워');

end;

/

# PL-SQL-insert문

declare

```
g salgrade.grade%type;    l salgrade.losal%type;  
h salgrade.hisal%type;
```

Begin

```
g := &grade; l := &losal;    h := &hisal;  
insert into salgrade (grade,losal,hisal)    values(g,l,h);  
dbms_output.put_line(g);  
dbms_output.put_line(l);  
dbms_output.put_line(h);
```

end;

/

# PL-SQL-UPDATE문-

declare

g salgrade.grade%type;

h salgrade.hisal%type;

Begin

g := &grade;

h := &hisal;

update salgrade set hisal= h where grade=g;

dbms\_output.put\_line('grade가 ' || g || '인 hisal을 ' || h || '로 변경 성공');

end;

/



# PL-SQL-UPDATE문-사용자예외

declare

g salgrade.grade%type; h salgrade.hisal%type; e exception;

Begin

g := &grade; h := &hisal;

if h > 10000 then raise e;

end if;

update salgrade set hisal = h where grade = g;

dbms\_output.put\_line('grade가 ' || g || '인 급여 ' || h || '로 변경 성공');

exception

when e then

dbms\_output.put\_line('너 미쳤니?');

end;

/

# PL-SQL-DELETE문

```
declare
```

```
    g salgrade.grade%type;
```

```
Begin
```

```
    g := &grade;
```

```
    delete from salgrade      where grade=g;
```

```
    dbms_output.put_line('grade가 '||g||'인 행 삭제 성공');
```

```
end;
```

```
/
```

# 프로시저란

- 특정 작업을 수행할 수 있고, 이름이 있는 PL/SQL 블록으로서, 매개 변수를 받을 수 있고 반복적으로 사용할 수 있습니다. 보통 연속 실행 또는 구현이 복잡한 트랜잭션을 수행하는 PL/SQL 블록을 데이터 베이스에 저장하기 위해 생성합니다.
  - ⊙ CREATE OR REPLACE 구문을 사용하여 생성합니다.
  - ⊙ IS 로 PL/SQL의 블록을 시작합니다.
  - ⊙ LOCAL 변수는 IS 와 BEGIN 사이에 선언합니다.
- [Syntax]  
CREATE OR REPLACE procedure name  
    IN argument  
    OUT argument  
    IN OUT argument  
IS  
    [변수의 선언]  
BEGIN --> 필수  
    [PL/SQL Block]  
    -- SQL문장, PL/SQL제어 문장  
    [EXCEPTION] --> 선택  
    -- error가 발생할 때 수행하는 문장  
END; --> 필수

# 저장 프로시저

- 오라클은 사용자가 만든 PL/SQL 문을 데이터베이스에 저장 할 수 있도록 저장프로시저라는 것을 제공합니다.
- 이렇게 저장 프로시저를 사용하면 복잡한 DML 문들 필요할 때마다 다시 입력할 필요 없이 간단하게 호출만 해서 복잡한 DML 문의 실행 결과를 얻을 수 있음
- 저장 프로시저를 사용하면 성능도 향상되고, 호환성 문제도 해결됩니다.
- 저장 프로시저를 생성하려면 CREATE PROCEDURE 다음에 생성하고자 하는 프로시저 이름을 기술합니다.
- 이렇게 해서 생성한 저장 프로시저는 여러 번 반복해서 호출해서 사용할 수 있다는 장점이 있습니다.
- 생성된 저장 프로시저를 제거하기 위해서는 DROP PROCEDURE 다음에 제거하고자 하는 프로시저 이름을 기술합니다.
- OR REPLACE 옵션은 이미 같은 이름으로 저장 프로시저를 생성할 경우 기존 프로시저는 삭제하고 지금 새롭게 기술한 내용으로 재 생성하도록 하는 옵션
- 프로시저는 어떤 값을 전달받아서 그 값에 의해서 서로 다른 결과물을 구함
- 값을 프로시저에 전달하기 위해서 프로시저 이름 다음에 괄호로 둘러 쓴 부분에 전달 받을 값을 저장할 변수를 기술합니다.
- 프로시저는 매개 변수의 값에 따라 서로 다른 동작을 수행하게 됩니다.
- [MODE] 는 IN과 OUT, INOUT 세 가지를 기술할 수 있는데 IN 데이터를 전달 받을 때 쓰고 OUT은 수행된 결과를 받아갈 때 사용합니다. INOUT은 두 가지 목적에 모두 사용됩니다.

# Parameter란

- ⊙ 실행 환경과 program사이에 값을 주고 받는 역할을 합니다.
- ⊙ 블록 안에서의 변수와 똑같이 일시적으로 값을 저장하는 역할을 합니다.
- ⊙ Parameter의 타입
  - IN : 실행환경에서 program으로 값을 전달
  - OUT : program에서 실행환경으로 값을 전달
  - INOUT : 실행환경에서 program으로 값을 전달하고,  
다시 program에서 실행환경으로 변경된 값을 전달

# 변수 선언 방법

identifier [CONSTANT] 데이터타입 [NOT NULL] [:= 상수값이나 표현식] ;

- ⊙ Identifier의 이름은 sql의 object명과 동일한 규칙을 따릅니다.
- ⊙ Identifier를 상수로 지정하고 싶은 경우는 CONSTANT라는 KEYWORD를 명시하고 반드시 초기화를 할당합니다.
- ⊙ NOT NULL이 정의되어 있으면 초기값을 반드시 지정하고, 정의되어 있지 않을 때는 생략 가능합니다.
- ⊙ 초기값은 할당 연산자(:=)를 사용하여 정의 합니다.
- ⊙ 초기값을 정의하지 않으면 Identifier는 NULL값을 가지게 됩니다.
- ⊙ 일반적으로 한 줄에 한 개의 Identifier를 정의 합니다.

## ※ 스칼라 데이터 타입은 단수 데이터형으로 한가지의 데이터 값

BINARY_INTEGER	-2147483647에서 2147483647 사이의 정수
NUMBER(P, S)]	고정 및 부동 소숫점 수에 대한 기본 유형
CHAR[(최대길이)]	고정 길이 문자에 대한 기본형은 32767바이트까지 입니다. 지정하지 않는다면 디폴트 길이는 1로 설정됩니다.
LONG	고정 길이 문자에 대한 기본형은 32760바이트까지 입니다. LONG 데이터베이스 열의 최대 폭은 2147483647바이트입니다.
LONG RAW	이진 데이터와 바이트 문자열에 대한 기본형은 32760Byte까지 입니다. LONG RAW 데이터는 PL/SQL에 의해 해석되지 않습니다.
VARCHAR2 (최대길이)	3변수 길이 문자 데이터에 대한 기본형은 32767Byte까지 입니다. VARCHAR2변수와 상수에 대한 디폴트 크기는 없습니다.
DATE	날짜와 시간에 대한 기본형. DATE값은 지정 이후의 초 단위로 날에 대한 시간을 포함합니다. 날짜의 범위는 BC 4712년 1월1일부터 AD 9999년 12월 31일 사이 입니다.
BOOLEAN	논리연산에 사용되는 세 가지 값(TRUE, FALSE, NULL) 중 하나를 저장 하는 데이터 유형

# 실습하기

사원 테이블에 저장된 모든 사원을 삭제하는 프로시저를 작성해보도록 하겠습니다.

1. 모든 사원을 삭제하는 프로시저를 실행시키기 위해서 미리 사원 테이블을 복사해서 새로운 사원 테이블을 만들어 놓읍시다.

```
CREATE OR REPLACE PROCEDURE DEL_emp01
    (v_empno in number)
IS
BEGIN
    DELETE FROM EMP01 where empno=v_empno;
END;
/
```



# 실습하기

2. 생성된 저장 프로시저는 EXECUTE 명령어로 실행시킵니다.

```
EXECUTE DEL_emp01(1112)  
EXEC DEL_emp01(1111)
```

```
BEGIN  
    DEL_EMP01(1113);  
END;  
/
```

# 프로시저 작성 예제

```
SQL> create or replace
PROCEDURE update_sal1
/* IN ? Parameter */
(v_empno IN NUMBER)
IS
BEGIN
    UPDATE emp
    SET sal = sal * 1.1
    WHERE empno = v_empno;
    COMMIT;
END;
/
```

프로시저의 이름은 update\_sal이고 프로시저 update\_sal은 사번(v\_empno)를 입력 받아서 급여를 update시켜주는 sql문입니다.  
프로시저를 끝마칠 때에는 항상 "/"를 지정 합니다.  
프로시저의 실행은 EXECUTE 문을 이용해 프로시저를 실행합니다.

**SQL> execute update\_sal(7369);**

PL/SQL 처리가 정상적으로 완료되었습니다.

7369번 사원의 급여가 10% 인상되었습니다.

SELECT 문을 실행시켜보면 데이터가 수정된 것을 확인할 수 있습니다.

# 프로시저 기초

```
create procedure p1  
is  
begin  
    null;  
end p1;  
/
```

# 프로시저 수정 / 실행법

```
create or replace procedure p1
is
begin
    dbms_output.put_line('hello_world');
end;
/
-----

begin
p1;
end;
/
-----

exec p1;
```

# 인자 값 사용

```
create table t  
(n number);
```

```
-----  
create or replace procedure p2 (p in number)  
is  
begin  
    insert into t values(p);  
    commit;  
end;  
/  
exec p2(p=>200);  
exec p2(300);
```

# 한번에 2개씩 들어가기

```
create or replace procedure p2
    (p1 in number,p2 in number)
is
begin
    insert into t values(p1);
    insert into t values(p2);
    commit;
end p2;
/

exec p2(200,300);
```

## 2열에 2개의 인자 받기

```
create table t2  
    (id in number, name in char);  
-----
```

```
create or replace procedure p3  
    (p1 number, p2 char)  
    is  
begin  
    insert into t2 values(p1,p2);  
end;  
/  
-----
```

```
exec p3(100,'a');
```

# 에러 보기

```
create or replace procedure p4  
is  
begin  
    select * from emp;  
end;  
/
```



# 기본값 사용하기

```
create or replace procedure p4
  (p1 in varchar2 , p2 in varchar2 default 'chris', p3 in varchar2 default
'sean')
is
begin
    dbms_output.put_line(p1);
    dbms_output.put_line(p2);
    dbms_output.put_line(p3);
end;
/
exec p4('tom');
exec p4('tom','bb');
exec p4('tom',p3=>'bb');
exec p4('tom',null,'bb');
```

# out 변수 사용

```
variable name varchar2(10);
```

```
variable sal number;
```

```
-----
```

```
create or replace procedure p5
```

```
    (p1 in number, p2 out emp.ename%type    , p3 out emp.sal%type)
```

```
is
```

```
begin
```

```
    select ename, sal  into p2,p3 from emp where empno=p1;
```

```
exception
```

```
    when NO_DATA_FOUND then          p2 := 'null';
```

```
    p3 := -1;
```

```
end;
```

```
/
```

```
-----
```

```
exec p5('7782',:name,:sal);
```

```
print name sal;
```

```
-----
```

```
exec p5('1234',:name,:sal);
```

```
print name sal;
```

# In out 변수 swap

create or replace procedure p6 (p1 in out number, p2 in out number)  
is

```
    t number;
```

```
begin
```

```
    t := p1;          p1 := p2;          p2 := t;
```

```
end;
```

```
/
```

```
declare
```

```
    n number :=100;          n2 number :=200;
```

```
begin
```

```
    p6(n,n2);
```

```
    dbms_output.put_line('n =' ||n);
```

```
    dbms_output.put_line('n2 =' ||n2);
```

```
end;
```

```
/
```

# insert 프로시저 만들기

```
create table jino
```

```
(id in number(2) not null, name in varchar(10), city in varchar(10));
```

```
-----
```

```
create or replace procedure p7
```

```
( p1 in jino.id%type , p2 in jino.name%type ,  
  p3 in jino.city%type default '서울' )
```

```
is
```

```
begin
```

```
    insert into jino (id,name, city) values (p1,p2,p3);
```

```
end;
```

```
/
```

```
-----
```

```
exec p7(1,'jino');
```

```
exec p7(2,'haha','광주');
```

```
exec p7(3,'aa','LA');
```

# update 프로시저 만들기

```
create or replace procedure p8
    ( p1 in jino.id%type, p2 in jino.city%type)
is
begin
    update jino  set city=p2  where id= p1;
end;
/
-----
exec p8(3,'newyork');
```

# delete 프로시저 만들기

```
create or replace procedure p9  
    ( p1 in jino.id%type )  
as  
begin  
    delete from jino where id= p1;  
end;  
/  
-----  
exec p9(3);
```

# PL/SQL에서 SELECT 문

- 데이터베이스에서 정보를 추출할 필요가 있을 때 또는 데이터베이스로 변경된 내용을 적용할 필요가 있을 때 SQL을 사용합니다.
- PL/SQL은 SQL에 있는 DML 명령을 지원합니다. 테이블의 행에서 정의된 값을 변수에 할당시키기 위해 SELECT문장을 사용합니다.
- PL/SQL의 SELECT 문은 INTO절이 필요한데, INTO절에는 데이터를 저장할 변수를 기술합니다.
- SELECT 절에 있는 컬럼은 INTO절에 있는 변수와 1대1대응을 하기 위해 개수와 데이터의 형, 길이가 일치하여야 합니다.
- SELECT 문은 INTO절에 의해 하나의 행만을 저장할 수 있습니다.

# PL/SQL에서 SELECT 문

```
SELECT select_list
INTO {variable_name1[,variable_name2,...] / record_name}
FROM table_name
WHERE condition;
```

구문	설명
<i>select_list</i>	열의 목록이며 행 함수, 그룹 함수, 표현식을 기술할 수 있습니다.
<i>variable_name</i>	읽어들인 값을 저장하기 위한 스칼라 변수
<i>record_name</i>	읽어 들인 값을 저장하기 위한 PL/SQL RECORD 변수
Condition	PL/SQL 변수와 상수를 포함하여 열명,표현식,상수,비교 연산자로 구성되며 오직 하나의 값을 RETURN할 수 있는 조건이어야 합니다.



# PL/SQL에서 SELECT 문

```
SELECT EMPNO, ENAME INTO VEMPNO, VENAME  
FROM EMP  
WHERE ENAME='SCOTT' ;
```

- VEMPNO, VENAME 변수는 컬럼(EMPNO, ENAME)과 동일한 데이터형을 갖도록 하기 위해서 %TYPE 속성을 사용합니다.
- INTO 절의 변수는 SELECT에서 기술한 컬럼의 데이터형뿐만 아니라 컬럼의 수와도 일치해야 합니다.

# select 프로시저 만들기

```
create or replace procedure
    p_ename(p1 in emp.empno%type)
is
    e1 emp.ename%type;
begin
    select ename into e1 from emp where
empno=p1;
    dbms_output.put_line(e1);
end;
/
-----
exec p_ename(7782);
```

# 연습문제

1. EMP TABLE에 사번(empno), 이름(ename), 급여(sal), 부서번호(deptno)를 전달받아 등록하는 PROCEDURE를 작성하여라.
2. 사원 번호(empno)와 급여(sal)를 입력 받아 수정하는 PROCEDURE를 작성하여라.

