

# PL/SQL

# 1.PL/SQL

- ❖ Oracle's Procedural Language extension to SQL의 약자
- ❖ SQL은 비절차적이어서 여러 개의 쿼리문을 연결하거나 절차적으로 실행하는데는 사용할 수 없음
- ❖ SQL에 절차적인 프로그래밍 언어를 추가한 것이 PL/SQL
- ❖ SQL 문장에 변수 정의, 조건처리, 반복처리 등을 지원
- ❖ PL/SQL은 오라클이 고유하게 제시하는 것이기 때문에 다른 RDBMS에서는 사용할 수 없음
- ❖ 구조 : Declare – begin – Exception – end
  - ✓ Declare Section : 선언부
  - ✓ Executable Section : 실행부
  - ✓ Exception Section : 예외처리부
- ❖ 작성방법
  - ✓ 하나의 문장이 종료할 때마다 세미콜론을 사용
  - ✓ end 뒤에 세미콜론을 추가
  - ✓ 단일 행 주석은 -- 이고 여러 행 주석은 /\* \*/
  - ✓ 쿼리를 실행하기 위해서는 /가 반드시 필요
  - ✓ 행에 /가 있으면 종료로 간주

# 1.PL/SQL

❖ 아래 처럼 작성하고 실행

```
set serveroutput on
```

```
begin
```

```
    DBMS_OUTPUT.PUT_LINE('Hello PL/SQL');
```

```
end;
```

```
/
```



## 2. 변수 선언과 대입문

### ❖ 변수 선언

변수명 [CONSTANT] 자료형 [NOT NULL]  
[:= | DEFAULT Expression]

=>변수명 : 변수이름

=>CONSTANT : 상수화 하고자 하는 경우 사용

=>자료형 : 변수의 자료형 기재하는데 스칼라 타입 또는 레퍼런스 타입으로 선언

스칼라 타입 : 일반 자료형

레퍼런스 타입 : 테이블의 컬럼이나 ROWTYPE 으로 지정

EMP.EMPNO%TYPE, EMP%ROWTYPE

=>NOT NULL : 값을 반드시 포함

=>Expression : Literal, 다른 변수 또는 연산자 나 함수를 포함하는 식

### ❖ 변수에 값 할당


:= 연산자를 이용



## 2. 변수 선언과 대입문

❖ 변수 선언 과 출력 - DBeaver에서는 마지막 /제외하고 실행

```
DECLARE
  VEMPNO NUMBER(4);
  VENAME VARCHAR2(10);
BEGIN
  VEMPNO := 7999;
  VENAME := 'STEVE';
  DBMS_OUTPUT.PUT_LINE('  직원번호  이름');
  DBMS_OUTPUT.PUT_LINE(' -----');
  DBMS_OUTPUT.PUT_LINE(' ' || VEMPNO || ' ' || VENAME);
END;
/
```



