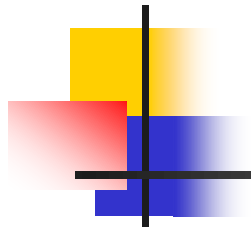


데이터베이스 프로그래밍

(소프트웨어 개발 트랙)



제 2부

저장 프로시저 프로그래밍(1)



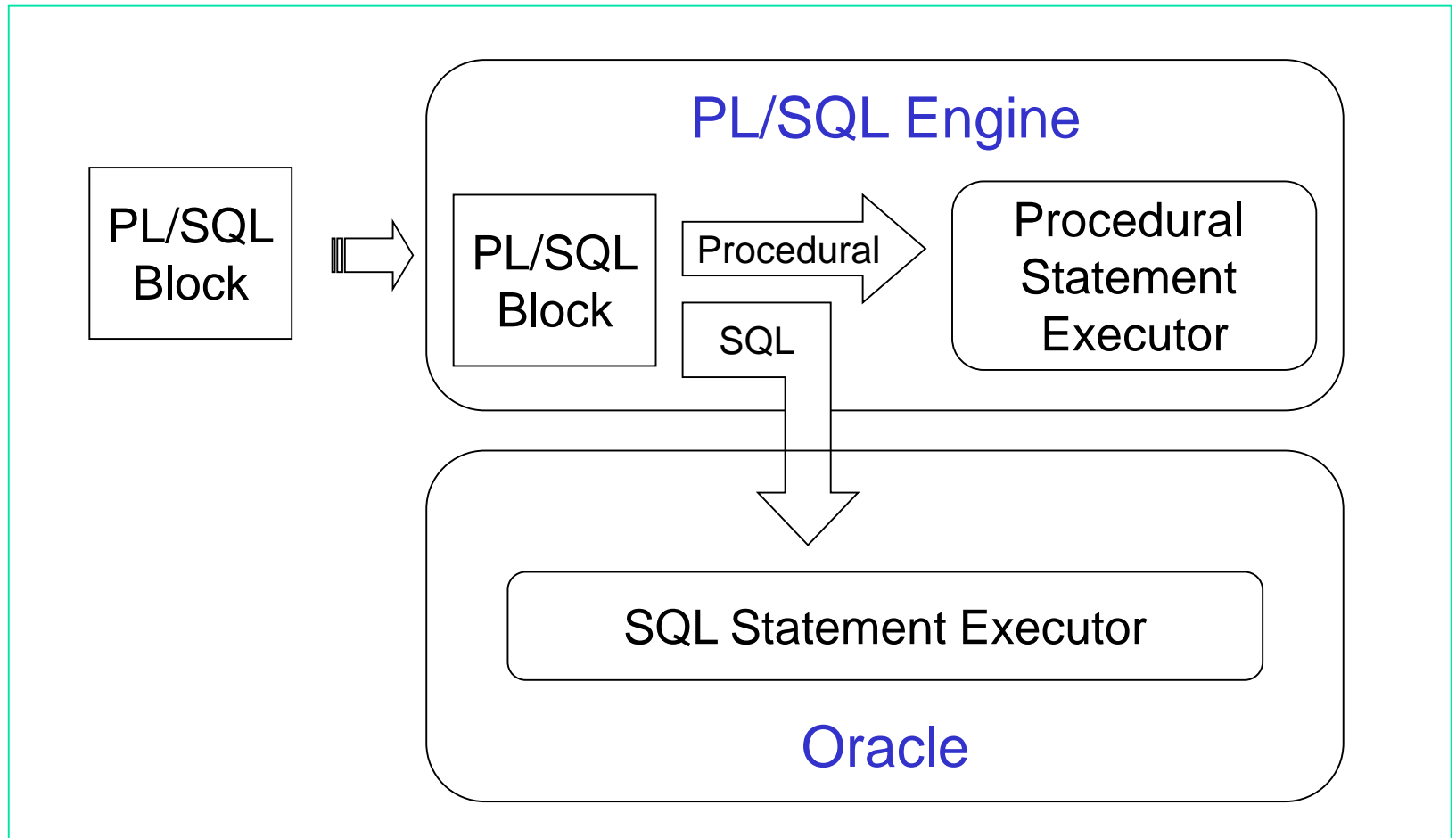
PL/SQL 소개



PL/SQL 개요

- Oracle's **P**rocedural **L**anguage extension to **SQL**
- 오라클에서 제공하는 **SQL**을 확장한 절차적 프로그래밍 언어
- 특징
 - **SQL** 문장에서 변수 정의, 조건 처리, 반복 처리 등 프로그래밍 언어가 보이는 제어 구조 제공
 - **SQL**과 프로그래밍 언어의 통합 접근 중 한 형태
 - **ESQL**, **SQLJ** 등의 기술과 달리 상용 프로그래밍 언어가 아닌 오라클 자체적인 프로그래밍 언어 사용

엔진 실행 구조





언어 구조

- 블록단위 구조

- 임의의 수의 중첩된 하위 블록 포함
- 선언부(DECLARE), 실행부(BEGIN... END), 예외 처리부(EXCEPTION)로 구성

DECLARE -- 선택

-- 변수, 상수, 커서, 사용자 지정 예외 선언

BEGIN -- 필수

-- SQL 문장

-- PL/SQL 제어 문장

EXCEPTION -- 선택

-- 에러 발생시 수행될 액션

END; -- 필수

- 익명(Anonymous)
 - 이름이 없는 PL/SQL 블록
- 저장 프로시저(Stored Procedure) 및 함수(Function)
 - 매개 변수를 받을 수 있고, 반복해서 사용할 수 있는 이름이 있는 PL/SQL 블록
- 패키지(Package)
 - 관련된 저장 프로시저, 함수를 모은 이름이 있는 PL/SQL 블록
- 트리거(Trieger)
 - 데이터베이스의 테이블과 연결되어 자동적으로 실행되는 이름이 있는 PL/SQL 블록



식별자(Identifiers)

- 첫 자리는 알파벳으로 시작 (단, 스페이스는 제외)
- 총 자릿수는 30자 이내
- 특수문자 사용 가능(단, &,-,/,space 제외)
- 대소문자 구별은 하지 않음
- 참고
 - 예약어, 식별자 사용 불가 문자 사용 방법
 - 식별자 이름에 따옴표 사용
 - 대소문자 구분
 - 예 : “Space Included”

연산자

연산자	설명
+, -, *, /, **	덧셈, 뺄셈, 곱셈, 나눗셈, 지수 연산자
=, <, >, <>, !=, <=, >=	관계 연산자
(,)	설명 또는 리스트 구분자
;	문장 끝마침 구분자
%	속성 인자
,	아이템 또는 문자열 구분자
:=	지정(assignment) 연산자
..	범위(range) 연산자
	문자열 연결자
--	주석 연산자(한 라인 이하)
/* */	주석 연산자(복수 라인)
<<. >>	레벨 구분자



리터럴

- 숫자

- 정수 및 실수의 숫자 값, 예:123, -4.6
- 지수로 표현된 숫자 값, 예:1.43E5

- 문자

- 작은 따옴표로 구분된 문자열, 예:'STRING'
- NULL 값 표현 가능, 예:"

- 불리언

- TRUE, FALSE, NULL



주석(Comment)

- `/* ~ */` : 한 줄 이상의 줄 주석
- `--` : 한 줄 단위의 주석
- 예)

```
DECLARE
    sMsg    VARCHAR2(10) := '테스트';
BEGIN
    -- sMsg := '주석 1';
    /*
        sMsg := '주석 2';
        DBMS_OUTPUT.PUT_LINE(sMsg);
    */
    DBMS_OUTPUT.PUT_LINE(sMsg);
END;
```



PL/SQL 사용

- SQL*PLUS에서 SQL언어를 작성하는 것과 유사하게 작성
 - **SQL>** 프롬프트상에서 사용
- 익명 블록 형태가 아닌 저장 프로시저, 함수, 트리거 등의 형태로 사용



PL/SQL 결과 출력

- PL/SQL의 결과를 확인하기 위해 출력하는 방법
 - SET SERVEROUTPUT ON;
 - DBMS_OUTPUT.PUT_LINE(...);
- 출력 방법 예

```
SQL> SET SERVEROUTPUT ON;  
SQL>  
BEGIN  
    DBMS_OUTPUT.PUT_LINE('HELLO');  
END;  
/  
HELLO
```



PL/SQL 데이터 타입

- SQL 제공 모든 데이터 타입 + 추가 데이터 타입
 - PL/SQL 추가 데이터 타입
 - BOOLEAN
 - BINARY_INTEGER, NATURAL, POSITIVE
 - %TYPE
 - %ROWTYPE
 - PL/SQL 테이블과 레코드
- 데이터 타입의 유형
 - 스칼라(Scalar) 데이터 타입
 - 복합(Composite) 데이터 타입
 - 참조 데이터 타입



변수 선언

■ 형식

식별자 [CONSTANT] 데이터타입 [NOT NULL]
[:=상수 값이나 표현식];

- 식별자: 변수나 상수의 이름
- **CONSTANT** : 식별자가 그 값이 변할 수 없도록 선언, 반드시 초기화
- 데이터 타입 : 스칼라 또는 복합 데이터 타입 선언 시 사용
- **NOT NULL** : **NOT NULL**로 제한된 변수는 반드시 초기화
- 초기값을 정의하지 않으면 식별자는 **NULL** 값을 가짐

