DATABASE WITH PL/SQL

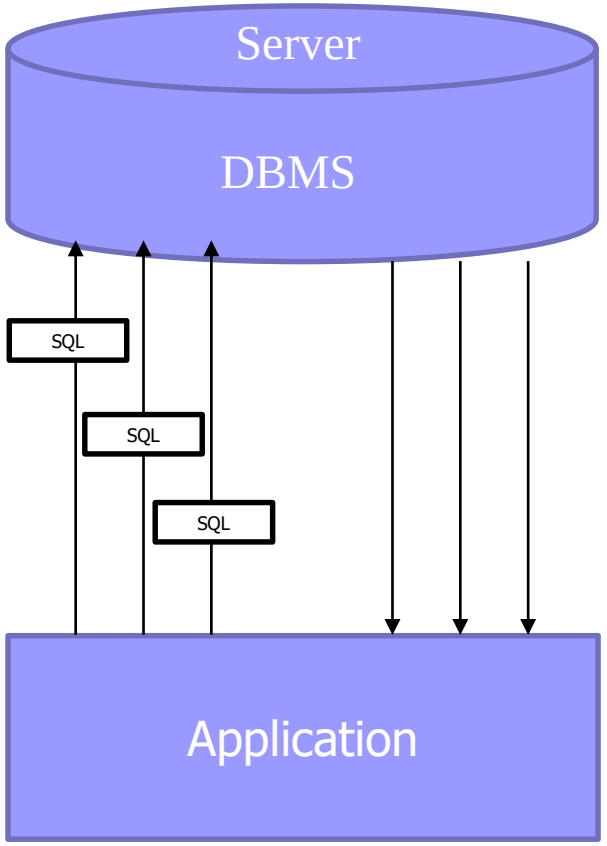
SQL언어의 장점과 단점

SQL의 장점

- 사용자가 이해하기 쉬운 구성
- 쉽게 배울 수 있다.
- 복잡한 로직을 간단하게 작성 가능
- **ANSI**에 의해 문법이 표준화

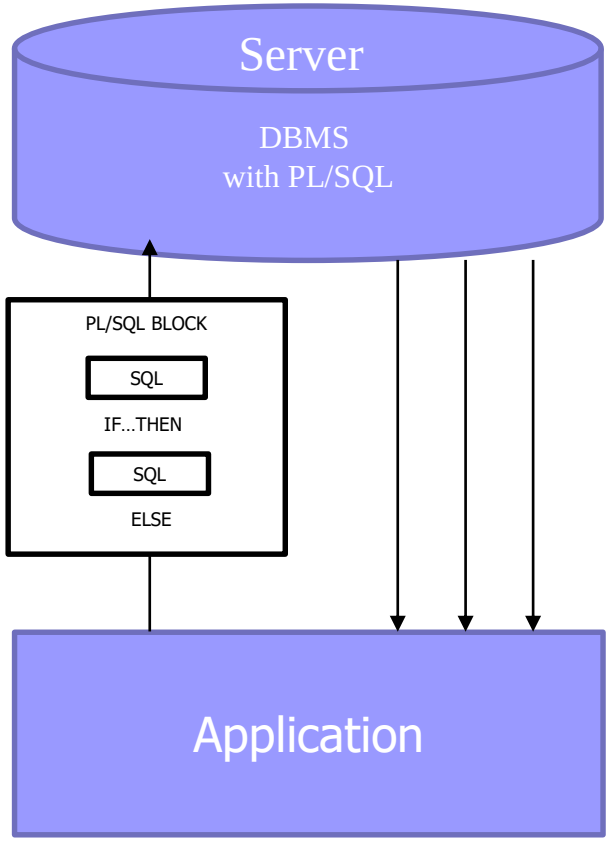
SQL의 단점

- 반복 처리를 할 수 없다**(LOOP)**
- 비교 처리를 할 수 없다**(IF)**
- **Error** 처리를 할 수 없다**(**예외 처리**)**
- 변수 선언을 할 수 없다.
- 네트워크 트레픽을 유발한다.