# 프로시저 생성

```
CREATE OR REPLACE PROCEDURE my_new_job_proc
( p_job_id    IN JOBS.JOB_ID%TYPE,
  p_job_title IN JOBS.JOB_TITLE%TYPE,
  p_min_sal   IN JOBS.MIN_SALARY%TYPE,
  p_max_sal   IN JOBS.MAX_SALARY%TYPE )
IS
BEGIN
    INSERT INTO JOBS
    ( job_id, job_title, min_salary, max_salary, create_date,
      update_date)      VALUES ( p_job_id, p_job_title, p_min_sal, p_max_sal,
      SYSDATE, SYSDATE);
    COMMIT;
END ;
/
-- 프로시저 실행
EXEC my_new_job_proc ('SM_JOB1', 'Sample JOB1', 1000, 5000);
SELECT * FROM jobs WHERE job_id = 'SM_JOB1';
```

# 프로시저 생성

```
CREATE OR REPLACE PROCEDURE my_new_job_proc
    ( p_job_id    IN JOBS.JOB_ID%TYPE,      p_job_title IN JOBS.JOB_TITLE%TYPE,
      p_min_sal   IN JOBS.MIN_SALARY%TYPE,   p_max_sal   IN
JOBS.MAX_SALARY%TYPE )
IS
    vn_cnt NUMBER := 0;
BEGIN    -- 동일한 job_id가 있는지 체크
    SELECT COUNT(*) INTO vn_cnt FROM JOBS WHERE job_id = p_job_id;
    IF vn_cnt = 0 THEN
        INSERT INTO JOBS ( job_id, job_title, min_salary, max_salary, create_date,
update_date) VALUES ( p_job_id, p_job_title, p_min_sal, p_max_sal, SYSDATE, SYSDATE);
    ELSE -- 있으면 UPDATE
        UPDATE JOBS SET job_title = p_job_title, min_salary = p_min_sal,
max_salary = p_max_sal, update_date = SYSDATE
        WHERE job_id = p_job_id;
    END IF;
END ;
/
EXEC my_new_job_proc ('SM_JOB1', 'Sample JOB1', 2000, 6000);
SELECT * FROM jobs WHERE job_id = 'SM_JOB1';
```

# ◆ 함수(Function)

- 보통 값을 계산하고 결과값을 반환하기 위해서 함수를 많이 사용
- 대부분 구성이 프로시저와 유사
- 반드시 반환될 값의 데이터 타입을 RETURN문에 선언해야 합니다.
- 또한 PL/SQL블록 내에서 RETURN문을 통해서 반드시 값을 반환해야 합니다.

## [Syntax]

CREATE OR REPLACE FUNCTION function name

[(argument...)]

RETURN datatype

-- Datatype은 반환되는 값의 datatype입니다.

IS

[변수 선언 부분]

BEGIN

[PL/SQL Block]

-- PL/SQL 블록에는 적어도 한 개의 RETURN 문이 있어야 합니다.

-- PL/SQL Block은 함수가 수행할 내용을 정의한 몸체부분입니다.

END;

# Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
      [(argument1 [mode1] datatype [{:= | DEFAULT} expression]
      [,argument2 [mode2] datatype [{:= | DEFAULT} expression], . . .)]
RETURN data_type
{IS | AS}
BEGIN
      pl/sql_block;
END;
```

**OR REPLACE**      function\_name이 존재할 경우 **FUNCTION**의 내용을 지우고 다시 생성

function\_name    Function의 이름은 표준 Oracle 명명법에 따른 함수이름

argument        매개변수의 이름

mode             3가지가 있다

IN : 입력 매개변수로 상용

OUT : 출력 매개변수로 사용

IN OUT : 입력, 출력 매개변수로 상용

data\_type        반환되는 값의 datatype

pl/sql\_block    **FUNCTION**를 구성하는 코드를 구성하는 PL/SQL의 블록

```

SQL>CREATE OR REPLACE FUNCTION FC_update_sal (v_empno      IN      NUMBER)
-- 리턴되는 변수의 데이터타입을 꼭 정의해야 합니다
RETURN NUMBER
IS
  v_sal emp.sal%type;
BEGIN
  UPDATE emp      SET sal  = sal  * 1.1      WHERE empno  = v_empno;
  COMMIT;
  SELECT sal INTO v_sal      FROM emp      WHERE empno = v_empno;
-- 리턴문이 꼭 존재해야 합니다
RETURN v_sal;
END;
/

```

- v\_sal이라는 %type 변수가 사용되고 있습니다. 테이블 데이터 타입을 참고하세요. 세미콜론(;)으로 블록을 종료한 뒤 "/"를 붙여 코드를 끝마칩니다.

- 함수의 실행 .

SQL> VAR salary NUMBER; 먼저 함수의 반환값을 저장할 변수를 선언합니다  
EXECUTE 문을 이용해 함수를 실행합니다.

SQL>EXECUTE :salary := FC\_update\_sal(7900);

오라클 SQL에서 선언된 변수의 출력은 PRINT문을 사용합니다.

PRINT문으로 함수의 반환값을 저장한 salary의 값을 확인하면 됩니다.

SQL>PRINT salary;

SALARY

-----

1045



# EMP 테이블에서 이름으로 부서 번호를 검색하는 함수를 작성

```
CREATE OR REPLACE FUNCTION ename_deptno(
    v_ename IN      emp.ename%TYPE)
RETURN NUMBER
IS
    v_deptno emp.deptno%TYPE;
BEGIN
    SELECT deptno
        INTO v_deptno
        FROM emp
        WHERE ename = UPPER(v_ename);
    DBMS_OUTPUT.PUT_LINE('부서번호 : ' || TO_CHAR(v_deptno));
    RETURN v_deptno;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('입력한 MANAGER는 없습니다.');
```

WHEN TOO\_MANY\_ROWS THEN  
 DBMS\_OUTPUT.PUT\_LINE('자료가 2건 이상입니다.');

WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('기타 에러입니다.');

```
END;
/
```

## **FUNCTION 실행**

- PL/SQL을 지원하는 어떤 툴이나 언어에서도 함수를 실행할 수 있고 PL/SQL내부에서 식의 일부로서 함수를 실행할 수 있다. SQL\*Plus에서 FUNCTION 호출은 Stored Function를 참조하는 PL/SQL 문을 실행하기 위해 EXECUTE 명령을 사용할 수 있다. EXECUTE는 명령 다음에 입력되는 Stored Function를 실행한다.

## **Syntax**

output\_variable := function\_name[(argument1[,argument2, . . . . .])]

## **SQL\*Plus에서 함수 실행**

```
SQL> EXECUTE :g_deptno := ename_deptno('ALLEN')
```

```
SQL> SET SERVEROUTPUT ON
```

```
SQL> VAR g_deptno NUMBER
```

```
SQL> EXECUTE :g_deptno := ename_deptno('SCOTT')
```

```
부서번호 : 10
```

```
SQL> PRINT g_deptno
```

```
G_DEPTNO
```

```
-----
```

```
10
```

# EMP 테이블에서 이름을 입력 받아 부서번호,부서명,급여를 검색

```
CREATE OR REPLACE FUNCTION emp_disp(  
    v_ename    IN    emp.ename%TYPE,    v_dname    OUT    dept.dname%TYPE,  
    v_sal OUT    emp.sal%TYPE)  
RETURN NUMBER  
IS  
    v_deptno    emp.deptno%TYPE;    v_dname_temp    dept.dname%TYPE;  
    v_sal_temp    emp.sal%TYPE;  
BEGIN  
    SELECT sal,deptno    INTO v_sal_temp,v_deptno  
        FROM emp    WHERE ename = UPPER(v_ename);  
    SELECT dname    INTO v_dname_temp  
        FROM dept    WHERE deptno = v_deptno;  
    v_dname := v_dname_temp;    v_sal := v_sal_temp;  
    DBMS_OUTPUT.PUT_LINE('성    명 : ' || v_ename);  
    DBMS_OUTPUT.PUT_LINE('부서번호 : ' || TO_CHAR(v_deptno));  
    DBMS_OUTPUT.PUT_LINE('부    서    명 : ' || v_dname_temp);  
    DBMS_OUTPUT.PUT_LINE('급    여 : ' || TO_CHAR(v_sal_temp,'$999,999'));  
    RETURN v_deptno;  
END;  
/
```

# 실행

```
SQL> SET SERVEROUTPUT ON
SQL> VAR g_deptno NUMBER
SQL> VAR g_dname VARCHAR2(20)
SQL> VAR g_sal NUMBER
SQL> EXECUTE :g_deptno := emp_disp('scott',:g_dname,:g_sal)
```

성명 : scott

부서번호 : 10

부서명 : ACCOUNTING

급여 : \$3,000

PL/SQL procedure successfully completed.

```
SQL> PRINT g_deptno
```

G\_DEPTNO

-----

10

```
SQL> PRINT g_dname
```

G\_DNAME

-----

ACCOUNTING

```
SQL> PRINT g_sal
```

G\_SAL

-----

3000

# 사용자 정의 함수

```
create or replace function f1    return varchar2
is
begin          return 'hello world';
end;
/
declare v varchar(100);
begin
          v := f1;          dbms_output.put_line(v);
end;
/
create or replace procedure p10    (p1 varchar2)
is
begin  dbms_output.put_line(p1);
end;
/
exec p10(f1);
```

# 사용자 정의 함수

```
create or replace function f2      (d date)
    return varchar
is
begin
    return to_char(d,'day');
end;
/
declare
    v varchar2(10);
begin
    v := f2('2005-12-27');  dbms_output.put_line(v);
end;
/
exec p10( f2('2005-12-27'));
select  hiredate, f2(hiredate) from emp;
```

# 사용자 정의 함수

```
create or replace function f3    (n number )  
    return varchar  
is  
begin  
    if mod( n, 2 ) = 0 then return '짝수' ;  
    else      return '홀수' ;  
end if;  
end;  
/
```

- 선언 예제  
v\_price CONSTANT NUMBER(4,2) := 12.34 ;  
-- 상수 숫자 선언(변할 수 없다)  
v\_name VARCHAR2(20) ;  
v\_Bir\_Type CHAR(1) ;  
v\_flag BOOLEAN NOT NULL := TRUE ;  
-- NOT NULL 값 TRUE로 초기화  
v\_birthday DATE;

## %TYPE 데이터형

- ⊙ %TYPE 데이터형은 기술한 데이터베이스 테이블의 컬럼 데이터 타입을 모를 경우 사용할 수 있고,
- ⊙ 또, 코딩이후 데이터베이스 컬럼의 데이터 타입이 변경될 경우 다시 수정할 필요가 없습니다.
- ⊙ 이미 선언된 다른 변수나 데이터베이스 컬럼의 데이터 타입을 이용하여 선언합니다.
- ⊙ 데이터 베이스 테이블과 컬럼 그리고 이미 선언한 변수명이 %TYPE앞에 올 수 있습니다.

## %TYPE 속성을 이용하여 얻을 수 있는 장점

- 기술한 DB column definition을 정확히 알지 못하는 경우에 사용할 수 있습니다.
- 기술한 DB column definition이 변경 되어도 다시 PL/SQL을 고칠 필요가 없습니다.

## 예제

```
v_empno emp.empno%TYPE := 7900 ;  
v_ename emp.ename%TYPE;
```



# 예제 프로시저

```
SQL>CREATE OR REPLACE PROCEDURE Emp_Info    ( p_empno IN emp.empno%TYPE )
```

```
IS
```

```
-- %TYPE 데이터형 변수 선언
```

```
v_empno emp.empno%TYPE;
```

```
v_ename emp.ename%TYPE;
```

```
v_sal   emp.sal%TYPE;
```

```
BEGIN
```

```
DBMS_OUTPUT.ENABLE;
```

```
-- %TYPE 데이터형 변수 사용
```

```
SELECT empno, ename, sal
```

```
INTO v_empno, v_ename, v_sal
```

```
FROM emp
```

```
WHERE empno = p_empno ;
```

```
-- 결과값 출력
```

```
DBMS_OUTPUT.PUT_LINE( '사원번호 : ' || v_empno );
```

```
DBMS_OUTPUT.PUT_LINE( '사원이름 : ' || v_ename );
```

```
DBMS_OUTPUT.PUT_LINE( '사원급여 : ' || v_sal );
```

```
END;
```

```
/
```

```
SQL>SET SERVEROUTPUT ON;    -- DBMS_OUTPUT 결과값을 화면에 출력 하기 위해
```

```
SQL> EXECUTE Emp_Info(7369);
```

```
사원번호 : 7369
```

```
사원이름 : SMITH
```

```
사원급여 : 880
```

```
CREATE OR REPLACE PROCEDURE emp_sal_update(
    p_empno IN emp.empno%TYPE, p_sal IN emp.sal%TYPE)
IS
BEGIN
    UPDATE emp
        SET sal = p_sal
        WHERE empno = p_empno;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(p_empno) ||
            '는 없는 사원번호입니다.');
```

# 실행

```
SQL> EXECUTE emp_sal_update(7788,3500)
```

PL/SQL procedure successfully completed.