스칼라 데이터 타입(1)

- 단순 데이터 형으로 하나의 데이터 값을 저장하는 데이터 타입

데이터 타입	설명
BINARY_INTEGER	-2147483647과 2147483647사이의 정수, 디폴트 값=1
NUMBER[(p,s)]	정수와 실수
CHAR[(최대길이)]	32767 바이트까지의 고정 길이 문자 최대길이 미지정시 디폴트 길이는 1
LONG	32767 바이트까지의 가변 길이 문자 데이터
VARCHAR2(최대길이)	32767 바이트까지의 가변 길이 문자 데이터
DATE	날짜와 시간 데이터
BOOLEAN	TRUE, FALSE, NULL 중 한 가지 값 저장



스칼라 데이터 타입(2)

■ 예

- v_gender CHAR(1);
- v_count BINARY_INTEGER := 0;
- v_total_sal NUMBER(9,2) := 0;
- v_order_date DATE := SYSDATE + 7;
- c_tax_rate CONSTANT NUMBER(3,2) := 8.25;
- v_valid BOOLEAN NOT NULL := TRUE;



스칼라 데이터 타입(3) : %TYPE

- %TYPE 데이터 타입
 - 컬럼 데이터 타입을 모를 경우 사용
 - 컬럼의 데이터 타입이 변경될 경우 다시 수정할 필요가 없음
 - 이미 선언된 다른 변수나 데이터베이스 컬럼의 데이터 타입을 이용하여 선언
 - %TYPE앞에 올 수 있는 것은 데이터베이스 테이블과 컬럼 그리고 이미 선언한 변수명



스칼라 데이터 타입(4) : %TYPE

- %TYPE 데이터 타입

- 예

- v_empno emp.empno%TYPE := 7900 ;
 - v_empno의 데이터 타입 : emp 테이블의 empno컬럼의 데이터 타입
 - v_ename emp.ename%TYPE;
 - v_ename의 데이터 타입 : emp 테이블의 ename컬럼의 데이터 타입



스칼라 데이터 타입(5) : %TYPE

- %TYPE 데이터 타입 예

```
DECLARE
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
  v_sal emp.sal%TYPE;
BEGIN
  SELECT empno, ename, sal
  INTO    v_empno, v_ename, v_sal
  FROM    emp
  WHERE empno = 7521;

  DBMS_OUTPUT.PUT_LINE( '사원번호 : ' || v_empno );
  DBMS_OUTPUT.PUT_LINE( '사원이름 : ' || v_ename );
  DBMS_OUTPUT.PUT_LINE( '사원급여 : ' || v_sal );
END;
```



복합 데이터 타입(1)

■ 개념

- 하나 이상의 데이터 값을 가지는 데이터 타입
- 배열과 유사한 역할
- PL/SQL 테이블과 레코드, %ROWTYPE

■ %ROWTYPE

■ 특징

- 테이블이나 뷰 내부의 컬럼 집합의 이름, 데이터 타입, 크기, 속성을 그대로 사용하여 선언
- %ROWTYPE 앞에 오는 것은 테이블명

■ 장점

- 컬럼들의 수나 데이터 타입을 모를 때 편리
- 해당 컬럼들의 수나 데이터 타입이 변경될 경우 수정하지 않아도 됨
- SELECT문을 이용하여 하나의 행을 조회할 때 편리



복합 데이터 타입(2)

- %ROWTYPE(Cont'd)

- 예

```
DECLARE
    v_emp emp%ROWTYPE;

BEGIN
    SELECT empno, ename
    INTO    v_emp.empno, v_emp.ename
    FROM    emp
    WHERE   empno = 1;

    DBMS_OUTPUT.PUT_LINE('번호 : ' || v_emp.empno );
    DBMS_OUTPUT.PUT_LINE('이름 : ' || v_emp.ename );
END;
```



복합 데이터 타입(3)

■ PL/SQL 테이블

■ 특징

- 일차원 배열과 유사
- **BINARY_INTEGER** 데이터 타입의 기본 키와 데이터를 저장하는 스칼라 데이터 타입 컬럼의 구성
- 테이블 변수 선언 시 초기값 부여 불가능
- 한 개의 컬럼 데이터를 저장
- 테이블의 크기는 제한이 없으며 그 **ROW**의 수는 데이터가 들어옴에 따라 자동 증가

■ 형식

```
TYPE type_name IS TABLE OF datatype [NOT NULL]
    INDEX BY BINARY_INTEGER;
Identifier type_name;
```




복합 데이터 타입(4)

- PL/SQL 테이블(Cont'd)

- 예

```
DECLARE  
  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;  
  
  empno_tab empno_table;  
  ename_tab ename_table;  
  
  i BINARY_INTEGER := 0;
```



복합 데이터 타입(5)

```
BEGIN
  FOR emp_list IN(SELECT empno, ename
                  FROM emp WHERE deptno = 20)
  LOOP
    i := i + 1;

    empno_tab(i) := emp_list.empno;
    ename_tab(i) := emp_list.ename;

    DBMS_OUTPUT.PUT_LINE
      ('번호 : ' || empno_tab(i));
    DBMS_OUTPUT.PUT_LINE
      ('이름 : ' || ename_tab(i));
  END LOOP;
END;
```



복합 데이터 타입(6)

- PL/SQL 테이블(Cont'd)

- 속성

- COUNT

- PL/SQL 테이블의 전체 행 수 리턴
 - 예 : `ename_tab`에 저장된 전체 데이터 행의 수
 - `i := ename_tab.COUNT;`

- DELETE

- PL/SQL 테이블의 특정 행 삭제
 - 예1 : `ename_tab`의 인덱스가 3인 행 삭제
 - `ename_tab.DELETE(3);`
 - 예 2: `ename_tab` 전체 행 삭제
 - `ename_tab.DELETE;`
 - 예 3 : `ename_tab`의 2~3행 삭제
 - `ename_tab.DELETE(2,3);`



복합 데이터 타입(7)

- PL/SQL 테이블(Cont'd)

- 속성

- EXISTS

- PL/SQL 테이블의 특정 행이 존재하면 **TRUE**, 존재하지 않으면 **FALSE** 리턴
 - 예 : `ename_tab`에 1행이 존재하는지 검사

```
IF ename_tab.EXISTS(1) THEN
    DBMS_OUTPUT.PUT_LINE('ename_tab(1) exists');
ELSE
    DBMS_OUTPUT.PUT_LINE('ename_tab(1) does not exist');
END IF;
```



복합 데이터 타입(8)

- PL/SQL 테이블(Cont'd)

- 속성

- FIRST, LAST

- PL/SQL 테이블의 첫 번째(마지막) 행의 인덱스 번호 리턴
 - 예

```
ename_tab(70) := '이름1';  
ename_tab(75) := '이름2';  
ename_tab(73) := '이름3';  
--- 인덱스 번호 70을 v_index_first에 지정  
v_index_first := ename_tab.FIRST;  
--- 인덱스 번호 75를 v_index_last에 지정  
v_index_last := ename_tab.LAST;
```



복합 데이터 타입(9)

- PL/SQL 테이블(Cont'd)

- 속성

- NEXT, PRIOR

- PL/SQL 테이블의 다음(이전) 행의 인덱스번호 리턴

- 예 :

```
--- 첫번째 인덱스 지정
v_index := ename_tab.FIRST;
LOOP
  --- 지정된 인덱스 값에 해당하는 ename 입력
  INSERT INTO emp(ename) values (ename_tab(v_index));
  --- 마지막 인덱스이면 LOOP 탈출
  EXIT WHEN v_index = ename_tab.LAST;
  --- 현재 인덱스 번호의 다음 인덱스 번호 지정
  v_index := ename_tab.NEXT(v_index);
END LOOP;
```



복합 데이터 타입(10)

- PL/SQL 레코드

- 특징

- 사용자 정의 레코드
 - 여러 개의 데이터 타입을 가지는 변수들의 집합
 - 데이터 처리를 위해 테이블에서 하나의 행씩 **FETCH**할 때 편리
 - 개별 필드 이름 부여가 가능하고, 타입 선언 시 초기화 가능
 - 스칼라, 레코드, **PL/SQL** 테이블 데이터 타입 중 하나 이상의 요소로 구성



복합 데이터 타입(11)

- PL/SQL 레코드 (Cont'd)
 - 형식

```
TYPE type_name IS RECORD
```

```
    (필드이름1 필드타입 [NOT NULL {:=|DEFAULT} 식],  
      필드이름2 필드타입 [NOT NULL {:=|DEFAULT} 식],...);
```

```
Identifier type_name;
```




복합 데이터 타입(12)

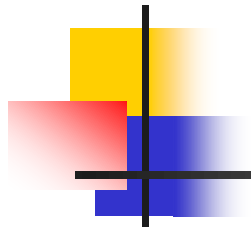
- PL/SQL 레코드(Cont'd)

- 예

```
DECLARE
  TYPE emp_record IS RECORD
    (v_empno  NUMBER,
     v_ename  VARCHAR2(30) );

  emp_rec  emp_record ;
BEGIN
  SELECT empno, ename
  INTO emp_rec.v_empno, emp_rec.v_ename
  FROM emp
  WHERE empno = 7369;

  DBMS_OUTPUT.PUT_LINE( '번호:' || emp_rec.v_empno);
  DBMS_OUTPUT.PUT_LINE( '이름:' || emp_rec.v_ename);
END;
```