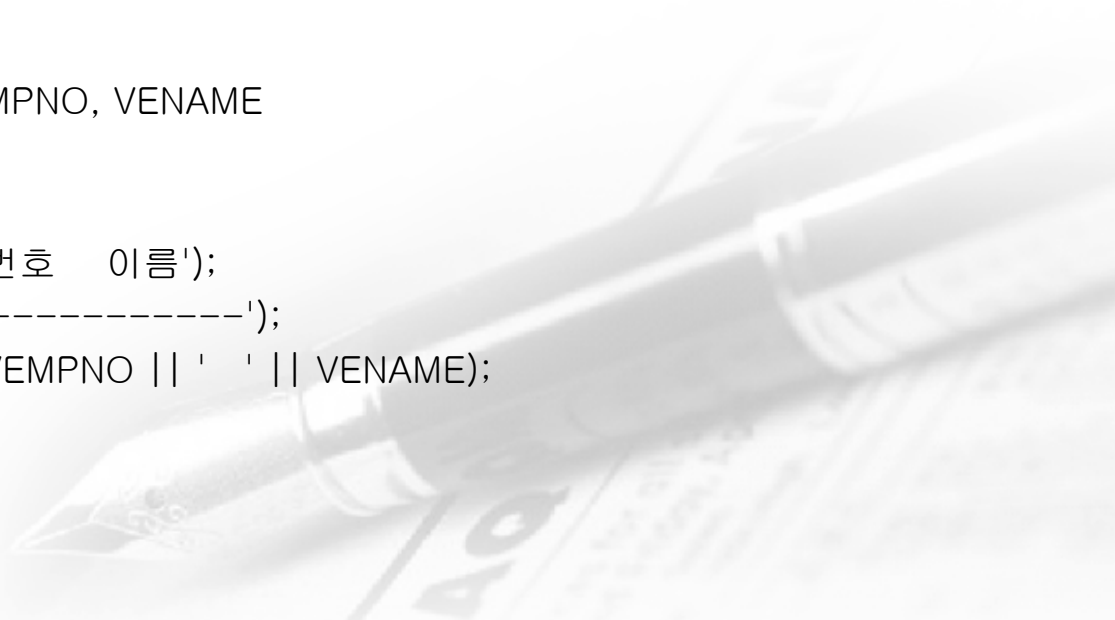
## 2. 변수 선언과 대입문

### ❖ SELECT 구문

SELECT \* 또는 컬럼 이름이나 연산식 나열  
INTO SELECT절의 데이터를 저장할 변수 나열  
FROM 테이블이름  
WHERE 조건

### ❖ EMP 테이블에서 MILLER 인 사원의 데이터 출력

```
DECLARE
  VEMPNO EMP.EMPNO%TYPE;
  VENAME EMP.ENAME%TYPE;
BEGIN
  SELECT EMPNO, ENAME INTO VEMPNO, VENAME
  FROM EMP
  WHERE ENAME='MILLER';
  DBMS_OUTPUT.PUT_LINE('  사원번호   이름');
  DBMS_OUTPUT.PUT_LINE('  -----');
  DBMS_OUTPUT.PUT_LINE('  ' || VEMPNO || '   ' || VENAME);
END;
/
```



# 3.제어문

## ❖ IF 조건 THEN 수행할 문장 END IF

조건이 참을 리턴하는 경우에만 문장을 수행

## ❖ EMP 테이블에서 사원번호가 7902 인 사원의 정보를 출력하고 DEPTNO 에 따라 부서명 출력하기

```
DECLARE
  VEMPNO          EMP.EMPNO%TYPE;
  VENAME          EMP.ENAME%TYPE;
  VDEPTNO  EMP.DEPTNO%TYPE;
  VDNAME          VARCHAR2(20) := NULL;
BEGIN
  SELECT EMPNO, ENAME, DEPTNO
  INTO VEMPNO, VENAME, VDEPTNO
  FROM EMP
  WHERE EMPNO=7902;
/
```