```
SQL> SELECT empno,ename,job,sal  
2 FROM emp  
3 WHERE empno = 7788;
```

EMPNO	ENAME	JOB	SAL
7788	SCOTT	ANALYST	3500

# 복합데이터타입

- 하나 이상의 데이터값을 갖는 데이터 타입으로 배열과 비슷한 역할을 하고 재사용이 가능합니다.  
%ROWTYPE 데이터 형과, PL/SQL 테이블과 레코드가 복합 데이터 타입에 속합니다.

## %ROWTYPE

- ⊙ 테이블이나 뷰 내부의 컬럼 데이터형, 크기, 속성등을 그대로 사용할 수 있습니다.
- ⊙ %ROWTYPE 앞에 오는 것은 데이터 베이스 테이블 이름입니다.
- ⊙ 지정된 테이블의 구조와 동일한 구조를 갖는 변수를 선언할 수 있습니다.
- ⊙ 데이터베이스 컬럼들의 수나 DATATYPE을 알지 못할 때 편리 합니다.
- ⊙ 테이블의 데이터 컬럼의 DATATYPE이 변경될 경우 프로그램을 재수정할 필요가 없습니다.

# %ROWTYPE 예제 프로시저

```
SQL> CREATE OR REPLACE PROCEDURE RowType_Test      ( p_empno IN  
      emp.empno%TYPE )
```

```
IS
```

```
      v_emp  emp%ROWTYPE ;
```

```
BEGIN
```

```
-- %ROWTYPE 변수 사용
```

```
      SELECT empno, ename, hiredate
```

```
      INTO v_emp.empno, v_emp.ename, v_emp.hiredate
```

```
      FROM emp      WHERE empno = p_empno;
```

```
      DBMS_OUTPUT.PUT_LINE( '사원번호 : ' || v_emp.empno );
```

```
      DBMS_OUTPUT.PUT_LINE( '사원이름 : ' || v_emp.ename );
```

```
      DBMS_OUTPUT.PUT_LINE( '입 사 일 : ' || v_emp.hiredate );
```

```
END;
```

```
/
```

```
SQL> SET SERVEROUTPUT ON ; -- DBMS_OUTPUT.PUT_LINE을 출력하기 위해 사용
```

```
SQL> EXECUTE RowType_Test(7900);
```

```
사원번호 : 7900
```

```
사원이름 : JAMES
```

```
입 사 일 : 81/12/03
```

# PL/SQL 테이블

- PL/SQL 에서의 테이블은 오라클 SQL에서의 테이블과는 다릅니다.  
PL/SQL에서의 테이블은 일종의 일차원 배열이다
- ⊙ 테이블은 크기에 제한이 없으면 그 ROW의 수는 데이터가 들어옴에 따라 자동 증가 합니다.
- ⊙ BINARY\_INTEGER 타입의 인덱스 번호로 순서가 정해집니다.
- ⊙ 하나의 테이블에 한 개의 컬럼 데이터를 저장 합니다.
- TYPE prdname\_table IS TABLE OF VARCHAR2(30)  
INDEX BY BINARY\_INTEGER;  
-- prdname\_table 테이블타입으로 prdname\_tab변수를 선언해서 사용  
prdname\_tab prdname\_table ;

```
TYPE table_name IS TABLE OF datatype  
INDEX BY BINARY INTEGER
```

```
Identifier type_name ;
```

# PL/SQL 테이블 예제 프로시저

```
CREATE OR REPLACE PROCEDURE Table_Test
(v_deptno IN emp.deptno%TYPE)
IS
    TYPE empno_table IS TABLE OF
        emp.empno%TYPE
        INDEX BY BINARY_INTEGER;
    TYPE ename_table IS TABLE OF
        emp.ename%TYPE
        INDEX BY BINARY_INTEGER;
    TYPE sal_table IS TABLE OF
        emp.sal%TYPE
        INDEX BY BINARY_INTEGER;
    -- 테이블타입으로 변수를 선언 사용
    empno_tab empno_table ;
    ename_tab ename_table ;
    sal_tab sal_table;
    i BINARY_INTEGER := 0;
BEGIN
    FOR emp_list IN(SELECT empno, ename,
        sal FROM emp WHERE deptno =
        v_deptno) LOOP
```

```
        /* emp_list는 자동선언되는 BINARY_INTEGER
        형 변수로 1씩 증가합니다.
        emp_list대신 다른 문자열 사용가능 */
        i := i + 1;
        -- 테이블 변수에 검색된 결과를 넣
        습니다
        empno_tab(i) := emp_list.empno ;
        ename_tab(i) := emp_list.ename ;
        sal_tab(i) := emp_list.sal ;
    END LOOP;
    -- 1부터 i까지 FOR 문을 실행
    FOR cnt IN 1..i LOOP
        -- TABLE변수에 넣은 값을 뿌려줌
        DBMS_OUTPUT.PUT_LINE( '사원번호 :
        ' || empno_tab(cnt) );
        DBMS_OUTPUT.PUT_LINE( '사원이름 :
        ' || ename_tab(cnt) );
        DBMS_OUTPUT.PUT_LINE( '사원급여 :
        ' || sal_tab(cnt) );
    END LOOP;
END;
/
```

# 실행 결과

SQL> SET SERVEROUTPUT ON ; -- (DBMS\_OUTPUT.PUT\_LINE을 출력  
하기 위해 사용)

SQL> EXECUTE Table\_Test(10);

사원번호 : 7782

사원이름 : CLARK

사원급여 : 2450

사원번호 : 7839

사원이름 : KING

사원급여 : 5000

사원번호 : 7934

사원이름 : MILLER

사원급여 : 1300

PL/SQL 처리가 정상적으로 완료되었습니다.

emp 테이블에 있는 데이터의 입력한 부서에 해당하는 사원번호,  
사원이름, 사원급여를  
뿌려주는 프로시저 입니다



# PL/SQL 레코드

여러 개의 데이터 타입을 갖는 변수들의 집합입니다.

- 스칼라, RECORD, 또는 PL/SQL TABLE datatype중 하나 이상의 요소로 구성됩니다.
- 논리적 단위로서 필드 집합을 처리할 수 있도록 해 줍니다.
- PL/SQL 테이블과 다르게 개별 필드의 이름을 부여할 수 있고, 선언시 초기화가 가능합니다.

```
TYPE record_name IS RECORD  
(필드이름1   필드유형1 [NOT NULL {:= | DEFAULT} 식 ],  
  필드이름2   필드유형2 [NOT NULL {:= | DEFAULT} 식 ],  
  필드이름3   필드유형3 [NOT NULL {:= | DEFAULT} 식 ]);  
  
Identifier record_name ;
```

```
TYPE record_test IS RECORD  
( record_empno   NUMBER,  
  record_ename   VARCHAR2(30),  
  record_sal     NUMBER);  
prd_record      record_test;
```

# PL/SQL RECORD 예제 프로시저

```
CREATE OR REPLACE PROCEDURE Record_Test ( p_empno IN emp.empno%TYPE )
IS
    TYPE emp_record IS RECORD
    (v_empno    NUMBER,      v_ename    VARCHAR2(30),    v_hiredate    DATE );
    emp_rec    emp_record ;
BEGIN
    SELECT empno, ename, hiredate
        INTO emp_rec.v_empno, emp_rec.v_ename, emp_rec.v_hiredate
        FROM emp    WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE( '사원번호 : ' || emp_rec.v_empno );
    DBMS_OUTPUT.PUT_LINE( '사원이름 : ' || emp_rec.v_ename );
    DBMS_OUTPUT.PUT_LINE( '입 사 일 : ' || emp_rec.v_hiredate );
END;
```

/

실행 결과

```
SQL> SET SERVEROUTPUT ON ; -- (DBMS_OUTPUT.PUT_LINE을 출력하기 위해 사용)
```

```
SQL> EXECUTE Record_Test(7369);
```

```
사원번호 : 7369
```

```
사원이름 : SMITH
```

```
입 사 일 : 80/12/17
```

# PL/SQL Table of Record

- ○ PL/SQL TABLE 변수 선언과 비슷하며 데이터타입을 %ROWTYPE 으로 선언하면 됩니다.  
○ PL/SQL TABLE과 RECORD의 복합 기능을 합니다.

DECLARE

TYPE dept\_table\_type IS TABLE OF dept%ROWTYPE

INDEX BY BINARY\_INTEGER;

-- Each element of dept\_table is a record

dept\_table dept\_table\_type ;

# PL/SQL TABLE OF RECORD 예제 프로시저

```
CREATE OR REPLACE PROCEDURE Table_Test
IS
    i BINARY_INTEGER := 0;
    -- PL/SQL Table of Record의 선언
    TYPE dept_table_type IS TABLE OF dept%ROWTYPE
    INDEX BY BINARY_INTEGER;
    dept_table dept_table_type;
BEGIN
    FOR dept_list IN (SELECT * FROM dept) LOOP
        i:= i+1;
        dept_table(i).deptno := dept_list.deptno ;
        dept_table(i).dname := dept_list.dname ;
        dept_table(i).loc := dept_list.loc ;
    END LOOP;
    FOR cnt IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE( '부서번호 : ' || dept_table(cnt).deptno ||
                               '부서명 : ' || dept_table(cnt).dname ||
                               '위치 : ' || dept_table(cnt).loc );
    END LOOP;
END;
/
```

# 실행결과

```
SQL>set serveroutput on;
```

```
SQL>exec Table_test;
```

부서번호 : 10부서명 : ACCOUNTING위치 : NEW\_YORK

부서번호 : 20부서명 : RESEARCH위치 : DALLAS

부서번호 : 30부서명 : 인사과위치 : CHICAGO

부서번호 : 40부서명 : OPERATIONS위치 : BOS%TON

PL/SQL 처리가 정상적으로 완료되었습니다.

# Insert문

```
CREATE OR REPLACE PROCEDURE Insert_Test
( v_empno  IN emp.empno%TYPE,
  v_ename  IN emp.ename%TYPE,
  v_deptno IN emp.deptno%TYPE )
IS
BEGIN
    INSERT INTO emp(empno, ename, hiredate, deptno)
    VALUES(v_empno, v_ename, sysdate, v_deptno);
    DBMS_OUTPUT.PUT_LINE( '사원번호 : ' || v_empno );
    DBMS_OUTPUT.PUT_LINE( '사원이름 : ' || v_ename );
    DBMS_OUTPUT.PUT_LINE( '사원부서 : ' || v_deptno );
    DBMS_OUTPUT.PUT_LINE( '데이터 입력 성공 ' );
END ;
/
```

SQL> SET SERVEROUTPUT ON ; -- (DBMS\_OUTPUT.PUT\_LINE을 출력하기 위해 사용)

SQL> EXECUTE Insert\_Test(1000, 'brave', 20);

사원번호 : 1000  
사원이름 : brave  
사원부서 : 20

# UPDATE

```
CREATE OR REPLACE PROCEDURE Update_Test
( v_empno IN emp.empno%TYPE,      -- 급여를 수정한 사원의 사번
  v_rate   IN   NUMBER )          -- 급여의 인상/인하율
IS
v_emp emp%ROWTYPE ;
BEGIN
UPDATE emp   SET sal = sal+(sal * (v_rate/100))  -- 급여를 계산
WHERE empno = v_empno ;
DBMS_OUTPUT.PUT_LINE( '데이터 수정 성공 ' );
-- 수정된 데이터 확인하기 위해 검색
SELECT empno, ename, sal   INTO v_emp.empno, v_emp.ename, v_emp.sal
FROM emp   WHERE empno = v_empno ;
DBMS_OUTPUT.PUT_LINE( ' **** 수 정 확 인 **** ');
DBMS_OUTPUT.PUT_LINE( '사원번호 : ' || v_emp.empno );
DBMS_OUTPUT.PUT_LINE( '사원이름 : ' || v_emp.ename );
DBMS_OUTPUT.PUT_LINE( '사원급여 : ' || v_emp.sal );
END ;
/
```

```
SQL> SET SERVEROUTPUT ON ; -- (DBMS_OUTPUT.PUT_LINE을 출력하기 위해 사용)
SQL> EXECUTE Update_Test(7900, -10);
데이터 수정 성공 **** 수 정 확 인 ****
사원번호 : 7900   사원이름 : JAMES
사원급여 : 855
```



# DELETE

```
CREATE OR REPLACE PROCEDURE Delete_Test ( p_empno IN emp.empno%TYPE )
IS
-- 삭제 데이터를 확인하기 레코드 선언
TYPE del_record IS RECORD
( v_empno      emp.empno%TYPE,
  v_ename      emp.ename%TYPE,
  v_hiredate    emp.hiredate%TYPE) ;
  v_emp del_record ;
BEGIN
-- 삭제된 데이터 확인용 쿼리
SELECT empno, ename, hiredate
INTO v_emp.v_empno, v_emp.v_ename, v_emp.v_hiredate
FROM emp WHERE empno = p_empno ;
DBMS_OUTPUT.PUT_LINE( '사원번호 : ' || v_emp.v_empno );
DBMS_OUTPUT.PUT_LINE( '사원이름 : ' || v_emp.v_ename );
DBMS_OUTPUT.PUT_LINE( '입 사 일 : ' || v_emp.v_hiredate );
-- 삭제 쿼리
DELETE FROM emp WHERE empno = p_empno ;
DBMS_OUTPUT.PUT_LINE( '데이터 삭제 성공 ' );
END;

/
SQL> SET SERVEROUTPUT ON ; -- (DBMS_OUTPUT.PUT_LINE을 출력하기 위해 사용)
SQL> EXECUTE Delete_Test(7900);
```

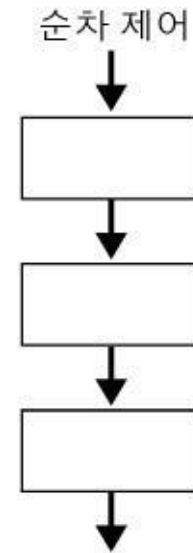
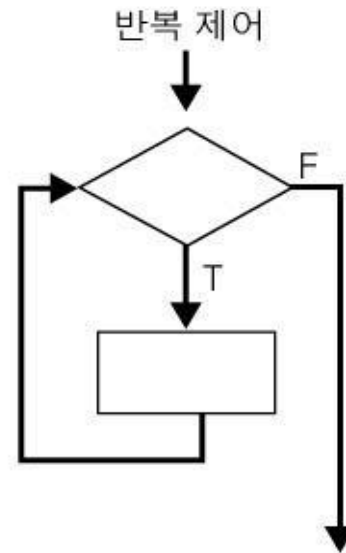
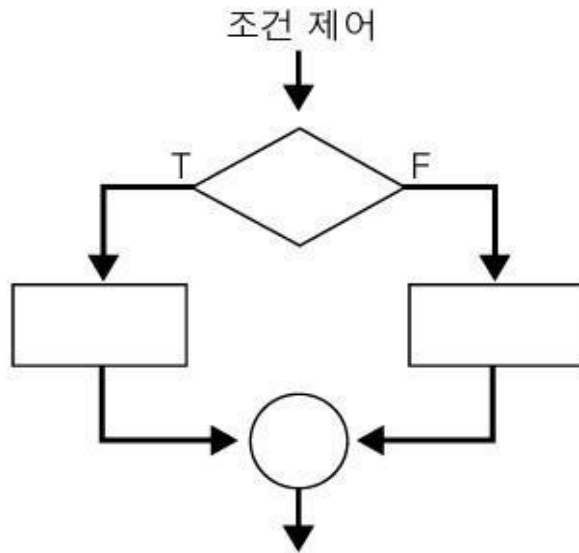


# PL/SQL 제어문

← 문장들의 논리적 흐름을 변경

← 종류

- 조건 제어(Conditional Control)
- 반복 제어(Iteractive Control)
- 순차 제어(Sequential Control)



# 반복 제어 (Iterative Control)

## ← 특징

- ☑ 한 문장 또는 일련의 문장들을 반복 실행 할 수 있는 루프(Loop)를 구성하기 위한 유형
- ☐ 기본(BASIC) 루프, FOR 루프, WHILE 루프

## ← 기본(BASIC) 루프

- ☑ LOOP와 END LOOP 사이에 반복되는 문장 부분들로 구성
- ☑ 실행상의 흐름이 END LOOP에 도달할 때마다 그와 짝을 이루는 LOOP 문으로 제어가 되돌아간다. => 무한 루프
- ☑ 루프에서 빠져나가려면 EXIT 문 사용.

LOOP

실행할 문장들;

EXIT [ WHEN 조건절 ] ;

END LOOP;

# 반복 제어 (Iterative Control)

## □ EXIT 문

- ✂ **END LOOP** 문 다음 문으로 제어를 보내기 때문에 루프 종료
- ✂ **IF** 문 내의 처리 작업으로서, 또는 루프 내의 독립적인 문장으로서도 사용 가능
- ✂ 조건에 따라 루프를 종료할 수 있도록 **WHEN** 절 추가 가능

### LOOP

...

**IF credit\_rating < 3 THEN**

...

**EXIT; -- 이 절을 만나면 바로 루프를 빠져 나가게 됨 END IF**

;

**END LOOP;**

### LOOP

**FETCH c1 INTO ...**

**EXIT WHEN c1%NOTFOUND; -- 조건이 참이면 루프를 빠져나감**

...

**END LOOP; CLOSE c1;**

# 반복 제어 (Iterative Control)

## ← FOR 루프

- ☑ PL/SQL이 수행할 반복 횟수를 정하기 위한 제어문 가짐