PL/SQL 제어 구조

- 조건 제어
 - IF-THEN-ELSE
- 반복 제어
 - LOOP, FOR-LOOP, WHILE-LOOP
- 순차 제어
 - GOTO, NULL



조건 제어 : 형식

- 조건 제어 : IF-THEN-ELSE
- 형식

```
IF condition THEN
    statements;
[ELSIF condition THEN
    statements;]
[ELSE
    statements;]
END IF;
```



조건 제어 : 예

- 조건 제어 예

```
DECLARE
    sMonth      CHAR(2);
BEGIN
    SELECT      TO_CHAR(SYSDATE, 'MM')
    INTO        sMonth
    FROM        DUAL;

    IF ( sMonth>='03' AND sMonth<='08' ) THEN
        DBMS_OUTPUT.PUT_LINE('1학기');
    ELSE
        DBMS_OUTPUT.PUT_LINE('2학기');
    END IF;
END;
```



반복 제어 : 형식(1)

- 반복 제어 : LOOP

```
LOOP  
    statements;  
    .....  
EXIT [WHEN condition];  
END LOOP;
```

- 반복 제어 : FOR-LOOP

```
FOR index in [REVERSE] 시작값..끝값 LOOP  
    statements;  
    .....  
END LOOP;
```



반복 제어 : 형식(2)

- 반복 제어 : WHILE-LOOP

```
WHILE condition LOOP  
    statements;  
    .....  
END LOOP;
```



반복 제어 : 예(LOOP)

```
DECLARE
  i          NUMBER := 0;
  nSum       NUMBER := 0;
BEGIN
  LOOP
    i        := i + 1;
    nSum     := nSum + i;
    EXIT WHEN i >= 100;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('1~100까지의 합 : ' || TO_CHAR(nSum));
END;
```




반복 제어 : 예(FOR-LOOP)

```
DECLARE
  i          NUMBER := 0;
  nSum       NUMBER := 0;
BEGIN
  FOR i IN 1..100 LOOP
    nSum      := nSum + i;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('1~100까지의 합 : ' || TO_CHAR(nSum));
END;
```



반복 제어 : 예(WHILE-LOOP)

```
DECLARE
  i          NUMBER := 0;
  nSum       NUMBER := 0;
BEGIN
  WHILE i < 100 LOOP
    i      := i + 1;
    nSum   := nSum + i;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('1~100까지의 합 : ' || TO_CHAR(nSum));
END;
```



순차 제어(1)

■ GOTO

- 제어가 건너뛰는 곳을 지정하는 레이블과 함께 쓰임
- 예)

```
DECLARE
    i  NUMBER;
BEGIN
    FOR i IN 1..50 LOOP
        IF i = 30 THEN
            GOTO my_label;
        END IF;
    END LOOP;

    <<my_label>>
    DBMS_OUTPUT.PUT_LINE('i = 30');
END;
```



순차 제어(2)

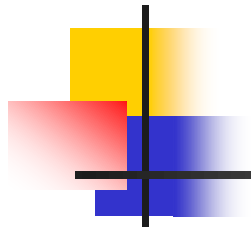
- NULL

- 실행을 하지 않음을 나타냄
- 예)

```
DECLARE
    i          NUMBER := 0;
    nValue     NUMBER := 0;
BEGIN
    FOR i IN 0..100 LOOP
        nValue := 1000/i;
    END LOOP;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        NULL;
END;
```

데이터베이스 프로그래밍

(소프트웨어 개발 트랙)



제 2부

저장 프로시저 프로그래밍(2)



PL/SQL문 내에서의 SQL문



SELECT문(1)

■ 형식

SELECT	select_list
INTO	variable_name record_name
FROM	table
WHERE	condition;

■ 주의사항

- 반드시 하나의 행만을 추출
- 추출되는 데이터 행이 없거나 하나를 초과할 경우 예외 발생
 - **TOO_MANY_ROWS** : 하나 이상의 데이터 행 추출 시
 - **NO_DATA_FOUND** : 어떤 데이터도 추출하지 못할 때
- 다수 개의 데이터 행을 하나씩 추출할 때는 명시적 커서 사용



SELECT문(2) : 예

- shop 테이블

shop_no	shop_name
1015	ART_STAR
1016	MUSIC_STAR



SELECT문(3) : 예

- TOO_MANY_ROWS 에러 발생 예

```
DECLARE
    v_no      shop.shop_no%TYPE;
    v_name    shop.shop_name%TYPE;
BEGIN
    SELECT    shop_no, shop_name
    INTO      v_no, v_name
    FROM      shop;

    DBMS_OUTPUT.PUT_LINE('상점번호: ' || v_no);
    DBMS_OUTPUT.PUT_LINE('상점이름: ' || v_name);
END;
```



SELECT문(4) : 예

- NO_DATA_FOUND 에러 발생 예

```
DECLARE
    v_no      shop.shop_no%TYPE;
    v_name    shop.shop_name%TYPE;
BEGIN
    SELECT    shop_no, shop_name
    INTO      v_no, v_name
    FROM      shop
    WHERE     shop_no=1000;

    DBMS_OUTPUT.PUT_LINE('상점번호: ' || v_no);
    DBMS_OUTPUT.PUT_LINE('상점이름: ' || v_name);
END;
```



INSERT문

- 형식
 - SQL 문과 동일
- 예

```
DECLARE
    v_no      shop.shop_no%TYPE;
    v_name    shop.shop_name%TYPE;
BEGIN
    SELECT      shop_no, shop_name
    INTO        v_no, v_name
    FROM        shop
    WHERE       shop_name = 'MUSIC_STAR';

    INSERT INTO shop (shop_no, shop_name)
    VALUES (v_no+1, v_name);
END;
```



UPDATE문

- 형식
 - SQL 문과 동일
- 예

```
DECLARE
    v_name  shop.shop_name%TYPE;
BEGIN
    v_name := 'PICTURE_STAR';
    UPDATE shop
    SET     shop_name=v_name
    WHERE  shop_name='MUSIC_STAR';
END;
```



DELETE문

- 형식
 - SQL 문과 동일
- 예

```
BEGIN
    DELETE FROM  shop
    WHERE        shop_name = 'PICTURE_STAR';
END;
```



커서(Cursor)

- SQL 처리 결과가 저장된 작업 영역에 이름을 지정하고 저장된 정보를 접근할 수 있게 함
 - SQL 명령을 실행시키면 서버는 명령을 **parse**하고 실행하기 위한 메모리 영역을 **open**하는데 이 영역을 **cursor**라고 부른다.
- 커서의 종류
 - 묵시적 커서(Implicit Cursor)
 - 모든 DML과 PL/SQL SELECT 문에 묵시적으로 PL/SQL이 선언
 - 명시적 커서(Explicit Cursor)
 - 프로그래머가 선언하고 명령하며 블록의 실행 가능한 부분에서 특정 명령을 통해 조작



목시적 커서

- SQL 문장이 처리되는 곳에 대한 익명의 어드레스
- 오라클 데이터 베이스에서 실행되는 모든 SQL 문장은 목시적인 커서
- SQL 문이 실행되는 순간 자동으로 열림과 닫힘 실행
- 속성

속성	설명
SQL%ROWCOUNT	해당 SQL 문에 영향을 받는 행의 수
SQL%FOUND	해당 SQL 문의 영향을 받는 행의 수가 1개 이상일 경우 TRUE
SQL%NOTFOUND	해당 SQL 문에 영향을 받는 행의 수가 없을 경우 TRUE
SQL%ISOPEN	목시적 커서가 열려 있는지의 여부 검색 항상 FALSE(실행한 후 바로 커서를 닫기 때문)



목시적 커서 : 예

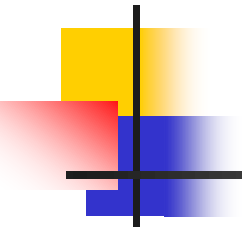
```
DECLARE
    v_sal                emp.sal%TYPE;
    v_update_row         NUMBER;
    v_empno              emp.empno%TYPE;
BEGIN
    v_empno := 20011234;
    SELECT  sal
    INTO    v_sal
    FROM    emp
    WHERE   empno = v_empno ;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('데이터 존재 : ' || v_sal);
    END IF;

    UPDATE emp
    SET     sal = sal * 1.1
    WHERE   empno = v_empno;

    v_update_row := SQL%ROWCOUNT;

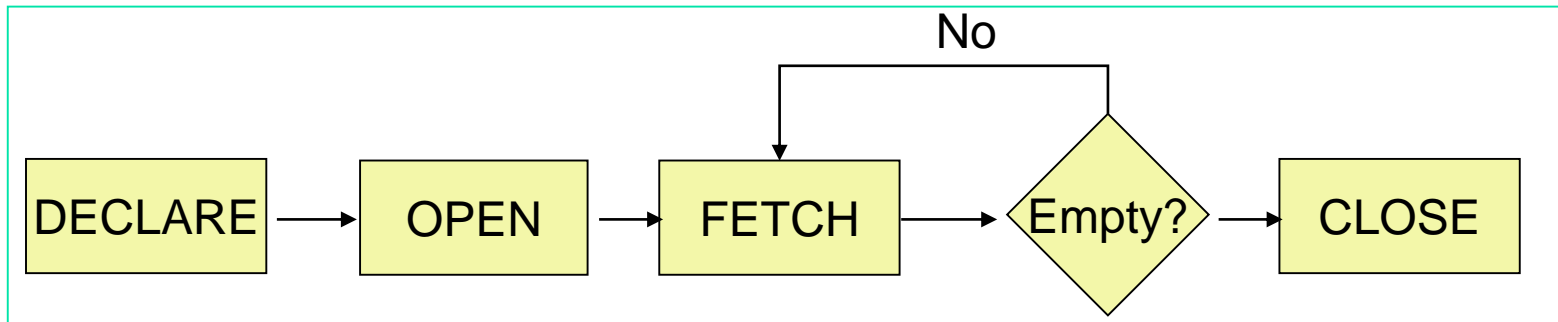
    DBMS_OUTPUT.PUT_LINE('급여인상 인원수: ' || v_update_row);
END;
```