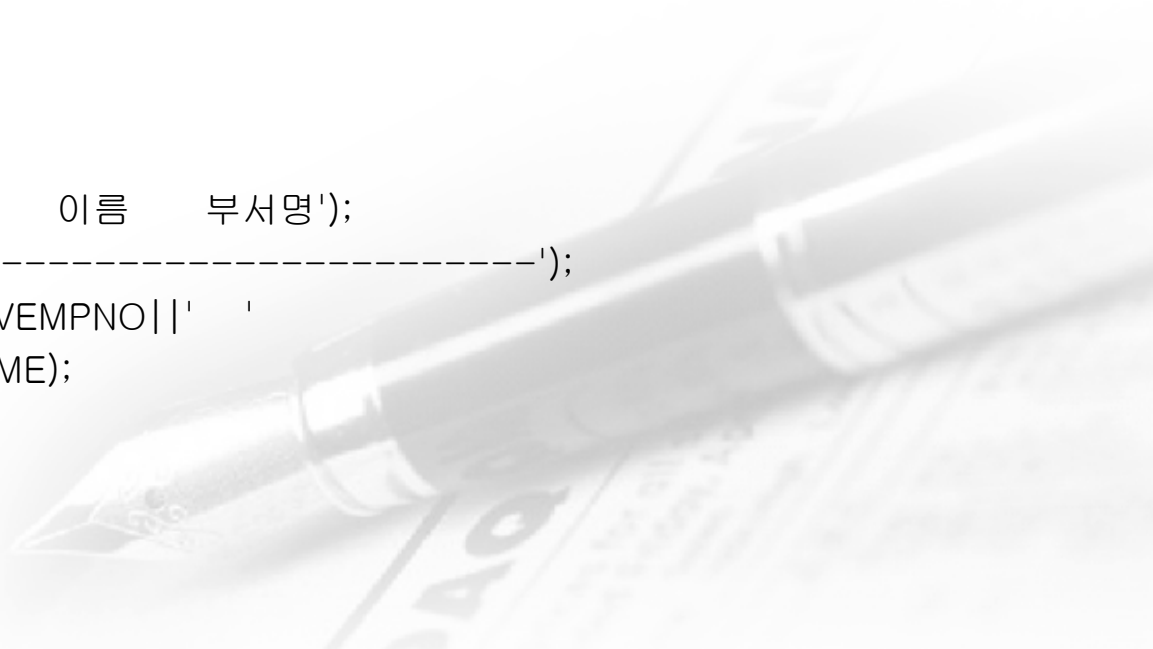


### 3. 제어문

```
IF (VDEPTNO = 10) THEN
    VDNAM := 'ACCOUNTING';
END IF;
IF (VDEPTNO = 20) THEN
    VDNAM := 'RESEARCH';
END IF;
IF (VDEPTNO = 30) THEN
    VDNAM := 'SALES';
END IF;
IF (VDEPTNO = 40) THEN
    VDNAM := 'OPERATIONS';
END IF;

DBMS_OUTPUT.PUT_LINE(' 사번   이름   부서명');
DBMS_OUTPUT.PUT_LINE(' -----');
DBMS_OUTPUT.PUT_LINE(' ' || VEMPNO || ' '
                    || VENAME || ' ' || VDNAM);

END;
/
```





# 3.제어문

❖ IF 조건 THEN 수행할 문장1 ELSE 수행할 문장2 END IF

조건이 참을 리턴하는 경우에는 THEN 뒤에 있는 문장을 수행하고 거짓을 리턴할 때는 ELSE 뒤의 문장을 수행

❖ EMP 테이블에서 ENAME이 MILLER 인 사원의 정보를 출력하고 연봉은 COMM의 값이 NULL이라면  $SAL*12$  를 하고 그렇지 않으면  $SAL*12 + COMM$ 으로 계산

DECLARE

VEMP EMP%ROWTYPE;

ANNSAL NUMBER(7,2);

BEGIN

SELECT \* INTO VEMP

FROM EMP

WHERE ENAME='김사랑';



### 3. 제어문

```
IF (VEMP.COMM IS NULL) THEN
```

```
    ANNSAL:=VEMP.SAL*12;
```

```
ELSE
```

```
    ANNSAL:=VEMP.SAL*12+VEMP.COMM;
```

```
END IF;
```

```
DBMS_OUTPUT.PUT_LINE(' 사번   이름   연봉');
```

```
DBMS_OUTPUT.PUT_LINE('-----');
```

```
DBMS_OUTPUT.PUT_LINE(' '||VEMP.EMPNO||'   '
```

```
    ||VEMP.ENAME||'   '||ANNSAL);
```

```
END;
```

```
/
```



# 3.제어문

❖ IF 조건 THEN 수행할 문장1 ELSE IF 조건 THEN 수행할 문장 2 ELSE 수행할 문장 END IF

첫번째 조건이 참을 리턴하는 경우에는 바로 뒤에 있는 THEN 뒤에 있는 문장을 수행하고 거짓을 리턴할 때는 다음 ELSE IF의 조건을 확인하고 만족하는 조건이 없을 때는 ELSE 뒤의 문장을 수행

❖ 첫번째 if 예제를 다른 방법으로 해결

```
DECLARE
```

```
VEMP EMP%ROWTYPE;
```

```
VDNAME VARCHAR2(14);
```

```
BEGIN
```

```
SELECT * INTO VEMP
```

```
FROM EMP
```

```
WHERE ENAME='MILLER';
```