PL/SQL의 생성과 실행

- 반복 처리를 할 수 있다(**LOOP**)
- 비교 처리를 할 수 있다(**IF**)
- **Error** 처리를 할 수 있다(예외 처리)
- 변수 선언을 할 수 있다.
- 네트워크 트레픽이 감소한다.

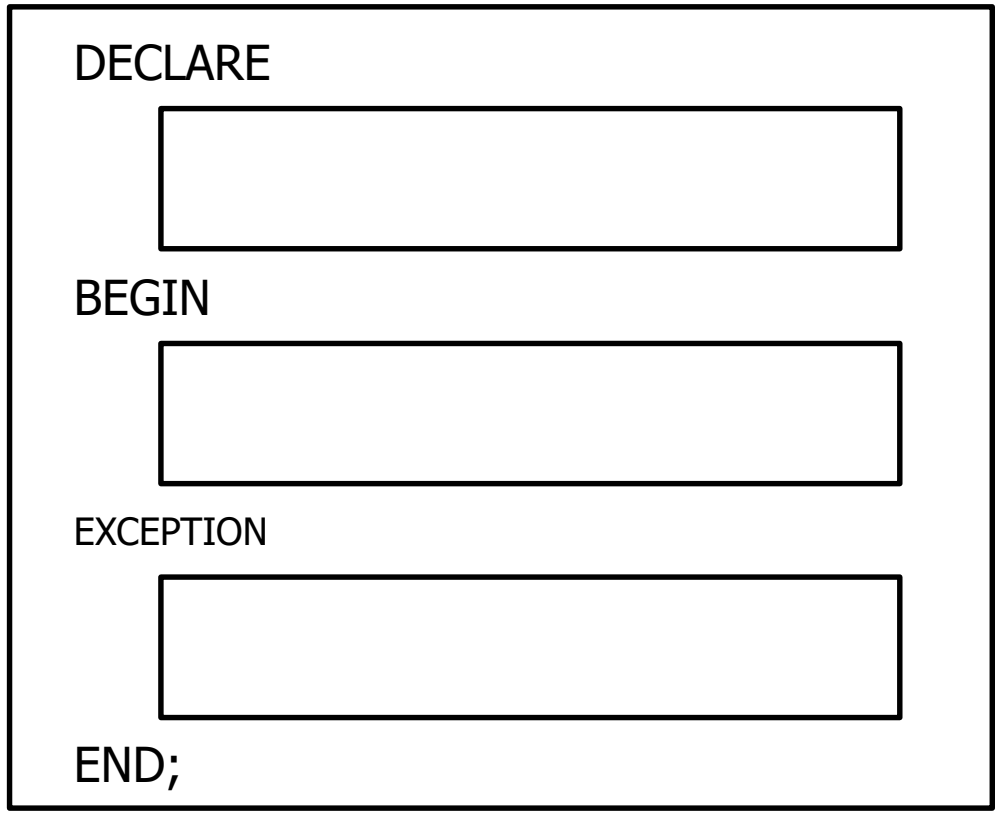


BUILD A PL/SQL BLOCK

PL/SQL의 종류

- **Anonymous Procedure** – 반복적으로 실행하려는 **SQL**문을 필요할 때마다 작성하여 실행하는 방법, 데이터베이스에 저장되지 않기 때문에 이름 없는 또는 제목 없는 **PL/SQL**이라고 불림
- **Stored Procedure** – 데이터베이스 내에 정보가 저장됨, 실행하려는 로직을 처리하고 끝남
- **Stored Function** – **Stored Procedure**와 동일한 개념, 동일한 기능을 가지고 있음. 결과를 반환해줌
- **Package** – **PL/SQL** 블록을 관리하기 위해 사용됨
- **Trigger** – **PL/SQL** 종류 중 가장 다양한 기능을 가지고 있음. **EX) UPDATE** 문을 실행하면 그 작업을 실행시킨 후 또는 실행 전 **TRIGGER**에 의해 로직을 실행
- **Object-Type** – 객체 옵션이 제공되는 데이터베이스를 객체관계형 데이터베이스라고한다. 이러한 객체에 대해 데이터를 입력, 수정, 삭제, 조회하기 위해서는 반드시 **PL/SQL** 언어를 사용한다.