PL/SQL문 내에서의 SQL문

: 명시적 커서

■ 명시적 커서의 흐름



- **DECLARE** : 이름이 있는 **SQL** 영역 생성
- **OPEN** : 커서 활성화
- **FETCH** : 커서의 현재 데이터 행을 해당 변수에 넘김
- **EMPTY** : 현재 데이터 행의 존재 여부 검사,
레코드가 없으면 **FETCH** 하지 않음
- **CLOSE** : 커서가 사용한 자원 해제



커서 선언과 처리(1)

- 커서 선언

```
DECLARE  
    CURSOR cursor_name IS  
        SELECT문;
```

- 커서 연결

```
OPEN cursor_name;
```

- 커서 안의 검색 실행
- 검색 시 데이터 행을 추출하지 못할 경우에는 예외 발생



커서 선언과 처리(2)

- 커서로부터의 데이터 **FETCH**

```
FETCH cursor_name INTO variable1, variable2,...;
```

- 현재 데이터 행을 **OUTPUT** 변수에 리턴
- 한 라인씩 데이터를 **FETCH**
- 주의할 점
 - 커서의 **SELECT** 문의 컬럼의 수와 **OUTPUT** 변수의 수가 동일해야 함
 - 커서 컬럼의 변수의 타입과 **OUTPUT** 변수의 데이터 타입이 동일해야 함



커서 선언과 처리(3)

- 커서 닫기

```
CLOSE cursor_name;
```

- 사용을 끝낸 커서는 반드시 닫아 주어야 함
- 필요하다면 커서를 **OPEN**을 통해 다시 열 수 있음
- 커서를 닫은 상태에서 **FETCH** 할 수 없음

커서 선언과 처리(4) : 예

```
DECLARE
    CURSOR dept_cnt IS
        SELECT      b.dname, COUNT(a.empno) cnt
        FROM        emp a, dept b
        WHERE       a.deptno = b.deptno AND b.deptno = 234
        GROUP BY   b.dname ;

    v_dname  dept.dname%TYPE;
    emp_cnt  NUMBER;
BEGIN
    OPEN dept_cnt;

    FETCH dept_cnt INTO v_dname, emp_cnt;

    DBMS_OUTPUT.PUT_LINE('부서명 : ' || v_dname);
    DBMS_OUTPUT.PUT_LINE('사원수 : ' || emp_cnt);

    CLOSE dept_cnt;

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE(SQLERRM || '에러 발생');
END;
```

명시적 커서의 속성(1)

■ 속성

속성	설명
%ROWCOUNT	현재까지 반환된 모든 데이터 행의 수
%FOUND	FETCH한 데이터가 행을 리턴하면 TRUE
%NOTFOUND	FETCH한 데이터가 행을 리턴하지 않으면 TRUE
%ISOPEN	커서가 열려 있으면 TRUE

- %ROWCOUNT : 정확한 숫자만큼의 행 추출에 사용
- %NOTFOUND : 루프 종료 시점 검사 시 사용
- %ISOPEN : 커서가 열려 있는지 검사 시 사용



명시적 커서의 속성(2) : 예

```
DECLARE
    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;
        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;
```



FOR문에서 커서 사용(1)

■ 특징

- FOR 문을 사용하면 커서의 OPEN, FETCH, CLOSE가 자동 발생하므로 따로 기술할 필요가 없음
- 레코드 이름이 자동 선언되므로 따로 선언 필요 없음

■ 형식

```
FOR record_name IN cursor_name LOOP
    statement1;
    .....
END LOOP;
```



FOR문에서 커서 사용(2)

```
DECLARE
  CURSOR dept_cnt IS
    SELECT  b.dname, COUNT(a.empno) cnt
    FROM    emp a, dept b
    WHERE   a.deptno = b.deptno
    GROUP BY b.dname;
BEGIN
  FOR emp_list IN dept_cnt LOOP
    DBMS_OUTPUT.PUT_LINE('부서명 : ' || emp_list.dname);
    DBMS_OUTPUT.PUT_LINE('사원수 : ' || emp_list.cnt);
  END LOOP;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLERRM || '에러 발생');
END;
```



파라미터가 있는 커서(1)

- 특징

- 커서가 열리고 질의가 실행되면 파라미터 값을 커서에 전달
- 다른 **active set**을 원할 때마다 커서를 따로 선언해야 함

- 형식

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



파라미터가 있는 커서(2) : 예

```
DECLARE
    param_deptno dept.deptno%TYPE;

    CURSOR emp_list(v_deptno emp.deptno%TYPE) IS
        SELECT  ename
        FROM    emp
        WHERE   deptno = v_deptno;
BEGIN
    DBMS_OUTPUT.PUT_LINE('** 입력한 부서 사람들 ** ');

    param_deptno := 10;

    -- Parameter변수의 값을 전달(OPEN될 때 값 전달)
    FOR emplst IN emp_list(param_deptno) LOOP
        DBMS_OUTPUT.PUT_LINE('이름 : ' || emplst.ename);
    END LOOP;

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('ERR MESSAGE : ' || SQLERRM);
END;
```



WHERE CURRENT OF(1)

■ 특징

- ROWID를 이용하지 않고도 현재 참조하는 행을 갱신하고 삭제할 수 있게 함
- FETCH 문에 의해 가장 최근에 처리된 행을 참조하기 위해서 "WHERE CURRENT OF 커서이름" 절로 DELETE나 UPDATE 문 작성 가능
- 주의사항
 - 이 절을 사용할 때 참조하는 커서가 있어야 함
 - FOR UPDATE 절이 커서 선언 질의문 안에 있어야 함 그렇지 않으면 에러발생



WHERE CURRENT OF(2) : 예

```
DECLARE
  CURSOR emp_list IS
    SELECT empno
    FROM emp
    WHERE empno = 7934
    FOR UPDATE;
BEGIN
  FOR emplst IN emp_list LOOP
    --emp_list커서에 해당하는 사람의 직업을 수정
    UPDATE emp
    SET job = 'SALESMAN'
    WHERE CURRENT OF emp_list;

    DBMS_OUTPUT.PUT_LINE('수정 성공');

  END LOOP;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('ERR MESSAGE : ' || SQLERRM);
END;
```



예외 처리



예외 종류

예외	설명	처리
미리 정의된 오라클 서버 예외 (Predefined Exceptions)	PL/SQL에서 자주 발생 하는 약 20개의 오류	선언할 필요도 없고, 발 생 시에 예외 절로 자동 트랩(Trap)
미리 정의되지 않은 오라클 서버 예외 (undefined exceptions)	미리 정의된 오라클 서버 오류를 제외한 모든 오류	선언부에서 선언해야 하고 발생 시 자동트랩
사용자 정의 예외 (user-defined exceptions)	개발자가 정한 조건에 만족하지 않을 경우 발생 하는 오류	선언부에서 선언하고 실행부에서 RAISE 문을 사용하여 발생



예외 처리 형식(1)

EXCEPTION

 WHEN exception1 [OR exception2] THEN
 statement;

 [WHEN exception3 [OR exception4] THEN
 statement;

]

 [WHEN OTHERS THEN
 statement;

]



예외 처리 형식(2)

- 주의사항
 - **WHEN OTHERS** 절은 맨 마지막에 위치
 - 예외 처리절은 **EXCEPTION**부터 시작
 - 여러 개의 예외 처리부 허용
 - 예외가 발생하면 여러 개의 예외 처리부 중 하나의 예외 처리부로 트랩(Trap)



미리 정의된 예외(1)

- NO_DATA_FOUND
 - SELECT 문이 아무런 데이터 행을 반환하지 못할 때
- TOO_MANY_ROWS
 - SELECT 문이 하나를 초과한 행을 반환할 때
- INVALID_CURSOR
 - 잘못된 커서 연산
- ZERO_DIVIDE
 - 0으로 나눌 때
- DUP_VAL_ON_INDEX
 - UNIQUE 제약 컬럼에 중복되는 데이터가 INSERT 될 때
- TIMEOUT_ON_RESOURCE
 - 자원을 기다리는 동안 타임아웃이 발생하는 경우



미리 정의된 예외(2)