```
FOR 인덱스 IN [ REVERSE ] 하한..상한 LOOP  
    문장1; 문장2;  
    ...  
END LOOP;
```

- ✎ 인덱스는 상한에 도달할 때까지 루프를 반복할 때마다 자동으로 1씩 증감하는 값으로 정수
- ✎ 인덱스는 정수로 자동 선언되므로 따로 선언할 필요가 없다.
- ✎ REVERSE는 상한에서 하한까지 인덱스가 반복 때마다 감소하게 하고, 하한은 인덱스 값의 범위에 대한 하한, 상한은 인덱스 값의 범위에 대한 상한 값이다.
- ✎ IN 다음에는 커서(cursor)나 select 문이 올 수 있다.

# FOR LOOP문

```
FOR index IN [REVERSE] 시작값 .. END값 LOOP  
    statement 1  
    statement 2  
    ....  
END LOOP;
```

- index 는 자동 선언되는 binary integer 형 변수이고 1씩 증가합니다.
- reverse 옵션이 사용될 경우 index 는 upper\_bound에서 lower\_bound로 1씩 감소합니다.
- IN 다음에는 cursor나 select 문이 올 수 있습니다.

# FOR문 예제

DECLARE

-- 사원 이름을 출력하기 위한 PL/SQL 테이블 선언  
TYPE ename\_table IS TABLE OF emp.ename%TYPE  
INDEX BY BINARY\_INTEGER;

-- 사원 급여를 출력하기 위한 PL/SQL 테이블 선언  
TYPE sal\_table IS TABLE OF emp.sal%TYPE  
INDEX BY BINARY\_INTEGER;

ename\_tab ename\_table;

sal\_tab sal\_table;

i BINARY\_INTEGER := 0;

BEGIN

FOR emp\_list IN (SELECT ename, sal FROM emp WHERE deptno = 10) LOOP

i := i + 1 ;

ename\_tab(i) := emp\_list.ename; -- 테이블에 사원 이름을 저장

sal\_tab(i) := emp\_list.sal; -- 테이블에 사원 급여를 저장

END LOOP;

FOR cnt IN 1..i LOOP -- 화면에 출력

DBMS\_OUTPUT.PUT\_LINE('사원이름 : ' || ename\_tab(cnt));

DBMS\_OUTPUT.PUT\_LINE('사원급여 : ' || sal\_tab(cnt));

END LOOP;

END;

/

# Loop문 while문

LOOP

PL/SQL statement...

다른 LOOP를 포함하여 중첩으로 사용 가능

EXIT [WHEN condition]

END LOOP;

EXIT 문이 사용되었을 경우, 무조건 LOOP문을 빠져나갑니다,

EXIT WHEN 이 사용될 경우 WHEN 절에 LOOP를 빠져 나가는 조건을 제어할 수 있습니다.

# LOOP 문 예제

```
SET SERVEROUTPUT ON ; -- (DBMS_OUTPUT.PUT_LINE을 출력용)
```

```
DECLARE
```

```
    v_cnt number(3) := 100;
```

```
BEGIN
```

```
    LOOP
```

```
        INSERT INTO emp(empno, ename , hiredate)
```

```
        VALUES(v_cnt, 'test'||to_char(v_cnt), sysdate);
```

```
        v_cnt := v_cnt+1;
```

```
        EXIT WHEN v_cnt > 110;
```

```
    END LOOP;
```

```
    DBMS_OUTPUT.PUT_LINE(v_cnt-100 || '개의 데이터 입력');
```

```
END;
```

```
/
```



# WHILE LOOP 문

DECLARE

cnt BINARY\_INTEGER := 1;

i BINARY\_INTEGER := 10;

BEGIN

WHILE cnt < 10 LOOP

INSERT INTO emp(empno, ename , hiredate)

VALUES(cnt, 'test', sysdate);

cnt := cnt + 1 ;

END LOOP ;

END;

/

# 조건 제어 (Conditional Control)

## ← 조건에 따라 선택적으로 작업을 수행하도록 하는 구문

- ☑ 조건에 따라 참일 경우와 거짓일 경우 각각 다른 문장을 수행하는 구조
- ☐ IF 문과 CASE 문

## ← IF 문

- ☑ 조건이 TRUE이면 THEN 이하의 문장을 실행하고, 조건이 FALSE나 NULL이면 ELSE 이하의 문장 실행
- ☑ 복수의 ELSIF 절을 사용 가능, ELSE 절은 하나 만 사용해야 한다.
- ☐ IF~THEN 문, IF~THEN~ELSE 문, IF~THEN~ELSIF 문

# 조건 제어 (Conditional Control)

## □ IF ~ THEN 문

- ✗ PL/SQL 블록이 조건이 참(TRUE)일 경우에만 조건문을 실행하는 구문
- ✗ 조건이 거짓(FALSE)이거나 NULL이면 PL/SQL은 조건문을 무시
- ✗ 조건이 참일 경우나 거짓일 경우, 어느 경우에도 제어는 END IF 다음의 문장에서 시작.

```
IF 조건문 THEN  
    조건이 참인 경우 실행할 문장들 ;  
END IF;
```

## □ IF ~ THEN ~ ELSE 문

- ✗ 조건이 TRUE이면 THEN 이하의 문장을 실행하고, 조건이 FALSE나 NULL이면 ELSE 이하의 문장을 실행

```
IF 조건문 THEN  
    조건이 참인 경우 실행할 문장들 ;  
ELSE  
    조건이 거짓인 경우 실행할 문장들 ;  
END IF;
```

# 조건 제어 (Conditional Control)

- ✍ THEN 절과 ELSE 절 안에 또 다른 IF 문을 중첩하여 사용 가능
  - 🌀 중첩된 IF문은 END IF와 반드시 짝을 이루어야 함

## □ IF ~ THEN ~ ELSIF 문

- ✍ 조건이 참과 거짓 두 경우로만 나뉘지 않고 경우의 수가 2개 이상일 경우에 사용하는 제어 구조

```
IF 조건문 THEN
    조건문이 참인 경우 실행할 문장들 ;
ELSIF 조건문_1 THEN
    조건문_1 이 참인 경우 실행할 문장들 ;
ELSE
    위 조건이 모두 거짓인 경우 실행할 문장들 ;
END IF;
```

# 조건 제어 (Conditional Control)

## ← CASE 문

- ☑ 조건에 따라 실행할 문장을 선택
- ☑ 실행할 문장은 CASE 절에 명시된 선택자(selector)에 의해 이루어짐

```
[<<label_name>>] CASE selector  
    WHEN expression1 THEN  
    WHEN expression2 THEN  
    ...  
    WHEN expressionN THEN  
    [ELSE sequence_of_statementsN+  
END CASE [label_name];
```

# IF문 예제 프로시저

```
SQL> CREATE OR REPLACE PROCEDURE Dept_Search (p_empno IN
      emp.empno%TYPE )
IS
  v_deptno emp.deptno%type ;
BEGIN
  SELECT deptno INTO v_deptno FROM emp WHERE empno = p_empno ;
  IF v_deptno <= 7000 THEN
    DBMS_OUTPUT.PUT_LINE( ' ACCOUNTING 부서 직원입니다. ' );
  ELSIF v_deptno < 7900 THEN
    DBMS_OUTPUT.PUT_LINE( ' RESEARCH 부서 직원입니다. ' );
  ELSE
    DBMS_OUTPUT.PUT_LINE( ' 부서가 없네요... ' );
  END IF ;
END ;
/

SET SERVEROUTPUT ON ; -- (DBMS_OUTPUT.PUT_LINE을 출력하기 위해 사용)
EXECUTE Dept_Search(7499);
```

## ■ 커서란 무엇인가? 명시적 커서(Explicit Cursor)

◆ 커서는 Private SQL의 작업영역 입니다.

◆ 오라클 서버에 의해 실행되는 모든 SQL문은 연관된 각각의 커서를 소유하고 있습니다.

◆ 커서의 종류

- 암시적 커서 : 모든 DML과 PL/SQL SELECT문에 대해 선언

- 명시적 커서 : 프로그래머에 의해 선언되며 이름이 있는 커서

SQL의 결과가 한 행이 아니고 여러 행일 경우에 하나씩 읽어서 처리하기 위하여 CURSOR개념이 생겼음

## ■ Explicit Cursor의 흐름도?



# 암시적인 커서

- 암시적인 커서는 오라클이나 PL/SQL 실행 메커니즘에 의해 처리되는 SQL 문장이 처리되는 곳에 대한 익명의 에드레스입니다. 오라클 데이터 베이스에서 실행되는 모든 SQL 문장은 암시적인 커서이며 그것들과 함께 모든 암시적인 커서 속성이 사용될 수 있습니다.
- 암시적 커서의 속성
  - ◆ SQL%ROWCOUNT : 해당 SQL 문에 영향을 받는 행의 수
  - ◆ SQL%FOUND : 해당 SQL 영향을 받는 행의 수가 1개 이상일 경우 TRUE
  - ◆ SQL%NOTFOUND : 해당 SQL 문에 영향을 받는 행의 수가 없을 경우 TRUE
  - ◆ SQL%ISOPEN : 항상 FALSE, 암시적 커서가 열려 있는지의 여부 검색(암시적 커서는 SQL 문이 실행되는 순간 자동으로 열림과 닫힘 실행)



# 암시적 커서 예제

```
CREATE OR REPLACE PROCEDURE Implicit_Cursor (p_empno emp.empno%TYPE)
is
  v_sal emp.sal%TYPE;
  v_update_row NUMBER;
BEGIN
  SELECT sal INTO v_sal FROM emp WHERE empno = p_empno ;
  -- 검색된 데이터가 있을경우
  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE('검색한 데이터가 존재합니다 : '||v_sal);
  END IF;
  UPDATE emp SET sal = sal*1.1 WHERE empno = p_empno;
  -- 수정한 데이터의 카운트를 변수에 저장
  v_update_row := SQL%ROWCOUNT;
  DBMS_OUTPUT.PUT_LINE('급여가 인상된 직원 수 : '|| v_update_row);
END;
/

SET SERVEROUTPUT ON ; -- (DBMS_OUTPUT.PUT_LINE을 출력용)
EXECUTE Implicit_Cursor(7369);
```

# ■ 문법(Syntax)

CURSOR cursor\_name IS  
SELECT statement

## ■ 커서 열기(OPEN)

◆ 커서의 열기는 OPEN문을 사용합니다.

◆ 커서안의 검색이 실행되며 아무런 데이터행을 추출하지 못해도 에러가 발생하지 않습니다.

OPEN cursor\_name;

## ■ 커서 패치(FETCH)

◆ 커서의 FETCH는 현재 데이터 행을 OUTPUT변수에 반환합니다.

◆ 커서의 SELECT문의 컬럼의 수와 OUTPUT변수의 수가 동일.

◆ 커서 컬럼의 변수의 타입과 OUTPUT변수의 데이터 타입도 동일

◆ 커서는 한 라인씩 데이터를 패치

FETCH cursor\_name INTO variable1, variable2 ;

## ■ 커서 닫기(CLOSE)

◆ 사용을 마친 커서는 반드시 닫아 주어야 합니다.

◆ 필요하다면 커서를 다시 열 수 있습니다.

◆ 커서를 닫은 상태에서 FETCH를 할 수 없습니다.

CLOSE cursor\_name;

# Explicit Cursor 예제

```
CREATE OR REPLACE PROCEDURE ExpCursor_Test
```

```
(v_deptno in dept.deptno%TYPE)
```

```
IS
```

```
CURSOR dept_avg IS
```

```
SELECT b.dname, COUNT(a.empno) cnt, ROUND(AVG(a.sal),3) salary
```

```
FROM emp a, dept b
```

```
WHERE a.deptno = b.deptno
```

```
AND b.deptno = v_deptno
```

```
GROUP BY b.dname ;
```

```
-- 커서를 패치하기 위한 변수 선언
```

```
v_dname dept.dname%TYPE;
```

```
emp_cnt NUMBER;
```

```
sal_avg NUMBER;
```

```
BEGIN
```

```
OPEN dept_avg;
```

```
-- 커서의 패치
```

```
FETCH dept_avg INTO v_dname, emp_cnt, sal_avg;
```

```
DBMS_OUTPUT.PUT_LINE('부서명 : ' || v_dname);
```

```
DBMS_OUTPUT.PUT_LINE('직원수 : ' || emp_cnt);
```

```
DBMS_OUTPUT.PUT_LINE('평균급여 : ' || sal_avg);
```

```
-- 커서의 CLOSE
```

```
CLOSE dept_avg;
```

```
EXCEPTION
```

```
WHEN OTHERS THEN
```

```
DBMS_OUTPUT.PUT_LINE(SQLERRM||'에러 발생 ');
```

```
END;
```

```
/
```

```
)
```

```
SQL> EXECUTE ExpCursor_Test(30)
```

```
부서명 : SALES
```

```
직원수 : 6
```

```
평균급여 : 1550.833
```

# FOR문에서 커서 사용(Cursor FOR Loops)

- ◆ FOR문을 사용하면 커서의 OPEN, FETCH, CLOSE가 자동 발생하므로 따로 기술할 필요가 없습니다
- ◆ 레코드 이름도 자동 선언되므로 따로 선언할 필요가 없습니다.