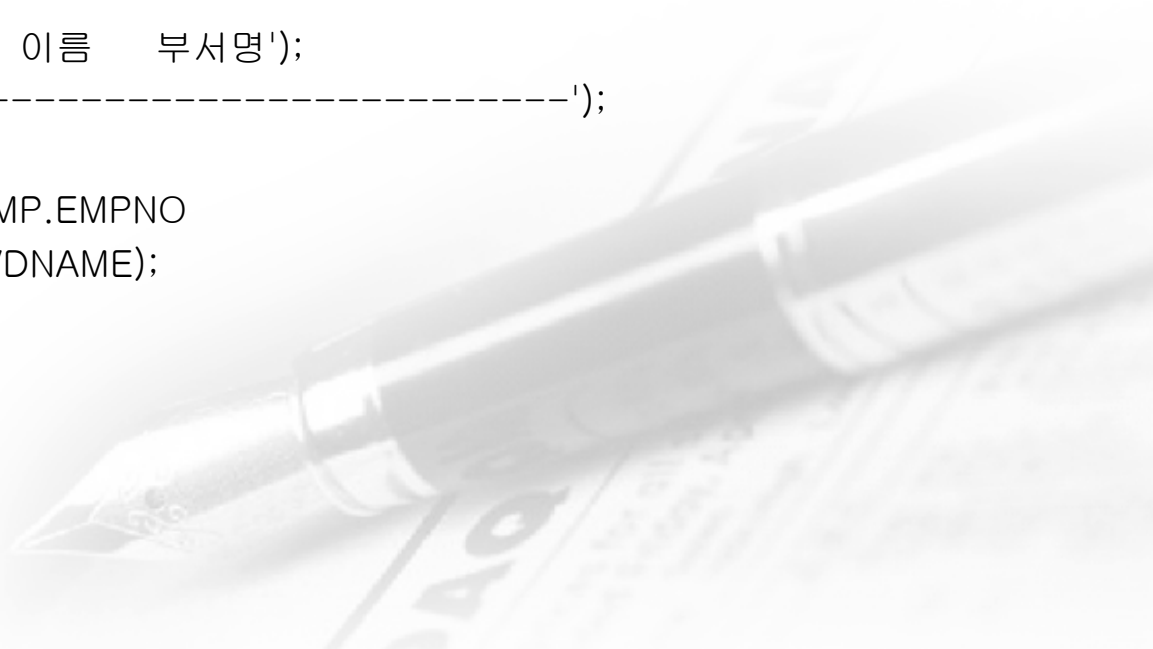


### 3.제어문

```
IF (VEMP.DEPTNO = 10) THEN
    VDNAM := 'ACCOUNTING';
ELSIF (VEMP.DEPTNO = 20) THEN
    VDNAM := 'RESEARCH';
ELSIF (VEMP.DEPTNO = 30) THEN
    VDNAM := 'SALES';
ELSIF (VEMP.DEPTNO = 40) THEN
    VDNAM := 'OPERATIONS';
END IF;

DBMS_OUTPUT.PUT_LINE(' 사번   이름   부서명 ');
DBMS_OUTPUT.PUT_LINE('-----');

DBMS_OUTPUT.PUT_LINE(' ||VEMP.EMPNO
    ||' ||VEMP.ENAME||' ||VDNAME);
END;
/
```



# 3.제어문

## ❖ LOOP EXIT END LOOP

LOOP

반복적으로 수행할 문장

IF 중단조건 THEN

EXIT

END IF;

END LOOP;

## ❖ 1부터 5까지 출력하기

DECLARE

N NUMBER := 1;

BEGIN

LOOP

DBMS\_OUTPUT.PUT\_LINE(N);

N := N + 1;

IF N > 5 THEN

EXIT;

END IF;

END LOOP;

END;

/



# 3. 제어문

## ❖ FOR LOOP

FOR 인덱스변수 in [REVERSE] 시작값..종료값 LOOP  
반복적으로 수행할 문장  
END LOOP;

## ❖ 1부터 5까지 출력하기

```
DECLARE  
BEGIN  
  FOR N IN 1..5 LOOP  
    DBMS_OUTPUT.PUT_LINE( N );  
  END LOOP;  
END;  
/
```



# 3.제어문

## ❖ WHILE LOOP

WHILE 조건 LOOP

반복적으로 수행할 문장

END LOOP;

## ❖ 1부터 5까지 출력하기

DECLARE

N NUMBER := 1;

BEGIN

WHILE N <= 5 LOOP

DBMS\_OUTPUT.PUT\_LINE( N );

N := N + 1;

END LOOP;

END;