- **INVALID_NUMBER**
 - 숫자 데이터 에러 : '3D2'는 숫자가 아님
- **STORAGE_ERROR**
 - 메모리 부족으로 일어나는 PL/SQL 내부 에러
- **PROGRAM_ERROR**
 - 내부 PL/SQL 에러
- **VALUE_ERROR**
 - 숫자의 계산, 변환 또는 버림 등에서 발생하는 에러
- **ROWTYPE_MISMATCH**
 - 호스트의 커서 변수와 PL/SQL 커서 변수의 타입이 맞지 않을 때 발생
- **CURSOR_ALREADY_OPEN**
 - 이미 열려있는 커서를 다시 열려고 할 때 발생

미리 정의된 예외(3) : 예

결과 : 'TOO_MANY_ROWS'에러 발생'

```
DECLARE
    v_emp emp%ROWTYPE;
BEGIN
    SELECT empno, ename, deptno
    INTO    v_emp.empno, v_emp.ename, v_emp.deptno
    FROM    emp
    WHERE   deptno = 20 ;

    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('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;
```



미리 정의되지 않은 예외(1)

■ 처리 방법

- 1단계 : 예외의 이름을 선언(선언절)
- 2단계 : **PRAGMA EXCEPTION_INIT** 문장으로
예외의 이름과 오라클 서버 오류 번호를 결합 (선언절)
- 3단계 : 예외가 발생할 경우 해당 예외 참조(예외절)

미리 정의되지 않은 예외(2) : 예

DECLARE

`not_null_test EXCEPTION;` -- 단계1

/ not_null_test는 선언된 예외 이름*

*-1400 Error 처리번호는 표준 Oracle Server Error 번호 */*

`PRAGMA EXCEPTION_INIT(not_null_test, -1400);` -- 단계2

BEGIN

-- empno를 입력하지 않아서 NOT NULL 에러 발생

`INSERT INTO emp(ename, deptno)`

`VALUES('tiger', 30);`

EXCEPTION

`WHEN not_null_test THEN` -- 단계 3

`DBMS_OUTPUT.PUT_LINE('not null 에러 발생');`

END;

결과 : not null 에러 발생'



사용자 정의 예외(1)

- 오라클 저장함수 `RAISE_APPLICATION_ERROR`를 사용하여 오류코드 -20000부터 -20999의 범위 내에서 사용자 정의 예외를 만들 수 있음
- 처리 방법
 - 1단계 : 예외의 이름을 선언(선언절)
 - 2단계 : **RAISE**문을 사용하여 직접적으로 예외 발생 (실행절)
 - 3단계 : 예외가 발생할 경우 해당 예외 참조 (예외절)

사용자 정의 예외(2) : 예

```
DECLARE
  -- 예외의 이름을 선언
  user_define_error EXCEPTION;    -- 단계 1
  cnt  NUMBER;
BEGIN
  SELECT COUNT(empno)
  INTO   cnt
  FROM   emp
  WHERE  deptno = 234;

  IF cnt < 5 THEN
    -- RAISE문을 사용하여 직접적으로 예외 발생
    RAISE user_define_error;      -- 단계 2
  END IF;

  EXCEPTION
    -- 예외가 발생할 경우 해당 예외를 참조
    WHEN user_define_error THEN    -- 단계 3
      RAISE_APPLICATION_ERROR(-20001, '사원 부족');
END;
```



SQLCODE, SQLERRM(1)

- WHEN OTHERS 문으로 트랩(Trap)되는 오류들의 실제 오류 코드와 설명을 볼 때 사용
- SQLCODE
 - 실행된 프로그램이 성공적으로 종료하였을 때는 오류번호 0을 포함하며, 그렇지 못할 경우에는 해당 오류코드 번호 포함
- SQLERRM
 - SQLCODE에 포함된 오라클 오류 번호에 해당하는 메시지를 가짐



SQLCODE, SQLERRM(2)

SQLCODE 값	설명
0	오류 없이 성공적으로 종료
1	사용자 정의 예외 번호
+100	NO_DATA_FOUND 예외 번호
음수	위의 것을 제외한 오라클 서버 에러 번호

SQLCODE, SQLERRM(3) : 예

```
DECLARE
    v_emp emp%ROWTYPE ;
BEGIN
    SELECT *
    INTO    v_emp
    FROM    emp;

    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;
```

결과>

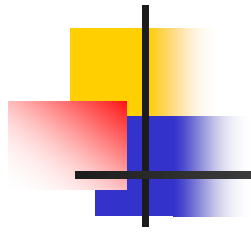
ERR CODE : -1422

ERR MESSAGE : ORA-01422 : exact fetch returns more than requested number of rows

PL/SQL procedure successfully completed.

데이터베이스 프로그래밍

(소프트웨어 개발 트랙)



제 2부

저장 프로시저 프로그래밍(3)



프로시저(함수)

- 개념

- 매개변수를 받을 수 있고 반복해서 사용할 수 있는 이름이 있는 PL/SQL 블록

- 용도

- 연속 실행 또는 구현이 복잡한 트랜잭션을 수행하는 PL/SQL 블록을 데이터베이스에 저장하기 위해 생성



저장 프로시저(함수) 사용 이유

- 저장 프로시저(함수)를 사용하는 이유
 - 정보 캡슐화
 - 기능의 재사용
 - 트랜잭션 제어
 - 데이터베이스 내에서 미리 컴파일 되어 저장되므로 필요할 때마다 매번 다시 변환해야 하는 **SQL** 문보다 빠르게 실행
 - 저장 프로시저에서 발생하는 문법 오류는 실행시간이 아닌 컴파일 할 때 바로 잡을 수 있음

- CREATE OR REPLACE 구문을 사용하여 생성
- IS 로 PL/SQL의 블록 시작
- LOCAL 변수는 IS 와 BEGIN 사이에 선언

```
CREATE [OR REPLACE] PROCEDURE procedure name
  IN argument
  OUT argument
  IN OUT argument
IS
  [변수 선언]
BEGIN -- 필수
  [PL/SQL Block]
  -- SQL문장, PL/SQL제어 문장

  [EXCEPTION] --> 선택
  -- error가 발생할 때 수행하는 문장
END; -- 필수
```



파라미터

■ 특징

- 실행 환경과 프로그램 사이에 값을 주고 받는 역할
- 블록 안에서의 변수와 똑같이 일시적으로 값을 저장하는 역할

■ 파라미터 종류

■ IN

- 실행환경에서 프로그램으로 값을 전달
- 상수, 수식 또는 초기화된 변수 사용
- 디폴트, 생략 가능

■ OUT

- 프로그램으로부터 실행환경으로 값을 전달
- 초기화되지 않은 변수를 매개변수로 사용
- 반드시 지정

■ INOUT

- 실행환경에서 프로그램으로 값을 전달하고, 다시 프로그램으로부터 실행환경으로 변경된 값을 전달
- 초기화된 변수를 사용
- 반드시 지정



프로시저의 생성과 실행

- 생성
 - CREATE OR REPLACE 구문을 사용하여 생성
 - 프로시저를 끝마칠 때는 “/”를 지정
- 실행
 - EXECUTE 프로시저명;
- 프로시저 에러 검사
 - SHOW ERROR
- 삭제
 - DROP PROCEDURE 프로시저명



프로시저 예(1)

- 생성

```
SQL>CREATE OR REPLACE PROCEDURE update_sal
/* IN Parameter */
(v_empno      IN   NUMBER)
IS

BEGIN

    UPDATE emp
    SET      sal = sal * 1.1
    WHERE empno = v_empno;

    COMMIT;

END;
/
```

프로시저가 생성되었습니다.



프로시저 예(2)

- 에러 검사

```
SQL> SHOW ERROR
```

```
No errors.
```

- 실행

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

```
PL/SQL 처리가 정상적으로 완료되었습니다.
```



함수

■ 개념

- 매개 변수를 받을 수 있고, 반복해서 사용할 수 있는 이름이 있는 **PL/SQL 블록**

■ 용도

- 연속 실행 또는 구현이 복잡한 트랜잭션을 수행하는 **PL/SQL 블록**을 데이터베이스에 저장하기 위해 생성

■ 프로시저와의 차이점 및 특징

- 결과값 리턴
- 대부분 구성이 프로시저와 유사하지만 **IN** 파라미터만 사용 가능
- 리턴 될 값의 데이터 타입을 **RETURN**문에 선언
- **PL/SQL 블록** 내에서 **RETURN**문을 통해서 반드시 값을 반환해야 함

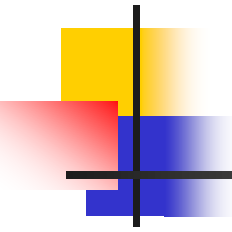
- CREATE OR REPLACE 구문을 사용하여 생성
- IS 로 PL/SQL의 블록 시작
- LOCAL 변수는 IS 와 BEGIN 사이에 선언

```
CREATE [OR REPLACE] FUNCTION function name
    [(argument...)]
    RETURN datatype
    -- datatype은 반환되는 값의 datatype
IS
    [변수 선언 부분]
BEGIN
    [PL/SQL Block]
    -- PL/SQL 블록에는 적어도 한 개의 RETURN 문이 있어야 함
END;
```



함수의 생성과 실행

- 생성
 - CREATE OR REPLACE 구문을 사용하여 생성
 - 함수를 끝마칠 때는 “/”를 지정
- 실행
 - 함수의 리턴 값을 저장할 변수 선언
 - EXECUTE :변수명 := 함수명
 - PRINT 변수명
- 함수 에러 검사
 - SHOW ERROR
- 삭제
 - DROP FUNCTION 함수명



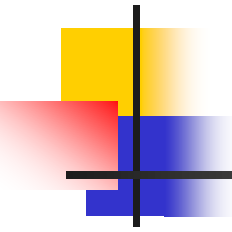
함수 예(1)

- 생성

```
SQL> CREATE OR REPLACE FUNCTION FC_update_sal
      (v_empno IN NUMBER)
      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;

      RETURN v_sal;
END;
/
```

함수가 생성되었습니다.