```
FOR record_name IN cursor_name LOOP
    statement 1
    statement 2
    ....
END LOOP;
```

# FOR문에서 커서 사용 예제

```
CREATE OR REPLACE PROCEDURE ForCursor_Test
IS
    CURSOR dept_sum IS
        SELECT b.dname, COUNT(a.empno) cnt, SUM(a.sal) salary
        FROM emp a, dept b
        WHERE a.deptno = b.deptno
        GROUP BY b.dname;
BEGIN
    -- Cursor를 FOR문에서 실행시킨다
    FOR emp_list IN dept_sum LOOP
        DBMS_OUTPUT.PUT_LINE('부서명 : ' || emp_list.dname);
        DBMS_OUTPUT.PUT_LINE('사원수 : ' || emp_list.cnt);
        DBMS_OUTPUT.PUT_LINE('급여합계 : ' || emp_list.salary);
    END LOOP;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE(SQLERRM||'에러 발생 ');
END;
/
```

```
SQL> EXECUTE ForCursor_Test;
```

## ■ 명시적 커서의 속성(Explicit Cursor Attributes)

### ◆%ISOPEN

- 커서가 OPEN되어 있으면 TRUE
- %ISOPEN속성을 이용하여 커서가 열려있는지 알 수 있습니다.

### ◆%NOTFOUND

- 패치한 데이터가 행을 반환하지 않으면 TRUE
- %NOTFOUND속성을 이용하여 루프를 종료할 시점을 찾습니다.

### ◆%FOUND

- 패치한 데이터가 행을 반환하면 TRUE

### ◆%ROWCOUNT

- 현재까지 반환된 모든 데이터 행의 수
- %ROWCOUNT속성을 이용하여 정확한 숫자만큼의 행을 추출합니다.

# 커서의 속성 예제

```
CREATE OR REPLACE PROCEDURE AttrCursor_Test
IS
    v_empno emp.empno%TYPE;
    v_ename emp.ename%TYPE;
    v_sal emp.sal%TYPE;
    CURSOR emp_list IS SELECT empno, ename, sal FROM emp;
BEGIN
    OPEN emp_list;
    LOOP
        FETCH emp_list INTO v_empno, v_ename, v_sal;
        -- 데이터를 찾지 못하면 빠져 나갑니다
        EXIT WHEN emp_list%NOTFOUND;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('전체데이터 수 ' || emp_list%ROWCOUNT);
    CLOSE emp_list;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('ERR MESSAGE : ' || SQLERRM);
END;
/
SQL> EXECUTE AttrCursor_Test;
```



# 파라미터가 있는 커서(Cursors with Parameters)

- ◆ 커서가 열리고 질의가 실행되면 매개 변수 값을 커서에 전달한다.
- ◆ 다른 active set을 원할때 마다 explicit커서를 따로 선언해야 한다  
프로시저 실행할 때 넘기는 변수가 아닌 프로시저 안에서  
select된 값에 의해 커서의 조건이 달라질 경우 사용  
외부의 변수를 커서의 SELECT문 안에서는 사용할 수 없음.  
그렇기 때문에 따로 파라미터를 정의해줘서 전달함.

## ■ 문법(Syntax)

```
CURSOR cursor_name  
[(parameter_name datatype, ...)]  
IS  
SELECT statement
```



# 파라미터가 있는 커서 예제

```
CREATE OR REPLACE PROCEDURE ParamCursor_Test
(param_deptno in emp.deptno%TYPE)
IS
  CURSOR emp_list IS SELECT ename FROM emp
    WHERE deptno = param_deptno;
BEGIN
  DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE('==== 입력한 부서에 해당하는 사람들
  ==== ');
  FOR emplst IN emp_list LOOP
    DBMS_OUTPUT.PUT_LINE('이름 : ' || emplst.ename);
  END LOOP;
  EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('ERR MESSAGE : ' || SQLERRM);
  END;
/

EXECUTE ParamCursor_Test(10);
```

## ◆ WHERE CURRENT OF

- ROWID를 이용하지 않고도 현재 참조하는 행을 갱신하고 삭제할 수 있게 합니다.
- 추가적으로 FETCH문에 의해 가장 최근에 처리된 행을 참조하기 위해서 "WHERE CURRENT OF 커서이름 " 절로 DELETE나 UPDATE 문 작성이 가능합니다.
- 이 절을 사용할 때 참조하는 커서가 있어야 하며, FOR UPDATE절이 커서 선언 query문장 안에 있어야 합니다. 그렇지 않으면 error가 발생합니다

# WHERE CURRENT OF 예제

```
CREATE OR REPLACE PROCEDURE where_current  
IS
```

```
CURSOR emp_list IS
```

```
    SELECT empno FROM emp
```

```
    WHERE empno = 7934      FOR UPDATE;
```

```
BEGIN
```

```
--DBMS_OUTPUT.PUT_LINE명령을 사용하기 위해서
```

```
DBMS_OUTPUT.ENABLE;
```

```
FOR emplst IN emp_list LOOP
```

```
    --emp_list커서에 해당하는 사람의 직업을 SALESMAN으로 업데이트
```

```
    UPDATE emp SET job = 'SALESMAN' WHERE CURRENT OF emp_list;
```

```
    DBMS_OUTPUT.PUT_LINE('수정 성공');
```

```
END LOOP;
```

```
EXCEPTION
```

```
    WHEN OTHERS THEN
```

```
        -- 에러 발생시 에러 메시지 출력
```

```
        DBMS_OUTPUT.PUT_LINE('ERR MESSAGE : ' || SQLERRM);
```

```
END;
```

```
/
```

비교 확인

```
SELECT job FROM emp WHERE empno = 7934;
```

```
EXECUTE where_current;
```

```
SELECT job FROM emp WHERE empno = 7934;
```

# 연습문제

매개변수로 부서번호를 넘기면 해당 부서에 속한 사원번호, 사원이름, 부서이름을 출력하는 프로시저를 만들어보자. 단, 사원번호, 사원이름, 부서이름 정보를 조회해 가져와 중첩 테이블에 담은 다음, 다시 이 중첩 테이블에 들어간 데이터를 출력하도록 작성해보자.

<정답>

```
create or replace procedure emp_proc (v_deptno in dept.deptno%type)
```

```
is
```

```
  cursor c1 is select empno,ename,dname from emp e,dept d where  
e.deptno=d.deptno;
```

```
  type nt_emp is table of c1%rowtype; -- 사번,이름,부서명을 여러건 저장할 수 있는 데이  
터형
```

```
  v_emp nt_emp; -- v_emp변수는 사번,이름,부서명을 여러개 저장할 수 있는 변수
```

```
begin -- bulk collect 여러건을 데이터를 한번에 처리
```

```
  select empno,ename,dname bulk collect into v_emp from emp e, dept d  
  where e.deptno=d.deptno and d.deptno=v_deptno;
```

```
  for i in v_emp.first .. v_emp.last loop
```

```
    dbms_output.put_line('사번 : '||v_emp(i).empno);
```

```
    dbms_output.put_line('이름 : '||v_emp(i).ename);
```

```
    dbms_output.put_line('부서명 : '||v_emp(i).dname);
```

```
    dbms_output.put_line('=====');
```

```
  end loop;
```

```
end;
```

```
/
```

```
exec emp_proc(10);
```

# ■ 예외(Exception)란?

- ◆ 오라클 PL/SQL의 오류를 예외라고 부릅니다.
- ◆ 오류는 PL/SQL을 컴파일 할때 문법적인 오류로 발생하는 컴파일 타임 오류와, 프로그램을 실행할때 발생하는 실행타임 오류로 구분할 수 있습니다.

예 외	설 명	처 리
미리 정의된 오라클 서버 오류 (Predefined Oracle Server)	PL/SQL에서 자주 발생하는 약20개의 오류	선언할 필요도 없고, 발생시에 예외 절로 자동 트랩(Trap)된다.
미리 정의되지 않은 오라클 서버 오류 (Non-Predefined Oracle Server)	미리 정의된 오라클 서버 오류를 제외한 모든 오류	선언부에서 선언해야 하고 발생시 자동 트랩된다.
사용자 정의 오류 (User-Defined)	개발자가 정한 조건에 만족하지 않을경우 발생하는 오류	선언부에서 선언하고 실행부에서 <b>RAISE</b> 문을 사용하여 발생시켜야 한다

## ■ Exception 문법(Syntax)

- ◆ WHEN OTHERS절은 맨 마지막에 옵니다.
- ◆ 예외 처리절은 EXCEPTION부터 시작합니다.
- ◆ 허용합니다.
- ◆ 예외가 발생하면 여러 개의 예외 처리부 중에 하나의 예외 처리부에 트랩(Trap)됩니다.

### EXCEPTION

```
WHEN 예외1[OR 예외2] THEN  
    statements 1  
    statements 2...
```

```
[ WHEN 예외3[OR 예외4] THEN  
    statements1... ]
```

```
[ WHEN OTHERS THEN  
    statements1... ]
```

# 미리 정의된 예외(Predefined Exceptions)

- ◆ 오라클 PL/SQL은 자주 일어나는 몇가지 예외를 미리 정의해 놓았으며, 이러한 예외는 개발자가 따로 선언할 필요가 없습니다.

## ▣ 미리 정의된 예외의 종류?

- ◆ NO\_DATA\_FOUND : SELECT문이 아무런 데이터 행을 반환하지 못할 때
- ◆ TOO\_MANY\_ROWS : 하나만 리턴해야하는 SELECT문이 하나 이상의 행을 반환할 때
- ◆ INVALID\_CURSOR : 잘못된 커서 연산
- ◆ ZERO\_DIVIDE : 0으로 나눌때
- ◆ DUP\_VAL\_ON\_INDEX : UNIQUE 제약을 갖는 컬럼에 중복되는 데이터가 INSERT될 때 이 외에도 몇 개가 더 있습니다.



# 미리 정의된 예외

예외	오라클 에러 (SQLCODE)	설명
ACCESS_INTO_NULL	ORA-06530 (-6530)	초기화되지 않은 객체에 값을 할당하는 경우
CASE_NOT_FOUND	ORA-06592 (-6592)	CASE 문에 ELSE 절이 없는 경우
CURSOR_ALREADY_OPEN	ORA-06511 (-6511)	이미 열려있는 커서를 다시 열려고 하는 경우
DUP_VAL_ON_INDEX	ORA-00001 (-1)	UNIQUE 제약을 가지는 컬럼에 중복되는 데이터를 삽입하려고 하는 경우
INVALID_CURSOR	ORA-01001 (-1001)	잘못된 커서 연산을 수행하려고 하는 경우
INVALID_NUMBER	ORA-01722(-1722)	잘못된 숫자를 표현한 경우
LOGIN_DENIED	ORA-01017 (-1017)	사용자 아이디와 암호를 가지고 오라클에 로그인하는 경우
NO_DATA_FOUND	ORA-01403 (+100)	SELECT 문이 반환할 데이터 행이 없는 경우
NOT_LOGGED_ON	ORA-01012 (-1012)	오라클에 연결되지 않은 데이터베이스를 호출하는 경우
PROGRAM_ERROR	ORA-06501(-6501)	내부 PL/SQL 오류



# 미리 정의된 예외

예외	오라클 에러 (SQLCODE)	설명
SELF_IS_NULL	ORA-30625 (-30625)	<b>NULL</b> 인스턴스에 대해 <b>MEMBER</b> 메소드를 호출한 경우
STORAGE_ERROR	ORA-06500(-6500)	메모리 부족으로 일어나는 <b>PL/SQL</b> 내부 오류
SUBSCRIPT_BEYOND_COUNT	ORA-06533 (-6533)	엘리먼트의 수보다 큰 인덱스를 가지고 중첩된 테이블이나 변수를 호출한 경우
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532 (-6532)	범위 이외의 수를 가지고 중첩된 테이블이나 변수를 호출한 경우
TIMEOUT_ON_RESOURCE	ORA-00051 (-51)	자원을 기다리는 동안 <b>TIME-OUT</b> 이 발생한 경우
TOO_MANY_ROWS	ORA-01422 (-1422)	<b>SELECT</b> 문이 하나 이상의 행을 반환하는 경우
VALUE_ERROR	ORA-06502 (-6502)	숫자의 계산, 변환, 버림 등에서 발생한 오류
ZERO_DIVIDE	ORA-01476 (-1476)	<b>0</b> 으로 나누려 하는 경우

# 미리 정의된 예외 예제

```
CREATE OR REPLACE PROCEDURE PreException_test
(v_deptno IN emp.empno%TYPE)
IS
    v_emp emp%ROWTYPE;
BEGIN
    SELECT empno, ename, deptno
    INTO v_emp.empno, v_emp.ename, v_emp.deptno
    FROM emp
    WHERE deptno = v_deptno ;
    DBMS_OUTPUT.PUT_LINE('사번 : ' || v_emp.empno);
    DBMS_OUTPUT.PUT_LINE('이름 : ' || v_emp.ename);
    DBMS_OUTPUT.PUT_LINE('부서번호 : ' || v_emp.deptno);
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE('데이터가 존재 합니다.');
```

DBMS\_OUTPUT.PUT\_LINE('DUP\_VAL\_ON\_INDEX 에러 발생');

```
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('TOO_MANY_ROWS에러 발생');
```

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('NO\_DATA\_FOUND에러 발생');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('기타 에러 발생');

```
END;
/
```

# 실행결과

```
SQL> SET SERVEROUTPUT ON ; -- (출력하기 위해 사용)
SQL> EXECUTE PreException_Test(20);
TOO_MANY_ROWS에러 발생
PL/SQL 처리가 정상적으로 완료되었습니다.
```

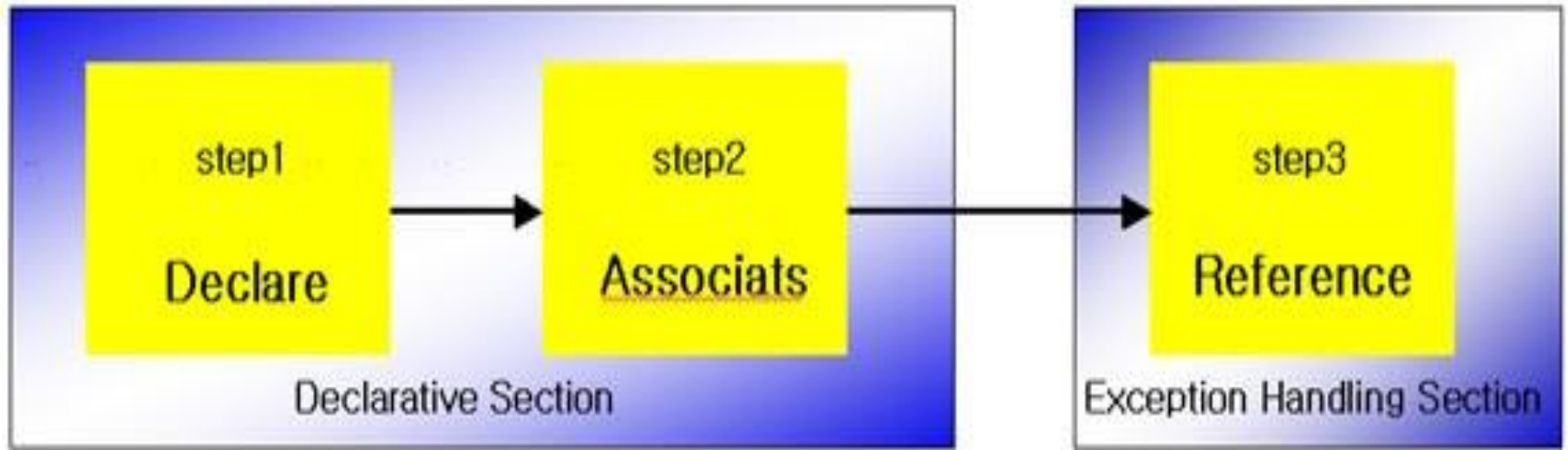
※ TOO\_MANY\_ROWS에러를 타는 이유?

- SELECT문의 결과가 1개 이상의 행을 리턴하기 때문이다..
- TOO\_MANY\_ROWS를 피하기 위해서는 FOR문이나 LOOP문으로 SELECT문을 처리해야 합니다.