/



# 프로시저와 트리거



# 1. 저장 프로시저

- ❖ 오라클은 사용자가 만든 PL/SQL 문을 데이터베이스에 저장 할 수 있도록 저장프로시저를 제공
- ❖ 저장 프로시저를 사용하면 DML 문을 필요할 때마다 다시 입력할 필요없이 간단하게 호출만 해서 DML 문을 실행
- ❖ 저장 프로시저를 사용하면 성능도 향상되고 프로그래밍 언어와의 호환성 문제도 해결



# 1. 저장 프로시저

- ❖ 저장 프로시저를 생성하기 위한 CREATE PROCEDURE의 형식  
CREATE [OR REPLACE ] PROCEDURE *prcedure\_name*  
  ( *argument1* [mode] *data\_taye*,  
    *argument2* [mode] *data\_taye* . . .  
  )  
IS  
  *local\_variable declaration*  
BEGIN  
  *statement1*;  
  *statement2*;  
  . . .  
END;  
/



# 1. 저장 프로시저

- ❖ 저장 프로시저를 생성하려면 CREATE PROCEDURE 다음에 생성하고자 하는 프로시저 이름을 기술
- ❖ 생성한 저장 프로시저는 여러 번 반복해서 호출해서 사용할 수 있음
- ❖ 생성된 저장 프로시저를 제거하기 위해서는 DROP PROCEDURE 다음에 제거하고자 하는 프로시저 이름을 기술
- ❖ OR REPLACE 옵션은 이미 존재하는 이름으로 저장 프로시저를 생성할 경우 기존 프로시저는 삭제하고 새롭게 기술한 내용으로 재 생성하도록 하는 옵션
- ❖ 프로시저는 어떤 값을 전달받아서 그 값에 의해서 서로 다른 결과물을 구하도록 작성할 수 있음
- ❖ 값을 프로시저에 전달하기 위해서 프로시저 이름 다음에 괄호로 둘러 쓴 부분에 전달 받을 값을 저장할 변수를 기술
- ❖ [MODE] 는 IN과 OUT, INOUT 세 가지를 기술할 수 있는데 IN은 데이터를 전달 받을 때 쓰고 OUT은 수행된 결과를 받아갈 때 사용하며 INOUT은 두 가지 목적에 모두 사용