함수 예(2)

- 에러 검사

```
SQL> SHOW ERROR  
No errors.
```

- 실행

```
SQL> VAR salary NUMBER;  
SQL> EXECUTE :salary := FC_update_sal(7900);  
SQL> PRINT salary;  
    SALARY  
-----  
      1045
```



트리거(Trigger)

■ 트리거 (Trigger)

- 데이터베이스에 특정한 변경이 가해졌을 때 **DBMS**가 이에 대응해서 자동적으로 호출하는 일종의 프로시저
- 프로시저와 함수는 그 실행이 외부적인 실행 명령에 의해 이루어지는데 반해, 트리거의 실행은 트리거링 사건(**Triggering Event**)에 의해 내부적으로 이루어짐
- 트리거를 일으키는 사건(**event**)
 - 데이터베이스 테이블에 **DML** 문이 발생할 때
- **INSERT, UPDATE, DELETE**문의 사용에 사건을 정의할 수 있으며 이들을 실행할 때 정의된 트리거도 자동 실행
- 테이블과 별도로 데이터베이스에 저장
- 뷰가 아니라 테이블에 대해서만 정의

- 트리거 구성
 - 사건(event) : 트리거를 가동
 - 조건(condition) : 트리거 수행 여부 검사
 - 동작 (action) : 트리거가 수행될 때 일어나는 일
- 예

```
CREATE TRIGGER incr_count
  BEFORE INSERT ON student      -- 사건
  FOR EACH ROW
  WHEN (:new.age < 18)           -- 조건
  BEGIN                          -- 동작
    DBMS_OUTPUT.PUT_LINE('미성년자 : ' || :new.ename);
  END;
```


■ 트리거의 용도

- 테이블 생성시 참조 무결성과 데이터 무결성 그 밖의 다른 제약 조건으로 정의할 수 없는 복잡한 요구 사항에 대한 제약조건을 생성할 수 있다.
- 테이블의 데이터에 생기는 작업을 감시, 보안할 수 있다.
- 테이블에 생기는 변화에 따라 필요한 다른 프로그램을 실행시킬 수 있다.

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

- BEFORE
 - INSERT, UPDATE, DELETE 문이 실행되기 전 트리거 실행
- AFTER
 - INSERT, UPDATE, DELETE 문이 실행된 후 트리거 실행
- trigger_event
 - INSERT, UPDATE, DELETE 중 한 개 이상
- FOR EACH ROW
 - 행 트리거



문장 트리거와 행 트리거(1)

- 문장 트리거(Statement-Level Trigger)
 - 트리거링 사건에 의해 단 한 번 실행
 - 컬럼의 각 데이터 행 제어 불가능
 - 컬럼의 데이터 값에 상관없이 그 컬럼에 변화가 일어남을 감지하여 실행되는 트리거
- 행 트리거(Row-Level Trigger)
 - 컬럼의 각각의 데이터 행에 변화가 생길 때마다 실행
 - 변화가 생긴 데이터 행의 실제 값 제어 가능
 - 데이터 행의 실제 값을 수정, 변경 또는 저장할 때 사용



문장 트리거와 행 트리거(2)

- 행 트리거(Row-Level Trigger)의 컬럼 값 참조
 - “:old”, “:new” 연산자 사용
 - INSERT 문
 - 입력할 데이터 값은 :new.column_name 으로 참조
(단, column_name은 테이블의 컬럼 이름)
 - UPDATE 문
 - 변경하기 전 컬럼 데이터 값은 :old.column_name 으로 참조
 - 수정할 새로운 데이터 값은 :new.column_name 으로 참조
 - DELETE 문
 - 삭제되는 데이터 값은 :old.column_name 으로 참조



트리거 예(1)

```
SQL> CREATE OR REPLACE TRIGGER trigger_test
  BEFORE
  UPDATE ON dept
  FOR EACH ROW
  BEGIN
    DBMS_OUTPUT.PUT_LINE('변경 전 컬럼 값 : ' || :old.dname);
    DBMS_OUTPUT.PUT_LINE('변경 후 컬럼 값 : ' || :new.dname);
  END;
/
```

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

```
SQL>UPDATE dept
  SET dname = '총무부'
  WHERE deptno = 30
```

-- UPDATE문 실행 전 트리거링 발생

변경 전 컬럼 값 : 인사과

변경 후 컬럼 값 : 총무부

1 행이 갱신되었습니다.



트리거 예(2)

```
SQL>CREATE OR REPLACE TRIGGER sum_trigger
BEFORE
INSERT OR UPDATE ON emp
FOR EACH ROW

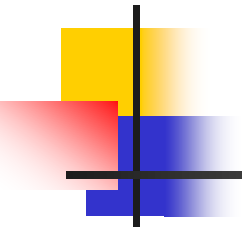
DECLARE
    -- 변수를 선언할 때 DECLARE문 사용
    avg_sal NUMBER;
BEGIN
    SELECT ROUND(AVG(sal), 3)
    INTO avg_sal
    FROM emp;
    DBMS_OUTPUT.PUT_LINE('급여 평균 : ' || avg_sal);
END;
/
```

트리거가 생성되었습니다.

```
SQL> SET SERVEROUTPUT ON ;
SQL> INSERT INTO EMP(EMPNO, ENAME, JOB, HIREDATE, SAL)
VALUES(1000, 'LION', 'SALES', SYSDATE, 5000);
```

```
-- INSERT문이 실행되기 전까지의 급여 평균 출력
급여 평균 : 2073.214
```

1 개의 행이 만들어졌습니다.



프로시저, 함수, 트리거 실습 : 수강신청 시스템



수강신청 입력 : 프로시저 및 함수 적용

- 수강신청 입력 시 예외 처리가 매우 많음
 - 이를 프로시저와 함수를 이용하여 작성
 - 수강신청을 입력할 때 복잡한 예외 처리를 그때마다 할 필요 없이 프로시저와 함수를 이용함으로써 손쉽게 처리 가능



수강신청 입력 요구사항 리뷰(1)

- 사용 사례명 : 수강신청 입력

- 액터 : 학생

- 선행조건 : 로그인

- 주요 흐름

1. 시스템은 아직 수강 신청하지 않은 과목들을 학생에게 보여준다.

2. 학생은 수강 신청하고자 하는 과목을 선택한다.

3. 시스템은 선택된 과목을 수강 신청된 것으로 등록한다.

이 때, 시스템은 최대 학점을 초과했는지(E-1),

동일한 과목을 신청했는지 (E-2),

해당 과목에 대한 수강신청 인원이 초과되었는지(E-3),

동일한 시간의 다른 과목이 이미 수강 신청되었는지(E-4)를 검사한다.



수강신청 입력 요구사항 리뷰(2)

- 예외 흐름

- E-1

1. 시스템은 선택된 과목에 따라, 총 수강신청 과목의 총 학점이 18학점이 초과되는지 검사한다.
2. 18학점을 초과하면, 시스템은 수강신청이 될 수 없음을 알린다.

- E-2

1. 시스템은 선택된 과목이 이미 수강 신청되어 있는 과목인지 검사한다.
2. 이미 수강 신청되어 있는 과목인 경우, 시스템은 수강신청이 될 수 없음을 알린다.



수강신청 입력 요구사항 리뷰(3)

■ E-3

1. 시스템은 해당과목에 대한 수강신청 인원이 초과되었는지 검사한다.
2. 수강신청 인원이 초과된 과목인 경우, 시스템은 수강신청이 될 수 없음을 알린다.

■ E-4

1. 시스템은 동일한 시간의 다른 과목이 이미 수강 신청되었는지 검사한다.
2. 동일한 시간의 다른 과목이 이미 수강 신청되어 있는 경우, 시스템은 수강신청이 될 수 없음을 알린다.