```
FOR emp_list IN
  (SELECT empno, ename, deptno
   FROM emp
   WHERE deptno = v_deptno) LOOP

  DBMS_OUTPUT.PUT_LINE('사번 : ' || emp_list.empno);
  DBMS_OUTPUT.PUT_LINE('이름 : ' || emp_list.ename);
  DBMS_OUTPUT.PUT_LINE('부서번호 : ' || emp_list.deptno);
END LOOP;
```

# 미리 정의되지 않은 예외(Non-Predefined Exceptions)



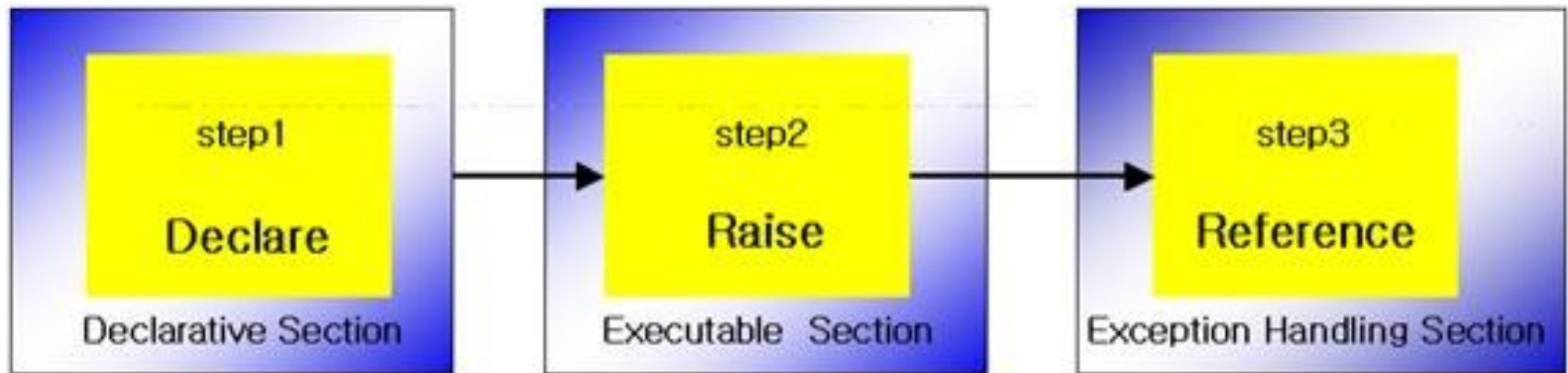
- ◆ STEP 1 : 예외의 이름을 선언(선언절)
- ◆ STEP 2 : PRAGMA EXCEPTION\_INIT문장으로 예외의 이름과 오라클 서버 오류 번호를 결합(선언절)
- ◆ STEP 3 : 예외가 발생할 경우 해당 예외를 참조한다(예외절)

# 미리 정의되지 않은 예외 예제

```
CREATE OR REPLACE PROCEDURE NonPreException_Test
IS
    not_null_test EXCEPTION;    -- STEP 1
    /* not_null_test는 선언된 예외 이름
       -1400 Error 처리번호는 표준 Oracle7 Server Error 번호 */
    PRAGMA EXCEPTION_INIT(not_null_test, -1400);    -- STEP 2
BEGIN
    DBMS_OUTPUT.ENABLE;
    -- empno를 입력하지 않아서 NOT NULL 에러 발생
    INSERT INTO emp(ename, deptno)
    VALUES('tiger', 30);
    EXCEPTION
    WHEN not_null_test THEN    -- STEP 3
        DBMS_OUTPUT.PUT_LINE('not null 에러 발생 ');
END;
/
SQL> EXECUTE NonPreException_Test;
not null 에러 발생
```

# 사용자 정의 예외(User-Defined Exceptions)

- ◆ 오라클 저장함수 RAISE\_APPLICATION\_ERROR를 사용하여 오류코드 -20000부터 20999의 범위 내에서 사용자 정의 예외를 만들수 있습니다.



- ◆ STEP 1 : 예외의 이름을 선언(선언절)
- ◆ STEP 2 : RAISE문을 사용하여 직접적으로 예외를 발생시킨다(실행절)
- ◆ STEP 3 : 예외가 발생할 경우 해당 예외를 참조한다(예외절)



# 사용자 정의 예외 예제 Procedure

```
CREATE OR REPLACE PROCEDURE User_Exception
  (v_deptno IN emp.deptno%type )
IS
  -- 예외의 이름을 선언
  user_define_error EXCEPTION;    -- STEP 1
  cnt    NUMBER;
BEGIN
  DBMS_OUTPUT.ENABLE;
  SELECT COUNT(empno)
  INTO cnt
  FROM emp
  WHERE deptno = v_deptno;
  IF cnt < 5 THEN
    -- RAISE문을 사용하여 직접적으로 예외를 발생시킨다
    RAISE user_define_error;        -- STEP 2
  END IF;
EXCEPTION
  -- 예외가 발생할 경우 해당 예외를 참조한다.
  WHEN user_define_error THEN      -- STEP 3
    RAISE_APPLICATION_ERROR(-20001, '부서에 사원이 몇명 안되네요..');
END;
/
```

```
SQL> EXECUTE user_exception(10); -- 10부서 사원이 5보다 적어서 사용자 정의 예외
SQL> EXECUTE user_exception(20); -- 20부서로 실행을 하면 정상
```

# SQLCODE, SQLERRM

- ◆ WHEN OTHERS문으로 트랩(Trap)되는 오류들의 실제 오류 코드와 설명
- ◆ SQLCODE : 실행된 프로그램이 성공적으로 종료하였을때는 오류번호 0을 포함하며, 그렇지 못할 경우에는 해당 오류코드 번호를 포함
- ◆ SQLERRM : SQLCODE에 포함된 오라클 오류 번호에 해당하는 메시지

SQLCODE Value	Description
0	오류 없이 성공적으로 종료
1	사용자 정의 예외 번호
+100	<b>NO_DATA_FOUND</b> 예외 번호
음수	위에 것을 제외한 오라클 서버 에러 번호



# SQLCODE, SQLERRM 예제 프로시저

```
CREATE OR REPLACE PROCEDURE Errcode_Exception
(v_deptno IN emp.deptno%type )
IS
  v_emp  emp%ROWTYPE ;
BEGIN
  -- ERROR발생 for문을 돌려야 됨
  SELECT * INTO v_emp      FROM emp
    WHERE deptno = v_deptno;
  DBMS_OUTPUT.PUT_LINE('사번 : ' || v_emp.empno);
  DBMS_OUTPUT.PUT_LINE('이름 : ' || v_emp.ename);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('ERR CODE : ' || TO_CHAR(SQLCODE));
    DBMS_OUTPUT.PUT_LINE('ERR MESSAGE : ' || SQLERRM);
END;
/

EXECUTE Errcode_Exception(30);
```

# package?

- ◆ 패키지(package)는 오라클 데이터베이스에 저장되어 있는 서로 관련있는 PL/SQL 프로시저와 함수들의 집합 입니다
- ◆ 패키지는 선언부와 본문 두 부분으로 나누어 집니다.

## 패키지 선언부

- 선언절은 패키지에 포함될 PL/SQL 프로시저나, 함수, 커서, 변수, 예외 절을 선언 합니다.
- 패키지 선언부에서 선언한 모든 요소들은 패키지 전체에 적용됩니다.
- 즉 선언부에서 선언한 변수는 PUBLIC 변수로 사용 됩니다.

```
CREATE [OR REPLACE] PACKAGE package_name IS | AS
    [ 변수선언절 ]
    [ 커서선언절 ]
    [ 예외선언절 ]
    [ Procedure 선언절 ]
    [ Function 선언절 ]
END package_name ;
```

# 패키지 선언부

```
CREATE OR REPLACE PACKAGE hr_pkg IS
  -- 사번을 받아 이름을 반환하는 함수
  FUNCTION fn_get_emp_name ( pn_employee_id IN NUMBER )
    RETURN VARCHAR2;
  -- 신규 사원 입력
  PROCEDURE new_emp_proc ( ps_emp_name   IN VARCHAR2,
                          pd_hire_date IN VARCHAR2 );
  -- 퇴사 사원 처리
  PROCEDURE retire_emp_proc ( pn_employee_id IN NUMBER );
END hr_pkg;
/
```

# 패키지 선언부

```
CREATE OR REPLACE PACKAGE ch_var IS
  -- 상수선언
  c_test CONSTANT VARCHAR2(10) := 'TEST';
  -- 변수선언
  v_test VARCHAR2(10);
END ch_var;
BEGIN
  DBMS_OUTPUT.PUT_LINE('상수 ch_var.c_test = ' || chvar.c_test);
  DBMS_OUTPUT.PUT_LINE('변수 ch_var.c_test = ' || ch_var.v_test);
END;
BEGIN
  DBMS_OUTPUT.PUT_LINE('값 설정 이전 = ' || ch_var.v_test);
  ch12_var.v_test := 'FIRST';
  DBMS_OUTPUT.PUT_LINE('값 설정 이후 = ' || ch_var.v_test);
END;
-- 신규세션
BEGIN
  DBMS_OUTPUT.PUT_LINE('ch_var.v_test = ' || ch_var.v_test);
END;
```

## 패키지 본문

- 패키지 본문은 패키지에서 선언된 부분의 실행을 정의 합니다.
- 즉 실제 프로시저나 함수의 내용에 해당하는 부분이 옵니다.

```
CREATE [OR REPLACE] PACKAGE BODY package_name IS | AS  
  [ 변수선언절 ]  
  [ 커서선언절 ]  
  [ 예외선언절 ]  
  [ Procedure 선언절 ]  
  [ Function 선언절 ]  
END package_name ;
```

## 간단한 패키지 예제

프로시저명	프로시저 기능	보기
<code>all_emp_info</code>	모든 사원의 사원 정보 (사번, 성명, 입사일)	
<code>all_sal_info</code>	모든 사원의 급여 정보 (평균급여, 최고급여, 최소급여)	
<code>dept_emp_info</code>	특정 부서의 사원 정보 (사번, 성명, 입사일)	
<code>dept_sal_info</code>	특정 부서의 급여 정보 (평균급여, 최고급여, 최소급여)	

# package 예제 (선언부)

```
CREATE OR REPLACE PACKAGE emp_info1 AS
    -- 모든 사원의 사원 정보
    PROCEDURE all_emp_info;
    -- 모든 사원의 급여 정보
    PROCEDURE all_sal_info;
    -- 특정 부서의 사원 정보
    PROCEDURE dept_emp_info (v_deptno IN  NUMBER) ;
    -- 특정 부서의 급여 정보
    PROCEDURE dept_sal_info (v_deptno IN  NUMBER) ;
END;
/
```

# package 예제 (본문)

```
CREATE OR REPLACE PACKAGE BODY emp_info1 AS
-- 모든 사원의 사원 정보
  PROCEDURE all_emp_info
  IS
    CURSOR emp_cursor IS
      SELECT empno, ename, to_char(hiredate, 'RRRR/MM/DD') hiredate
      FROM emp
      ORDER BY hiredate;
  BEGIN
    FOR aa IN emp_cursor LOOP
      DBMS_OUTPUT.PUT_LINE('사번 : ' || aa.empno);
      DBMS_OUTPUT.PUT_LINE('성명 : ' || aa.ename);
      DBMS_OUTPUT.PUT_LINE('입사일 : ' || aa.hiredate);
    END LOOP;
  EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE(SQLERRM||'에러 발생 ');
      END all_emp_info;
-- 모든 사원의 급여 정보
  PROCEDURE all_sal_info
```

```

IS
    CURSOR emp_cursor IS
    SELECT round(avg(sal),3) avg_sal, max(sal) max_sal, min(sal) min_sal
    FROM emp;
BEGIN
    FOR aa IN emp_cursor LOOP
        DBMS_OUTPUT.PUT_LINE('전체 급여 평균 : ' || aa.avg_sal);
        DBMS_OUTPUT.PUT_LINE('최대 급여 금액 : ' || aa.max_sal);
        DBMS_OUTPUT.PUT_LINE('최소 급여 금액 : ' || aa.min_sal);
    END LOOP;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE(SQLERRM||'에러 발생 ');
        END all_sal info;
--특정 부서의 자원 정보
PROCEDURE dept_emp_info (v_deptno IN  NUMBER)
IS
    CURSOR emp_cursor IS
    SELECT empno, ename, to_char(hiredate, 'RRRR/MM/DD') hiredate
    FROM emp
    WHERE deptno = v_deptno
    ORDER BY hiredate;
BEGIN
    FOR aa IN emp_cursor LOOP
        DBMS_OUTPUT.PUT_LINE('사번 : ' || aa.empno);
        DBMS_OUTPUT.PUT_LINE('성명 : ' || aa.ename);
        DBMS_OUTPUT.PUT_LINE('입사일 : ' || aa.hiredate);
    END LOOP;

```



EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE(SQLERRM||'에러 발생 ');

END dept\_emp\_info;

--특정 부서의 급여 정보

PROCEDURE dept\_sal\_info (v\_deptno IN NUMBER)

IS

CURSOR emp\_cursor IS

SELECT round(avg(sal),3) avg\_sal, max(sal) max\_sal, min(sal) min\_sal

FROM emp

WHERE deptno = v\_deptno;

BEGIN

FOR aa IN emp\_cursor LOOP

DBMS\_OUTPUT.PUT\_LINE('전체급여평균 : ' || aa.avg\_sal);

DBMS\_OUTPUT.PUT\_LINE('최대급여금액 : ' || aa.max\_sal);

DBMS\_OUTPUT.PUT\_LINE('최소급여금액 : ' || aa.min\_sal);

END LOOP;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE(SQLERRM||'에러 발생 ');

END dept\_sal\_info;

END;

/

**SQL> exec emp\_info1.all\_emp\_info;**

exec emp\_info1.all\_sal\_info;

exec emp\_info1.dept\_emp\_info(10);

exec emp\_info1.dept\_sal\_info(10);

# 트리거란?

- ◆ INSERT, UPDATE, DELETE문이 TABLE에 대해 행해질 때 묵시적으로 수행되는 PROCEDURE 입니다.
- ◆ Trigger는 TABLE과는 별도로 DATABASE에 저장됩니다.
- ◆ Trigger는 VIEW에 대해서가 아니라 TABLE에 관해서만 정의될 수 있습니다.

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE | AFTER
  trigger_event ON table_name
[ FOR EACH ROW ]
[ WHEN (condition) ]
PL/SQL block
```

- BEFORE : INSERT, UPDATE, DELETE문이 실행되기 전에 트리거가 실행됩니다.
- AFTER : INSERT, UPDATE, DELETE문이 실행된 후 트리거가 실행됩니다.
- trigger\_event : INSERT, UPDATE, DELETE 중에서 한 개 이상 올 수 있습니다.
- FOR EACH ROW : 이 옵션이 있으면 행 트리거가 됩니다.
- 행 트리거 : 컬럼의 각각의 행의 데이터 행 변화가 생길 때마다 실행되며, 그 데이터 행의 실제 값을 제어할 수 있습니다.
- 문장 트리거 : 트리거 사건에 의해 단 한번 실행되며, 컬럼의 각 데이터 행을 제어할 수 없습니다.

# 문장 트리거와 행 트리거

□ FOR EACH ROW 옵션절의 사용 유무에 따라 문장 트리거(Statement-Level Trigger)와 행 트리거(Row-Level Trigger)로 구분

✂ FOR EACH ROW 옵션절을 사용한 트리거가 행 트리거

☑ 문장 트리거

✂ 트리거링 사건에 대해 딱 한번만 실행

✂ 컬럼의 각 데이터 행을 제어할 수 없다

☞ 컬럼의 데이터 값에 관계없이 컬럼에 변화가 일어남을 감지하여 실행되는 트리거

☑ 행 트리거

✂ 컬럼의 각 데이터 행이 변경될 때마다 실행

✂ 실제 그 데이터 행의 값을 제어할 수 있는 트리거

✂ 실제 값을 수정, 변경 또는 저장하기 위해 사용.