# 실습

1. EMP 테이블과 동일한 구조와 데이터를 갖는 EMP01 테이블을 생성

```
create table emp01  
as  
select * from emp;
```

2. 프로시저 생성

```
CREATE OR REPLACE PROCEDURE DEL_ALL  
IS  
BEGIN  
DELETE FROM EMP01;  
END;  
/
```



# 실습

3. 생성된 저장 프로시저는 EXECUTE 명령어로 실행  
EXECUTE DEL\_ALL; - sqlplus에서 실행

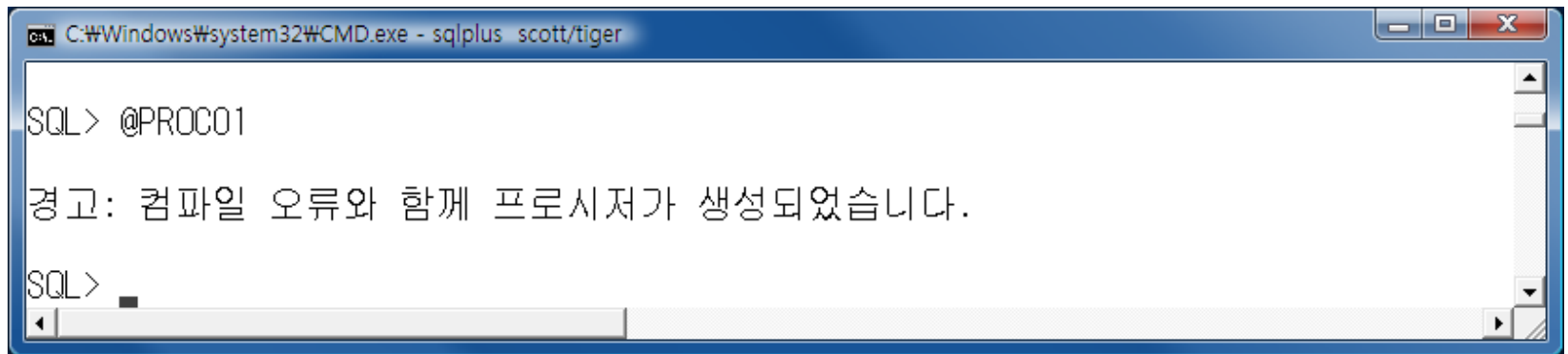
```
BEGIN  
  DEL_ALL();  
END;
```

- Dbeaver 나 sqldeveloper에서 실행



# 1.1 오류 원인 살피기

- ❖ ‘프로시저가 생성되었습니다.’란 메시지와 함께 한 번에 저장 프로시저를 성공적으로 생성할 수도 있지만, ‘경고: 컴파일 오류와 함께 프로시저가 생성되었습니다’와 같은 메시지가 출력되면 저장 프로시저를 생성하는 과정에서 오류가 발생해서 저장 프로시저 생성에 실패한 경우



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

SQL> @PROC01

경고: 컴파일 오류와 함께 프로시저가 생성되었습니다.

SQL>
```

## 1.2 저장 프로시저 조회하기

- ❖ 저장 프로시저를 작성한 후 사용자가 저장 프로시저가 생성되었는지 확인하려면 USER\_SOURCE를 확인
- ❖ USER\_SOURCE의 구조 확인
  - ✓ DESC USER\_SOURCE
- ❖ USER\_SOURCE의 내용을 조회하면 어떤 저장 프로시저가 생성되어 있는지와 해당 프로시저의 내용이 무엇인지 확인할 수 있음



## 1.3 매개 변수

- ❖ DEL\_ALL 저장 프로시저는 사원 테이블의 모든 내용을 삭제
- ❖ 만일 특정 사원 만을 삭제하는 프로시저를 만들고자 하면 삭제하고자 하는 사원의 이름이나 사원 번호를 프로시저에 전달해 주어 이와 일치하는 사원을 삭제
- ❖ 저장 프로시저에 값을 전달해 주기 위해서 매개 변수를 사용





# 1.3 매개 변수

- ❖ 매개변수가 있는 저장프로시저는 다음과 같이 정의

```
CREATE OR REPLACE PROCEDURE  
DEL_ENAME(VENAME EMP01.ENAME%TYPE)  
IS  
BEGIN  
DELETE FROM EMP01 WHERE ENAME=VENAME;  
END;  
/
```

- ❖ 저장 프로시저 이름인 DEL\_ENAME 다음에 ( )를 추가하여 그 안에 선언한 변수가 매개 변수이며  
값은 프로시저를 호출할 때 전달

```
BEGIN  
    DEL_ENAME('SMITH');  
END;
```



# 1.4 IN, OUT, INOUT

- ❖ CREATE PROCEDURE로 프로시저를 생성할 때 MODE를 지정하여 매개변수를 선언할 수 있는데 MODE 에 IN, OUT, INOUT 세 가지를 기술
- ❖ IN은 데이터를 전달 받을 때 쓰고 OUT은 수행된 결과를 받아갈 때 사용
- ❖ INOUT은 두 가지 목적에 모두 사용



## 1.5 IN 매개 변수

- ❖ DEL\_ENAME 프로시저에서 사용된 매개변수는 프로시저를 호출할 때 기술한 값을 프로시저 내부에서 받아서 사용

```
EXECUTE DEL_ENAME('SMITH');
```

```
CREATE PROCEDURE DEL_ENAME(VENAME EMP01.ENAME%TYPE)
```


- ❖ 프로시저 호출시 넘겨준 값을 받아오기 위한 매개변수는 MODE를 IN으로 지정해서 선언

```
CREATE PROCEDURE DEL_ENAME(VENAME IN EMP01.ENAME%TYPE)
```

# 1.6 OUT 매개 변수

❖ 프로시저에서 구한 결과 값을 저장하기 위해서 MODE를 OUT으로 지정

```
CREATE OR REPLACE PROCEDURE SEL_EMPNO
( VEMPNO IN      EMP.EMPNO%TYPE,
  VENAME OUT EMP.ENAME%TYPE,
  VSAL OUT EMP.SAL%TYPE,
  VJOB OUT EMP.JOB%TYPE
)
IS
BEGIN
    SELECT ENAME, SAL, JOB INTO VENAME, VSAL, VJOB
    FROM EMP
    WHERE EMPNO=VEMPNO;
END;
/
```



# 1.6 OUT 매개 변수