수강신청 입력 요구사항 리뷰(4)

- 수강신청 년도와 학기에 대한 요구사항
 - 수강신청 년도와 학기는 현재 날짜가 11월, 12월인 경우는 다음 년도 1학기인 것으로 하고, 1월~4월인 경우는 현재 년도 1학기로 하며, 5월~10월은 현재 년도 2학기인 것으로 한다.
(수강신청 삭제 사용사례 명세에 나타남)

```

CREATE OR REPLACE FUNCTION Date2EnrollYear(dDate IN DATE)
RETURN NUMBER
IS
  nYear  NUMBER;
  sMonth CHAR(2);
BEGIN
  /* 11월 ~ 4월 : 1학기,  5월 ~ 10월 : 2학기 */
  SELECT  TO_NUMBER(TO_CHAR(dDate, 'YYYY')),
          TO_CHAR(dDate, 'MM')
  INTO    nYear, sMonth
  FROM    DUAL;

  IF (sMonth ='11' OR sMonth='12') THEN
    nYear := nYear + 1;
  END IF;

  RETURN nYear;
END;
/

```

함수 사용



Cont.

```
CREATE OR REPLACE FUNCTION Date2EnrollSemester(dDate IN DATE)
RETURN NUMBER
IS
    nSemester  NUMBER;
    sMonth      CHAR(2);
BEGIN
    /* 11월 ~ 4월 : 1학기, 5월 ~ 10월 : 2학기 */
    SELECT TO_CHAR(dDate, 'MM')
    INTO    sMonth
    FROM    DUAL;

    IF (sMonth='11' OR sMonth='12' OR (sMonth >='01' AND sMonth<='04')) THEN
        nSemester := 1;
    ELSE
        nSemester := 2;
    END IF;

    RETURN nSemester;
END;
/
```



수강신청 입력 프로시저 : InsertEnroll

- InsertEnroll(p1, p2, p3, p4)
 - IN 파라미터
 - P1 : 학번
 - P2 : 과목번호
 - P3 : 분반
 - OUT 파라미터
 - p4 : 입력 결과 메시지
 - 수강신청 등록이 완료되었습니다.
 - 최대학점을 초과하였습니다.
 - 이미 등록된 과목을 신청하였습니다.
 - 수강신청 인원이 초과되어 등록이 불가능합니다
 - 이미 등록된 과목 중 중복되는 시간이 존재합니다.
 - 그 외 에러 : **SQLCODE**



수강신청 입력 프로시저 : InsertEnroll

- InsertEnroll(p1, p2, p3, p4)

- 결과

- 예외흐름이 아닌 경우 “enroll” 테이블에 해당 학번, 과목 번호, 분반, 현재 년도, 현재 학기가 입력된다.
 - 예외가 발생한 경우는 오류 메시지를 보내고 테이블에 입력되지 않는다.



수강신청 관련 함수

- Date2EnrollYear(p1)
 - IN 파라미터
 - p1 : 오늘 날짜
 - 리턴 결과
 - 숫자형
 - 수강 신청하고 있는 년도 리턴

- Date2EnrollSemester (p1)
 - IN 파라미터
 - p1 : 오늘 날짜
 - 리턴 결과
 - 숫자형
 - 수강 신청하고 있는 학기 리턴

```
CREATE OR REPLACE PROCEDURE InsertEnroll(sStudentId IN VARCHAR2,
                                         sCourseId   IN   VARCHAR2,
                                         nCourseIdNo  IN   NUMBER,
                                         result        OUT  VARCHAR2)
IS
    too_many_sumCourseUnit  EXCEPTION;
    too_many_courses       EXCEPTION;
    too_many_students      EXCEPTION;
    duplicate_time         EXCEPTION;
    nYear                  NUMBER;
    nSemester              NUMBER;
    nSumCourseUnit         NUMBER;
    nCourseUnit            NUMBER;
    nCnt                   NUMBER;
    nTeachMax              NUMBER;
BEGIN
    result := "";

    DBMS_OUTPUT.put_line('#');
    DBMS_OUTPUT.put_line(sStudentId || '님이 과목번호 ' || sCourseId || ', 분반 ' ||
        TO_CHAR(nCourseIdNo) || '의 수강 등록을 요청하였습니다.');
```

Cont.

/* 년도, 학기 알아내기 */

nYear := Date2EnrollYear(SYSDATE);

함수 사용

nSemester := Date2EnrollSemester(SYSDATE);

/* 에러 처리 1 : 최대학점 초과여부 */

SELECT SUM(c.c_unit)

INTO nSumCourseUnit

FROM course c, enroll e

WHERE e.s_id = sStudentId AND e.e_year = nYear AND e.e_semester = nSemester
AND e.c_id = c.c_id AND e.c_id_no = c.c_id_no;

SELECT c_unit

INTO nCourseUnit

FROM course

WHERE c_id = sCourseId and c_id_no = nCourseIdNo;

IF (nSumCourseUnit + nCourseUnit > 18) THEN *사용자 정의 예외*

RAISE too_many_sumCourseUnit;

END IF;



Cont.

/* 에러 처리 2 : 동일한 과목 신청 여부 */

```
SELECT COUNT(*)  
INTO    nCnt  
FROM    enroll  
WHERE   s_id = sStudentId AND c_id = sCourseId;
```

```
IF (nCnt > 0)  
THEN
```

```
    RAISE too many courses;
```

```
END IF;
```

/* 에러 처리 3 : 수강신청 인원 초과 여부 */

```
SELECT t_max  
INTO    nTeachMax  
FROM    teach  
WHERE   t_year= nYear AND t_semester = nSemester  
        AND c_id = sCourseId AND c_id_no= nCourseIdNo;
```

```
SELECT COUNT(*)  
INTO    nCnt  
FROM    enroll  
WHERE   e_year = nYear AND e_semester = nSemester  
        AND c_id = sCourseId AND c_id_no = nCourseIdNo;
```

Cont.

```
IF (nCnt >= nTeachMax) THEN  
    RAISE too many students;  
END IF;
```

```
/* 에러 처리 4 : 신청한 과목들 시간 중복 여부 */  
SELECT COUNT(*)  
INTO    nCnt  
FROM  
(  
    SELECT t_time  
    FROM    teach  
    WHERE   t_year=nYear AND t_semester = nSemester  
           AND c_id = sCourseId AND c_id_no = nCourseIdNo  
    INTERSECT  
    SELECT t.t_time  
    FROM    teach t, enroll e  
    WHERE   e.s_id=sStudentId AND e.e_year=nYear  
           AND e.e_semester = nSemester AND t.t_year=nYear  
           AND t.t_semester = nSemester AND e.c_id=t.c_id  
           AND e.c_id_no=t.c_id_no  
);  
IF (nCnt > 0) THEN  
    RAISE duplicate time;  
END IF;
```



Cont.

```
/* 수강 신청 등록 */  
INSERT INTO enroll(S_ID,C_ID,C_ID_NO,E_YEAR,E_SEMESTER)  
VALUES (sStudentId, sCourseId, nCourseIdNo, nYear, nSemester);  
  
COMMIT;  
result := '수강신청 등록이 완료되었습니다.';  
  
EXCEPTION  
  WHEN too_many_sumCourseUnit THEN  
    result := '최대학점을 초과하였습니다';  
  WHEN too_many_courses THEN  
    result := '이미 등록된 과목을 신청하였습니다';  
  WHEN too_many_students THEN  
    result := '수강신청 인원이 초과되어 등록이 불가능합니다';  
  WHEN duplicate_time THEN  
    result := '이미 등록된 과목 중 중복되는 시간이 존재합니다';  
  WHEN OTHERS THEN  
    ROLLBACK;  
    result := SQLCODE;  
END;  
/
```



수강신청 결과 확인 : 프로시저

: 명시적 커서를 이용한 프로시저 실습

■ SelectTimeTable(p1, p2, p3)

■ IN 파라미터

- p1 : 학번
- p2 : 년도
- p3 : 학기

■ 결과

- 파라미터로 입력한 학번, 년도, 학기에 해당하는 수강신청 시간표를 보여준다.
 - 시간표 정보로 교시, 과목번호, 과목명, 분반, 학점, 장소를 보여줌
 - 총 신청 과목수와 총 학점을 보여줌

```
CREATE OR REPLACE PROCEDURE SelectTimeTable
    (sStudentId IN VARCHAR2,
     nYear IN NUMBER,
     nSemester IN NUMBER)

IS
    sId          COURSE.C_ID%TYPE;
    sName        COURSE.C_NAME%TYPE;
    nIdNo        COURSE.C_ID_NO%TYPE;
    nUnit        COURSE.C_UNIT%TYPE;
    nTime        TEACH.T_TIME%TYPE;
    sWhere       TEACH.T_WHERE%TYPE;
    nTotUnit     NUMBER := 0;
```

명시적 커서 사용

```
CURSOR cur (sStudentId VARCHAR2, nYear NUMBER, nSemester NUMBER) IS
    SELECT e.c_id, c.c_name, e.c_id_no, c.c_unit, t.t_time, t.t_where
    FROM   enroll e, course c, teach t
    WHERE  e.s_id = sStudentId AND e.e_year = nYear
           AND e.e_semester=nSemester AND t.t_year = nYear
           AND t.t_semester = nSemester AND e.c_id = c.c_id
           AND e.c_id_no=c.c_id_no AND c.c_id=t.c_id AND c.c_id_no = t.c_id_no
    ORDER BY 5;
```


Cont.