✂ 행 트리거에서 실제 데이터를 제어하기 위해 사용 하는 약자

☞ " :old"와 " :new"

✂ SQL 문에 따른 사용 방법

☞ INSERT 문의 경우 : 입력할 데이터의 값이 " :new.컬럼이름" 에 저장.

☞ UPDATE 문의 경우 : 변경 전의 데이터는 " :old.컬럼이름" 에 저장되고, 새로운 데이터 값은 " :new.컬럼이름" 에 저장.

☞ DELETE 문의 경우 : 삭제되는 컬럼 값이 " :old.컬럼이름" 에 저장.

# 문장 트리거와 행 트리거

## ← BEFORE/AFTER 옵션을 사용한 트리거링 시점 제어하기

### ☑ 오라클 트리거 유형

#### ✍ 문장 트리거

- BEFORE INSERT, AFTER INSERT, BEFORE UPDATE, AFTER UPDATE, AFTER DELETE, BEFORE DELETE

#### ✍ 행 트리거

- BEFORE INSERT, AFTER INSERT, BEFORE UPDATE, AFTER UPDATE,

SALARY 테이블

EMP_NO	EMP_NAME	SAL	BONUS	
1001	SAM	1500000	500000	→ BEFORE 문장 트리거
1002	KIM	2500000	450000	→ BEFORE 행 트리거
1003	PARK	2800000	720000	→ AFTER 행 트리거
				→ AFTER 문장 트리거

# 트리거 insert

```
create table emp02 (empno number(4) primary key, ename
varchar2(20),
sal number(10));
create or replace trigger emp_sum
after insert on emp02
declare
sum_t emp.sal%type;
begin
select sum(sal) into sum_t from emp02;
dbms_output.put_line('급여합계 :'||sum_t);
end;
/
insert into emp02 values (1116,'철수',2200);
```

# 트리거 insert

```
create table foo ( id number primary key, data varchar2(100) );
```

```
-----  
create sequence foo_seq;
```

```
-----  
create or replace trigger tri_1  
      before insert on foo      for each row  
begin  
      select foo_seq.nextval into :new.id from dual;  
end;  
/
```

```
-----  
insert into foo(data)      values('aaa');
```

```
-----  
select * from foo;
```

# 트리거 삭제와 활성화/비활성화

## ☑ 트리거 삭제와 활성화/비활성화

### ✍ 트리거 삭제

- DROP TRIGGER 문 사용

```
DROP TRIGGER 트리거이름 ;
```

### ✍ 트리거 활성화/비활성화

- ALTER TRIGGER 문 사용

```
ALTER TRIGGER 트리거이름 DISABLE ;  
ALTER TRIGGER 트리거이름 ENABLE ;
```

# 트리거

- TRIGGER가 사용되는 경우
  - 테이블 생성시 CONSTRAINT로 선언 제한이 불가능하고 복잡한 무결성 제한을 유지
  - DML문장을 사용한 사람,변경한 내용,시간 등을 기록함으로써 정보를 AUDIT하기
  - 테이블을 변경할 때 일어나야 할 동작을 다른 테이블 또는 다른 프로그램들에게 자동적으로 신호하기
- TRIGGER에 대한 제한
  - TRIGGER는 트랜잭션 제어 문(COMMIT,ROLLBACK,SAVEPOINT)장을 사용하지 못한다.
  - TRIGGER 주요부에 의해 호출되는 프로시저나 함수는 트랜잭션 제어 문장을 사용하지 못한다.
  - TRIGGER 주요부는 LONG또는 LONG RAW변수를 선언할 수 없다.
  - TRIGGER 주요부가 액세스하게 될 테이블에 대한 제한이 있다.



# 실습하기

EMP01 테이블을 생성

```
drop table emp01;
```

```
create table emp01(  
empno number(4) primary key,  
ename varchar2(10) not null,  
job varchar2(20));
```

사원 테이블에 새로운 데이터가 들어오면(즉, 신입 사원이 들어오면) 급여 테이블에 새로운 데이터(즉 신입 사원의 급여 정보)를 자동으로 생성하도록 하기 위해서 사원 테이블에 트리거를 작성해 봅시다. (신입사원의 급여는 일괄적으로 100으로 합니다. )

1. 급여를 저장할 테이블을 생성하자.

```
CREATE TABLE SAL01(  
SALNO NUMBER(4) PRIMARY KEY,  
SAL NUMBER(7,2),  
EMPNO NUMBER(4) REFERENCES EMP01(EMPNO)  
);
```

# 실습하기

2. 급여번호를 자동 생성하는 시퀀스를 정의하고 이 시퀀스로부터 일련번호를 얻어 급여번호에 부여합시다.

```
CREATE SEQUENCE SAL01_SALNO_SEQ;
```

3. 아래 처럼 TRIG01 트리거를 생성

```
CREATE OR REPLACE TRIGGER TRG_01  
AFTER INSERT  
ON EMP01  
FOR EACH ROW  
BEGIN  
INSERT INTO SAL01 VALUES(  
SAL01_SALNO_SEQ.NEXTVAL, 100, :NEW.EMPNO);  
END;  
/
```

# 실습하기

4. 사원 테이블에 데이터를 추가합니다.

```
INSERT INTO EMP01 VALUES(1, ' 홍길동', '프로그래머');
```

```
SELECT * FROM EMP01;
```

```
SELECT * FROM SAL01;
```

# 실습하기

사원이 삭제되면 그 사원의 급여 정보도 자동 삭제되는 트리거를 작성해 보도록 합시다.

1. 이번에는 사원 테이블의 로우를 삭제해보자.

```
DELETE FROM EMP01 WHERE EMPNO=1;
```

2. 사원번호 1를 급여 테이블에서 참조하고 있기 때문에 삭제가 불가능하다. 사원이 삭제되려면 그 사원의 급여 정보도 급여 테이블에서 삭제되어야 합니다. 사원의 정보가 제거 될 때 그 사원의 급여 정보도 함께 삭제하는 내용을 트리거로 작성하도록 합시다.

```
CREATE OR REPLACE TRIGGER TRG_02  
AFTER DELETE ON EMP01  
FOR EACH ROW  
BEGIN  
DELETE FROM SAL01 WHERE EMPNO=:old.EMPNO;  
END;  
/
```

# 실습하기

3. 사원 테이블에 데이터를 삭제해 봅시다.

```
DELETE FROM EMP01 WHERE EMPNO=1;  
SELECT * FROM EMP01;  
SELECT * FROM SAL01;
```

DROP TIGGER 다음에 삭제할 트리거 명을 기술합니다.

```
DROP TRIGGER TRG_01;  
DROP TRIGGER TRG_02;
```

# 간단한 행 트리거 예제

```
create or replace TRIGGER triger_test
BEFORE
  UPDATE ON dept
  FOR EACH ROW    -- old , new 사용하기 위해
BEGIN
  DBMS_OUTPUT.PUT_LINE('변경 전 컬럼 값 : ' || :old.dname);
  DBMS_OUTPUT.PUT_LINE('변경 후 컬럼 값 : ' || :new.dname);
END;
/
```

테스트

```
UPDATE dept SET dname = '총무부'
WHERE deptno = 30;
```

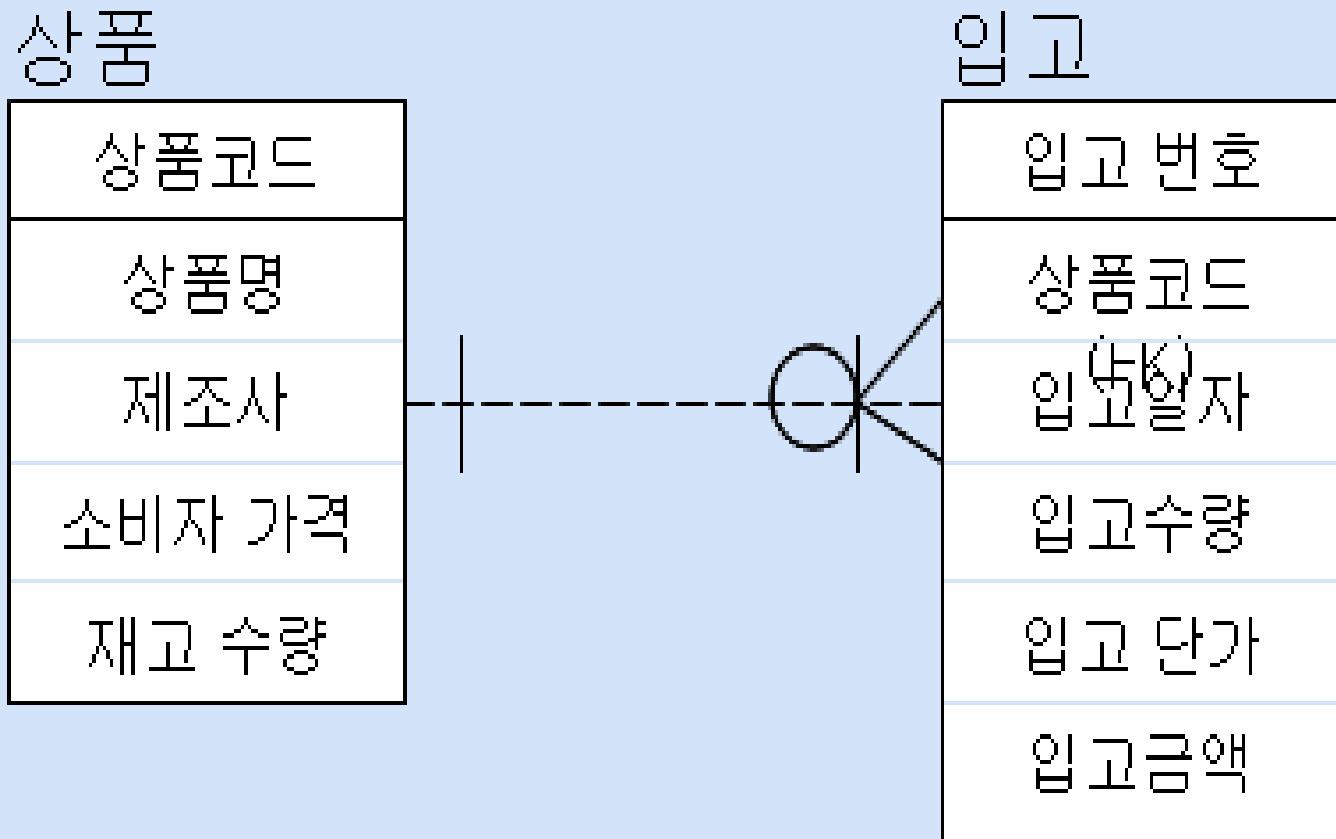
# 간단한 행 트리거 예제 (PLSQL BLOCK이 있는 트리거)

```
CREATE OR REPLACE trigger sum_trigger
after
  INSERT OR UPDATE ON emp
DECLARE
  -- 변수를 선언할 때는 DECLARE문을 사용해야 합니다
  avg_sal NUMBER;
BEGIN
  SELECT ROUND(AVG(sal),3) INTO avg_sal FROM emp;
  DBMS_OUTPUT.PUT_LINE('급여 평균 : ' || avg_sal);
END;
/
```

```
SET SERVEROUTPUT ON ; -- (출력하기 위해 사용)
INSERT INTO EMP(EMPNO, ENAME, JOB, HIREDATE, SAL,deptno)
VALUES(1000, 'LION', 'SALES', SYSDATE, 5000,20);
급여 평균 : 2073.214
```

# 예제를 통한 트리거의 적용

- 상품 테이블의 예제를 통해서 실질적인 트리거의 적용 예를 살펴보도록 합시다.





# 실습하기

입고 테이블에 상품이 입력되면 입고 수량을 상품 테이블의 재고 수량에 추가하는 트리거 작성해 봅시다.

1. 상품 테이블을 생성합시다.

```
CREATE TABLE 상품(  
  상품코드 CHAR(6) PRIMARY KEY,  
  상품명 VARCHAR2(12) NOT NULL,  
  제조사 VARCHAR(12),  
  소비자가격 NUMBER(8),  
  재고수량 NUMBER DEFAULT 0  
);
```

# 실습하기

## 2. 입고 테이블을 생성합니다.

```
CREATE TABLE 입고(  
    입고번호 NUMBER(6) PRIMARY KEY,  
    상품코드 CHAR(6) REFERENCES 상품(상품코드),  
    입고일자 DATE DEFAULT SYSDATE,  
    입고수량 NUMBER(6),  
    입고단가 NUMBER(8),  
    입고금액 NUMBER(8)  
);
```

## 3. 상품테이블의 재고수량 컬럼을 통해서 실질적인 트리거의 적용 예를 살펴보도록 하겠습니다. 우선 상품 테이블에 다음과 같은 샘플 데이터를 입력해봅시다.

```
INSERT INTO 상품(상품코드, 상품명, 제조사, 소비자가격)  
VALUES('A00001','세탁기', 'LG', 500);  
INSERT INTO 상품(상품코드, 상품명, 제조사, 소비자가격)  
VALUES('A00002','컴퓨터', 'LG', 700);  
INSERT INTO 상품(상품코드, 상품명, 제조사, 소비자가격)  
VALUES('A00003','냉장고', '삼성', 600);
```

# 실습하기

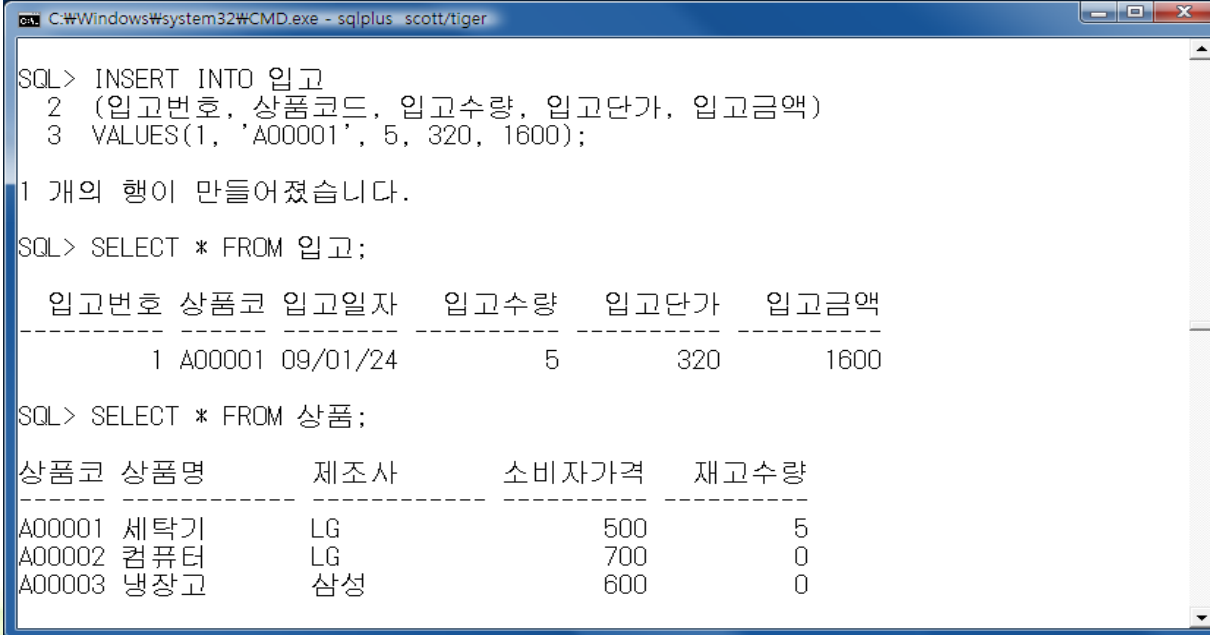
4. 입고 테이블에 상품이 입력되면 입고 수량을 상품 테이블의 재고 수량에 추가하는 트리거 작성

```
-- 입고 트리거
CREATE OR REPLACE TRIGGER TRG_03
AFTER INSERT ON 입고
FOR EACH ROW
BEGIN
UPDATE 상품
SET 재고수량 = 재고수량 + :NEW.입고수량
WHERE 상품코드 = :NEW.상품코드;
END;
/
```