```
VARIABLE VAR_ENAME VARCHAR2(15);  
VARIABLE VAR_SAL NUMBER;  
VARIABLE VAR_JOB VARCHAR2(9);  
  
BEGIN  
    SEL_EMPNO(7902, :VAR_ENAME, :VAR_SAL, :VAR_JOB);  
END;  
  
PRINT VAR_ENAME  
PRINT VAR_SAL  
PRINT VAR_JOB
```



## 1.7 프로시저의 장점

- ❖ 일반 SQL 문을 수행하는 경우는 클라이언트 응용 프로그램이 SQL을 직접 내장하고 있거나 쿼리 분석기와 같이 사용자가 직접 쿼리를 입력해야 하기 때문에 동일한 SQL 문장을 여러 번 수행하는 경우 계속 동일한 SQL 문장을 전송
- ❖ 저장 프로시저를 이용하면 서버가 프로시저의 내용을 가지고 있는 상태에서 클라이언트 응용 프로그램을 매개변수와 함께 호출하면 프로시저의 정의를 읽어서 실행하기 때문에 트래픽이 줄어듦



# 연습문제

1. EMP TABLE에 사번(EMPNO), 이름(ENAME), 급여(SAL), 부서번호(DEPTNO)를 전달받아 등록하는 PROCEDURE를 작성
2. EMP TABLE에 사원번호(empno)와 급여(sal)를 입력받아서 empno에 해당하는 데이터의 sal의 값을 입력받은 값으로 수정하는 PROCEDURE를 작성



## 2. 트리거

- ❖ 특정 테이블에 DML(INSERT,UPDATE,DELETE)문장이 수행되었을 때 데이터베이스에서 자동적으로 PL/SQL 블록을 수행 시키기 위해서 데이터베이스 TRIGGER를 사용
- ❖ TRIGGER는 트리거링 이벤트가 일어날 때마다 암시적으로 실행
- ❖ 트리거링 이벤트에는 데이터베이스 테이블에서 INSERT,UPDATE,DELETE 오퍼레이션
- ❖ 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 못함
  - ✓ TRIGGER 주요부에 의해 호출되는 프로시저나 함수는 트랜잭션 제어 문장을 사용하지 못함
  - ✓ TRIGGER 주요부는 LONG또는 LONG RAW변수를 선언할 수 없음
  - ✓ TRIGGER 주요부가 액세스하게 될 테이블에 대한 제한이 있음



## 2. 트리거

### ❖ 트리거를 만들기 위한 CREATE TRIGGER 문의 형식