BEGIN

OPEN cur(sStudentId, nYear, nSemester); *파라미터가 있는 커서 사용*

DBMS_OUTPUT.put_line('#');

DBMS_OUTPUT.put_line(TO_CHAR(nYear) || '년도 ' || TO_CHAR(nSemester) ||
'학기의 ' || sStudentId || '님의 수강신청 시간표입니다.');

LOOP

FETCH cur INTO sId, sName, nIdNo, nUnit, nTime, sWhere;

EXIT WHEN cur%NOTFOUND;

DBMS_OUTPUT.put_line('교시:' || TO_CHAR(nTime) ||

' , 과목번호:' || sId || ' , 과목명:' || sName || ' , 분반:' || TO_CHAR(nIdNo) ||

' , 학점:' || TO_CHAR(nUnit) || ' , 장소:' || sWhere);

nTotUnit := nTotUnit + nUnit;

END LOOP;

DBMS_OUTPUT.put_line('총 ' || TO_CHAR(cur%ROWCOUNT) || ' 과목과 총 ' ||
TO_CHAR(nTotUnit) || '학점을 신청하였습니다.');

CLOSE cur;

END;

/



수강신청 입력 및 입력 후 결과 확인

- InsertEnroll() 프로시저의 입력 결과 확인
 - InsertEnroll() 프로시저 호출
 - 과목번호 'C400' 입력 : 동일한 과목 신청 오류 발생
 - 과목번호 'C900' 입력 : 수강신청 인원 초과 오류 발생
 - 과목번호 'M100' 입력 : 신청한 과목들 시간 중복 발생
 - 과목번호 'C800' 입력 : 정상적 입력
 - 과목번호 'M700' 입력 : 최대 학점 초과 오류 발생
 - 입력 후 결과 확인
 - SelectTimeTable() 프로시저 호출

```
SET SERVEROUTPUT ON;
/

DECLARE
result VARCHAR2(50) := '';
BEGIN

DBMS_OUTPUT.ENABLE;

DBMS_OUTPUT.PUT_LINE
('***** Insert 및 에러 처리 테스트 *****');

/* 에러 처리 2 : 동일한 과목 신청 여부 :데이터베이스 과목 신청 */
InsertEnroll('20011234', 'C400', 3, result);
DBMS_OUTPUT.PUT_LINE('결과 : ' || result);
```



Cont.

```
/* 에러 처리 3 : 수강신청 인원 초과 여부 : 객체지향 윈도우즈 프로그래밍 신청*/
```

```
InsertEnroll('20011234', 'C900', 3, result);
```

```
DBMS_OUTPUT.PUT_LINE('결과 : ' || result);
```

```
/* 에러 처리 4 : 신청한 과목들 시간 중복 여부 : 멀티미디어 개론 신청 */
```

```
InsertEnroll('20011234', 'M100', 3, result);
```

```
DBMS_OUTPUT.PUT_LINE('결과 : ' || result);
```

```
/* 에러가 없는 경우 */
```

```
InsertEnroll('20011234', 'C800', 3, result);
```

```
DBMS_OUTPUT.PUT_LINE('결과 : ' || result);
```

```
/* 에러 처리 1 : 최대 학점 초과 여부 검사 : 게임 프로그래밍 신청 */
```

```
InsertEnroll('20011234', 'M700', 3, result);
```

```
DBMS_OUTPUT.PUT_LINE('결과 : ' || result);
```



Cont.

```
DBMS_OUTPUT.PUT_LINE
  ('***** CURSOR를 이용한 SELECT 테스트 *****');
/* 최종 결과 확인 */
SelectTimeTable('20011234', 2024, 1);

DELETE FROM enroll
WHERE s_id='20011234' AND c_id='C800' AND c_id_no=3;

END;
/
```



InsertTest.sql 실행 결과(1)

```
SQL> @InsertTest
```

```
***** Insert 및 에러 처리 테스트 *****
```

```
#  
20011234님이 과목번호 C400, 분반 3의 수강 등록을 요청하였습니다.  
결과 : 이미 등록된 과목을 신청하였습니다
```

```
#  
20011234님이 과목번호 C900, 분반 3의 수강 등록을 요청하였습니다.  
결과 : 수강신청 인원이 초과되어 등록이 불가능합니다.
```

```
#  
20011234님이 과목번호 M100, 분반 3의 수강 등록을 요청하였습니다.  
결과 : 이미 등록된 과목 중 중복되는 시간이 존재합니다.
```

```
#  
20011234님이 과목번호 C800, 분반 3의 수강 등록을 요청하였습니다.  
결과 : 수강신청 등록이 완료되었습니다.
```

```
#  
20011234님이 과목번호 M700, 분반 3의 수강 등록을 요청하였습니다.  
결과 : 최대학점을 초과하였습니다
```



InsertTest.sql 실행 결과(2)

***** CURSOR를 이용한 SELECT 테스트 *****

#

2024년도 1학기의 20011234님의 수강신청 시간표입니다.

교시:2, 과목번호:C300, 과목명:알고리즘, 분반:3, 학점:3, 장소:인-416

교시:3, 과목번호:C500, 과목명:운영체제, 분반:3, 학점:3, 장소:인-201

교시:4, 과목번호:C100, 과목명:컴퓨터 프로그래밍, 분반:3, 학점:3, 장소:인-201

교시:5, 과목번호:C200, 과목명:자료구조, 분반:3, 학점:3, 장소:인-201

교시:6, 과목번호:C400, 과목명:데이터베이스 시스템, 분반:3, 학점:3, 장소:인-201

교시:7, 과목번호:C800, 과목명:데이터베이스 프로그래밍, 분반:3, 학점:3,
장소:인-309

총 6 과목과 총 18학점을 신청하였습니다.

PL/SQL procedure successfully completed.



사용자 정보 수정 : 트리거 적용

- 사용자 정보 수정 시 비밀번호에 대한 부분의 처리
 - 참조 무결성과 데이터 무결성 그 밖의 다른 제약조건으로 정의할 수 없는 복잡한 요구 사항에 대한 제약조건

!!! 트리거를 적용해보자 !!!



사용자 정보 수정 요구사항 리뷰

- 사용 사례명 : 사용자 정보 수정
 - 액터 : 학생
 - 선행조건 : 로그인
 - 주요 흐름
 1. 시스템은 로그인한 사용자 정보(주소, 비밀번호)를 보여준다.
 2. 학생은 사용자 정보를 수정한다. 이 때, 시스템은 비밀번호가 올바른지 검사한다. (E-1)
 - 예외 흐름
 - E-1
 1. 시스템은 비밀번호가 4자리 이상이고, 공란이 포함되어 있지 않은지 검사한다.
 2. 비밀번호가 4자리 미만이거나 공란이 포함되어 있으면, 시스템은 수정이 불가능함을 알린다.



사용자 정보 수정 트리거 : BeforeUpdateStudent

■ BeforeUpdateStudent

- 관련 테이블 : student
- 트리거 발생 시기 : 수정 전
- 트리거 형태 : 행 트리거 - 행의 실제 값 제어
- 결과
 - 암호의 길이가 4자리 미만인 경우 오류 발생
 - 에러번호는 -20002
 - 에러설명은 '암호는 4자리 이상이어야 합니다'
 - 암호에 공란이 포함된 경우 오류 발생
 - 에러번호는 -20003
 - 에러설명은 '암호에 공란은 입력되지 않습니다.'
 - 암호가 정상적인 경우
 - 수정 완료

```
CREATE OR REPLACE TRIGGER BeforeUpdateStudent BEFORE  
UPDATE ON student  
FOR EACH ROW
```

```
DECLARE
```

underflow_length	EXCEPTION;
invalid_value	EXCEPTION;
nLength	NUMBER;
nBlank	NUMBER;

```
BEGIN
```

```
/* 학년 제약조건 : DDL에서 해결 */  
/* 보다 복잡한 제약조건을 다루기 편하게 함 */  
/* 암호 제약조건 : 4자리 이상, blank는 허용안함 */  
SELECT length(:new.s_pwd), instr(:new.s_pwd,'')  
INTO    nLength, nBlank  
FROM    DUAL;
```



Cont.

```
IF (nLength < 4) THEN
    RAISE underflow_length;
ELSIF (nBlank > 0) THEN
    RAISE invalid_value;
END IF;

EXCEPTION
    WHEN underflow_length THEN
        RAISE_APPLICATION_ERROR
            (-20002, '암호는 4자리 이상이어야 합니다');
    WHEN invalid_value THEN
        RAISE_APPLICATION_ERROR
            (-20003, '암호에 공란은 입력되지 않습니다.');
```

END;
/



사용자 정보 수정 후 결과 확인

- BeforeUpdateStudent 트리거 실행 결과 확인
 - 패스워드의 길이를 4자리 미만으로 하여 **student** 수정
 - “ORA-20002: 암호는 4자리 이상이어야 합니다” 에러 발생
 - **student** 테이블은 수정되지 않음
 - 패스워드에 공란이 포함되도록 하여 **student** 수정
 - “ORA-20003: 암호에 공란은 입력되지 않습니다.” 에러 발생
 - **student** 테이블은 수정되지 않음



사용자 정보 수정 후 결과

```
SQL> UPDATE student SET s_pwd = '12' WHERE s_id='20011234';  
UPDATE student SET s_pwd = '12' WHERE s_id='20011234'
```

*

ERROR at line 1:

ORA-20002: 암호는 4자리 이상이어야 합니다

ORA-06512: at "DB.BEFOREUPDATESTUDENT", line 23

ORA-04088: error during execution of trigger
'DB.BEFOREUPDATESTUDENT'

```
SQL> UPDATE student SET s_pwd = '13 45' WHERE s_id='20011234';  
UPDATE student SET s_pwd = '13 45' WHERE s_id='20011234'
```

*

ERROR at line 1:

ORA-20003: 암호에 공란은 입력되지 않습니다.

ORA-06512: at "DB.BEFOREUPDATESTUDENT", line 25

ORA-04088: error during execution of trigger '
DB.BEFOREUPDATESTUDENT'