# 실습하기

5. 트리거를 실행시킨 후 입고 테이블에 행을 추가합니다.  
입고 테이블에는 물론 상품 테이블의 재고 수량이 변경됨을 확인할 수 있습니다.

```
INSERT INTO 입고(입고번호, 상품코드, 입고수량, 입고단가, 입고금액)
VALUES(1, 'A00001', 5, 320, 1600);
SELECT * FROM 입고;
SELECT * FROM 상품;
```



```
C:\Windows\system32\CMD.exe - sqlplus scott/tiger

SQL> INSERT INTO 입고
  2  (입고번호, 상품코드, 입고수량, 입고단가, 입고금액)
  3  VALUES(1, 'A00001', 5, 320, 1600);

1 개의 행이 만들어졌습니다.

SQL> SELECT * FROM 입고;

   입고번호  상품코  입고일자   입고수량   입고단가   입고금액
-----
         1  A00001  09/01/24          5         320        1600

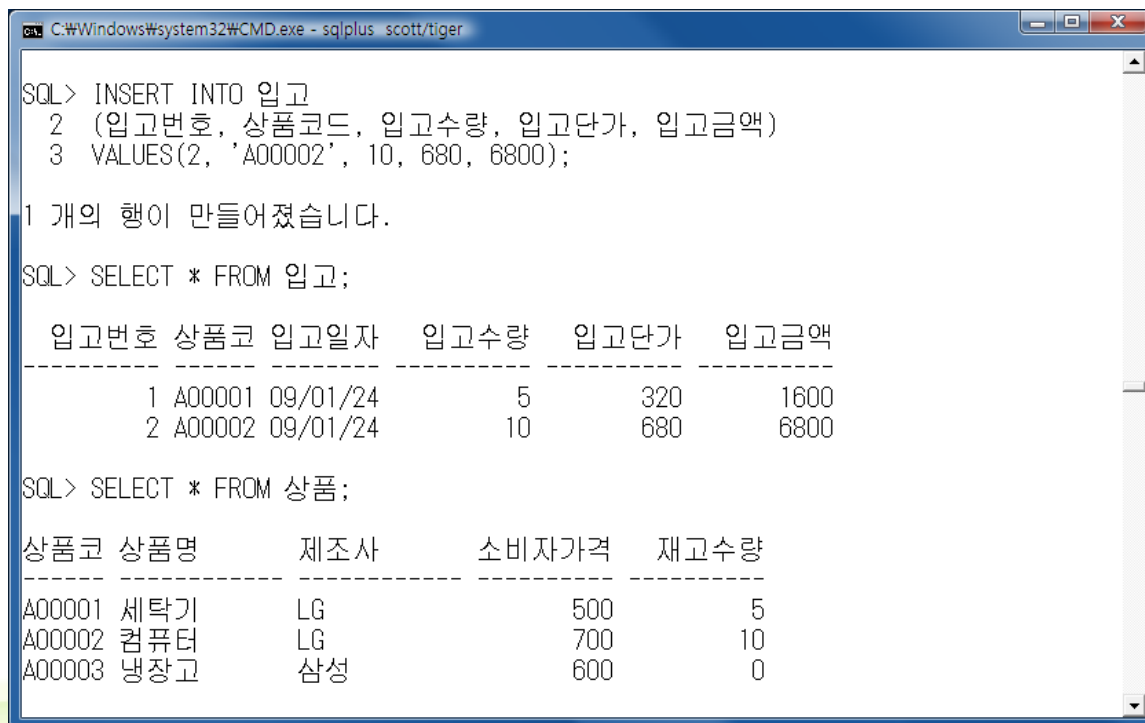
SQL> SELECT * FROM 상품;

상품코  상품명      제조사      소비자가격   재고수량
-----
A00001  세탁기       LG          500           5
A00002  컴퓨터       LG          700           0
A00003  냉장고       삼성        600           0
```

# 실습하기

6. 입고 테이블에 상품이 입력되면 자동으로 상품 테이블의 재고 수량이 증가하게 됩니다. 입고 테이블에 또 다른 상품을 입력합니다.

```
INSERT INTO 입고(입고번호, 상품코드, 입고수량, 입고단가, 입고금액)
VALUES(2, 'A00002', 10, 680, 6800);
SELECT * FROM 입고;
SELECT * FROM 상품;
```



```
C:\Windows\system32\CMD.exe - sqlplus scott/tiger

SQL> INSERT INTO 입고
  2  (입고번호, 상품코드, 입고수량, 입고단가, 입고금액)
  3  VALUES(2, 'A00002', 10, 680, 6800);

1 개의 행이 만들어졌습니다.

SQL> SELECT * FROM 입고;

   입고번호  상품코드  입고일자   입고수량   입고단가   입고금액
-----
          1  A00001  09/01/24         5        320        1600
          2  A00002  09/01/24        10        680        6800

SQL> SELECT * FROM 상품;

상품코드  상품명      제조사      소비자가격  재고수량
-----
A00001   세탁기      LG          500         5
A00002   컴퓨터      LG          700        10
A00003   냉장고      삼성        600         0
```

# 실습하기

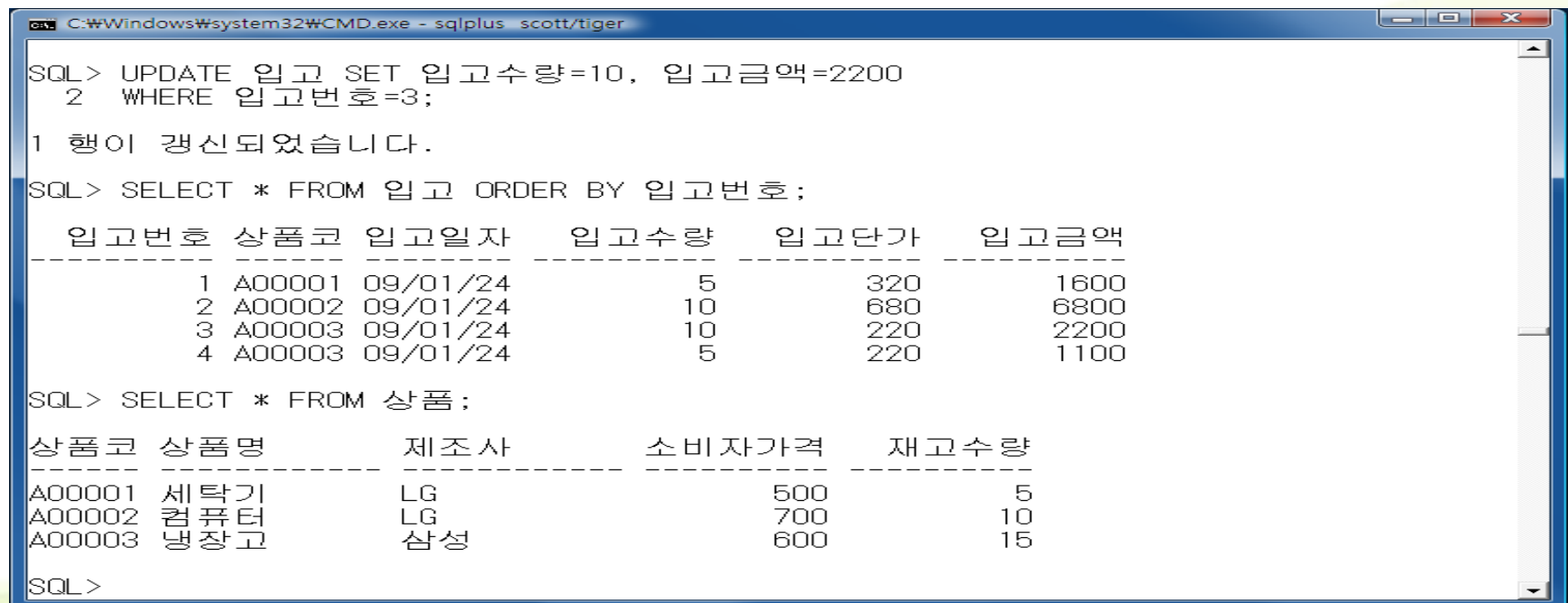
이미 입고된 상품에 대해서 입고 수량이 변경되면 상품 테이블의 재고 수량 역시 변경되어야 합니다. 이를 위한 갱신 트리거 작성해 봅시다.

```
-- 갱신 트리거
CREATE OR REPLACE TRIGGER TRG_04
AFTER UPDATE ON 입고
FOR EACH ROW
BEGIN
UPDATE 상품
SET 재고수량 = 재고수량 + (-:old.입고수량+:new.입고수량)
WHERE 상품코드 = :new.상품코드;
END;
/
```

# 실습하기

2. 입고 번호 3번은 냉장고가 입고된 정보를 기록한 것으로서 입고 번호 3번의 입고수량을 10으로 변경하였더니 냉장고의 재고 수량 역시 15로 변경되었습니다.

```
UPDATE 입고 SET 입고수량=10, 입고금액=2200
WHERE 입고번호=3;
SELECT * FROM 입고 ORDER BY 입고번호;
SELECT * FROM 상품;
```



```
C:\Windows\system32\CMD.exe - sqlplus scott/tiger

SQL> UPDATE 입고 SET 입고수량=10, 입고금액=2200
2  WHERE 입고번호=3;

1 행이 갱신되었습니다.

SQL> SELECT * FROM 입고 ORDER BY 입고번호;

   입고번호  상품코드 입고일자   입고수량   입고단가   입고금액
-----
1          1 A00001  09/01/24         5        320       1600
2          2 A00002  09/01/24        10        680      6800
3          3 A00003  09/01/24        10        220       2200
4          4 A00003  09/01/24         5        220       1100

SQL> SELECT * FROM 상품;

상품코드  상품명      제조사      소비자가격   재고수량
-----
A00001   세탁기      LG          500           5
A00002   컴퓨터      LG          700          10
A00003   냉장고      삼성        600          15

SQL>
```

# 실습하기

입고 테이블에서 입고되었던 상황이 삭제되면 상품 테이블에 재고수량에서 삭제된 입고 수량 만큼을 빼는 삭제 트리거 작성해 봅시다.

--삭제트리거

```
CREATE OR REPLACE TRIGGER TRG_05
AFTER DELETE ON 입고
FOR EACH ROW
BEGIN
UPDATE 상품
SET 재고수량 = 재고수량 - :old.입고수량
WHERE 상품코드 = :old.상품코드;
END;
/
```



# 실습하기

```
DELETE 입고 WHERE 입고번호=1;  
SELECT * FROM 입고 ORDER BY 입고번호;  
SELECT * FROM 상품;
```

C:\Windows\system32\CMD.exe - sqlplus scott/tiger

```
SQL> DELETE 입고 WHERE 입고번호=3;
```

1 행이 삭제되었습니다.

```
SQL> SELECT * FROM 입고 ORDER BY 입고번호;
```

입고번호	상품코	입고일자	입고수량	입고단가	입고금액
1	A00001	09/01/24	5	320	1600
2	A00002	09/01/24	10	680	6800
4	A00003	09/01/24	5	220	1100

```
SQL> SELECT * FROM 상품;
```

상품코	상품명	제조사	소비자가격	재고수량
A00001	세탁기	LG	500	5
A00002	컴퓨터	LG	700	10
A00003	냉장고	삼성	600	5

```
SQL>
```

# 실습하기

EMP 테이블에 INSERT, UPDATE, DELETE 문장이 하루에 몇 건의 ROW가 발생되는지 조사하려고 한다. 조사 내용은 EMP\_AUDIT\_ROW에 사용자 이름, 작업 구분, 작업 시간, 사원번호, 이전의 급여, 갱신된 급여를 저장하는 트리거를 작성하여라.

```
DROP SEQUENCE emp_row_seq;
CREATE SEQUENCE emp_row_seq
    INCREMENT BY 1
    START WITH 1
    MAXVALUE 999999
    MINVALUE 1
    NOCYCLE
    NOCACHE;

DROP TABLE emp_row_tab;
CREATE TABLE emp_row_tab(
    e_id          NUMBER(6)
                CONSTRAINT emp_row_pk PRIMARY KEY,
    e_name        VARCHAR2(30),
    e_gubun       VARCHAR2(10),
    e_date        DATE);
```

# 실습하기

```
CREATE OR REPLACE TRIGGER emp_row_aud
  AFTER insert OR update OR delete ON emp
  FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO emp_row_tab
      VALUES(emp_row_seq.NEXTVAL,USER,'inserting',SYSDATE);
  ELSIF UPDATING THEN
    INSERT INTO emp_row_tab
      VALUES(emp_row_seq.NEXTVAL,USER,'updating',SYSDATE);
  ELSIF DELETING THEN
    INSERT INTO emp_row_tab
      VALUES(emp_row_seq.NEXTVAL,USER,'deleting',SYSDATE);
  END IF;
END;
```

# 실습하기

```
SQL> UPDATE emp  
      SET deptno = 40  
      WHERE deptno = 10;
```

3 rows updated.

```
SQL> SELECT * FROM emp_row_tab;
```

# 실습하기

- 트리거를 이용해서 특정한 조건을 만족하지 않으면 작업을 수행하지 않도록 하기

WHEN 조건

BEGIN

raise\_application\_error(에러코드번호,  
' 메시지');

END;

# 실습하기

- EMP 테이블에서 급여를 수정시 현재의 값보다 적게 수정할 수 없으며 현재의 값보다 10% 이상 높게 수정할 수 없다. 이러한 조건을 만족하는 트리거를 작성하여라.

```
CREATE OR REPLACE TRIGGER emp_sal_chk
BEFORE UPDATE OF sal ON emp
FOR EACH ROW WHEN (NEW.sal < OLD.sal
                    OR NEW.sal > OLD.sal * 1.1)
BEGIN
    raise_application_error(-20502,
        'May not decrease salary. Increase must be < 10%');
END;
/
```

# 실습하기

- EMP 테이블을 사용할 수 있는 시간은 월요일부터 금요일까지 09시부터 18시까지만 사용할 수 있도록 하는 트리거를 작성하여라.

```
CREATE OR REPLACE TRIGGER emp_resource
    BEFORE insert OR update OR delete ON emp
BEGIN
    IF TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')
        OR TO_NUMBER(TO_CHAR(SYSDATE,'HH24'))
            NOT BETWEEN 9 AND 18 THEN
        raise_application_error(-20502,
            '작업할 수 없는 시간 입니다.');
```

END IF;

```
END;
/
```