```
CREATE TRIGGER trigger_name  
timing[BEFORE|AFTER] event[INSERT|UPDATE|DELETE]  
ON table_name  
[FOR EACH ROW]  
[WHEN conditions]  
BEGIN  
statement  
END
```

- ✓ *trigger\_name*: TRIGGER의 식별자
- ✓ BEFORE | AFTER: DML문장이 실행되기 전에 TRIGGER를 실행할 것인지 실행된 후에 TRIGGER를 실행할 것인지를 정의
- ✓ *triggering\_event*: TRIGGER를 실행하는 DML(INSERT,UPDATE,DELETE)문을 기술
- ✓ OF *column*: TRIGGER가 실행되는 테이블에서 COLUMN명을 기술
- ✓ *table\_name*: TRIGGER가 실행되는 테이블 이름
- ✓ FOR EACH ROW: 이 옵션을 사용하면 행 레벨 트리거가 되어 *triggering*문장에 의해 영향받은 행에 대해 각각 한번씩 실행하고 사용하지 않으면 문장 레벨 트리거가 되어 DML문장 당 한번만 실행된

## 2. 트리거

### ❖ 트리거의 타이밍

- ✓ [BEFORE] 타이밍은 어떤 테이블에 INSERT, UPDATE, DELETE 문이 실행될 때 해당 문장이 실행되기 전에 트리거가 가지고 있는 BEGIN ~ END 사이의 문장을 실행
- ✓ [AFTER] 타이밍은 INSERT, UPDATE, DELETE 문이 실행되고 난 후에 트리거가 가지고 있는 BEGIN ~ END 사이의 문장을 실행

### ❖ 트리거의 이벤트

- ✓ 사용자가 어떤 DML(INSERT, UPDATE, DELETE)문을 실행했을 때 트리거를 발생시킬 것인지를 결정

### ❖ 트리거의 몸체

- ✓ 해당 타이밍에 해당 이벤트가 발생하게 되면 실행될 기본 로직이 포함되는 부분으로 BEGIN ~ END에 기술
- ✓ IF INSERTING THEN 문장을 이용해서 삽입, 갱신(UPDATING), 삭제(DELETING) 시 다른 작업 수행 가능



## 2. 트리거

### ❖ 트리거의 유형

- ✓ 트리거의 유형은 FOR EACH ROW에 의해 문장 레벨 트리거와 행 레벨 트리거로 나눔
- ✓ FOR EACH ROW가 생략되면 문장 레벨 트리거이고 행 레벨 트리거를 정의하고자 할 때에는 반드시 FOR EACH ROW를 기술
- ✓ 문장 레벨 트리거는 어떤 사용자가 트리거가 설정되어 있는 테이블에 대해 DML(INSERT, UPDATE, DELETE)문을 실행할 때 단 한번만 트리거를 발생시킬 때 사용
- ✓ 행 레벨 트리거는 DML(INSERT, UPDATE, DELETE)문에 의해서 여러 개의 행이 변경된다면 각 행이 변경될 때마다 트리거를 발생시키는 방법입니다. 만약 5개의 행이 변경되면 5번 트리거가 발생

### ❖ TRIGGER에서 OLD와 NEW

- ✓ 행 레벨 TRIGGER에서만 사용할 수 있는 예약어로 트리거 내에서 현재 처리되고 있는 행을 액세스할 수 있는데 두개의 의사 레코드를 통하여 이 작업을 수행할 수 있는데 :OLD는 INSERT문에 의해 정의되지 않고 :NEW는 DELETE에 대해 정의되지 않지만 UPDATE는 :OLD와 :NEW를 모두 정의
- ✓ INSERT        모든 필드는 NULL로 정의 - 문장이 완전할 때 삽입된 새로운 값
- ✓ UPDATE       갱신하기 전의 원래 값 - 문장이 완전할 때 갱신된 새로운 값
- ✓ DELETE       행이 삭제되기 전의 원래 값 - 모든 필드는 NULL이다.

# 실습

- ❖ 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 작 (신입사원의 급여는 일괄적으로 100)

- ❖ EMP01 테이블을 생성

```
drop table emp01;
```

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

- ❖ 급여를 저장할 테이블을 생성

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

# 실습

- ❖ 급여번호를 자동 생성하는 시퀀스를 정의  
CREATE SEQUENCE SAL01\_SALNO\_SEQ;
- ❖ 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;  
/



# 실습

- ❖ 사원 테이블에 데이터를 추가

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

```
SELECT * FROM EMP01;
```

```
SELECT * FROM SAL01;
```



# 실습

- ❖ 사원이 삭제되면 그 사원의 급여 정보도 자동 삭제되는 트리거를 작성

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

- ❖ 데이터 삭제 후 확인

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

```
SELECT * FROM EMP01;
```

```
SELECT * FROM SAL01;
```



# 3. 트리거 삭제

❖ DROP TIGGER 다음에 삭제할 트리거 명을 기술  
DROP TRIGGER TRG\_01;

DROP TRIGGER TRG\_02;





# 실습

- ❖ 트리거를 이용해서 특정한 조건을 만족하지 않으면 작업을 수행하지 않도록 하기

WHEN 조건

BEGIN

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

END;

- ❖ EMP 테이블에서 급여를 수정시 현재의 값보다 적게 수정할 수 없으며 현재의 값보다 10% 이상 높게 수정할 수 없도록 하는 트리거를 작성

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;


/



# 실습 갱신 트리거

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

```
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;
/
```

# 연습문제

1. TRIGGER란 ?

2. EMP\_SAL\_TOT(부서번호,급여의 합) TABLE을 생성

```
CREATE TABLE EMP_SAL_TOT as  
  SELECT deptno,SUM(sal) sal_tot  
  FROM emp  
  GROUP BY deptno;
```

3. EMP\_SAL\_TOT TABLE의 내용은 EMP TABLE을 변경 작업(INSERT, UPDATE, DELETE)이 발생하면 내용도 자동적으로 변경되도록 해주는 TRIGGER를 작성