BUILD A PL/SQL BLOCK



- 모든 **PL/SQL** 프로그램 구성체의 구조를 기본 **PL/SQL** 블록에 기초한다.

Section	Description	Inclusion
Declaration Section 선언부	변수, 상수를 선언	의무적
Executable Section 실행부	데이터 조작 SQL 비교문, 제어문	의무적
Exception Handling 예외 처리부	예외 처리	선택적

BUILD A PL/SQL BLOCK

SCLAR 변수

변수명 [CONSTANT] [data_type] [NOT NULL] [:= DEFAULT [표현식]];

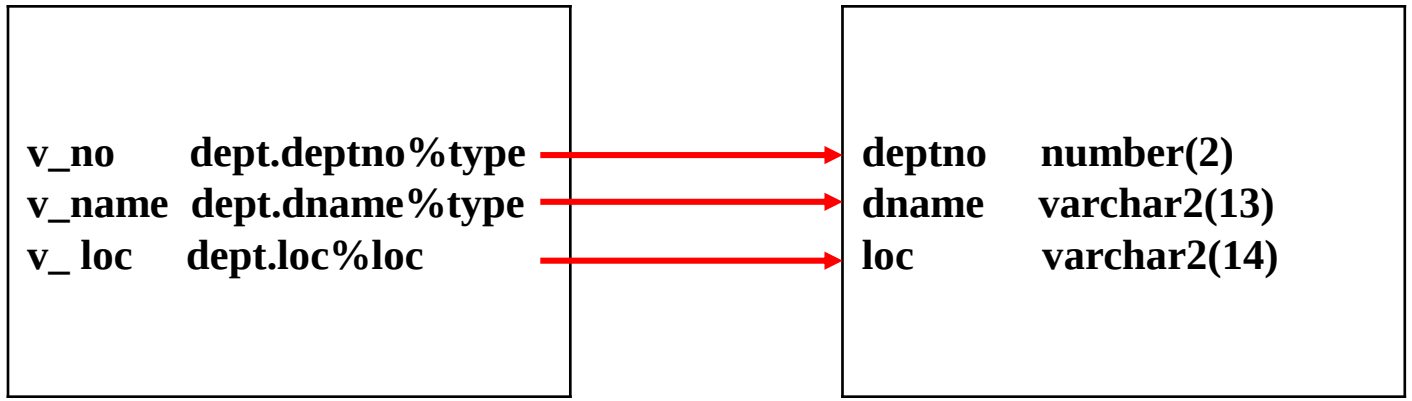
NUMBER	정수, 소수점을 포함한 숫자
BINARY_INTEGER	-2147483647 ~ +2147483647 사이의 정수
CHAR	고정길이 문자
VARCHAR2	가변길이 문자
LONG	대용량 고정 길이(2GB)
LONG RAW	대용량 이진 문자(2GB)
DATE	날짜와 시간
BOOLEAN	참과 거짓(T, F)
RAW	Binary 데이터 최대 32767Bytes

```
v_no        NUMBER
v_name      VARCHAR2(15) DEFAULT ‘unknown’
v_loc       CONSTANT NUMBER(7,2) := 0.2
```

BUILD A PL/SQL BLOCK

TYPE 변수

변수명 TABLE명.COLUMN%TYPE



BUILD A PL/SQL BLOCK

- **PL/SQL 블록 작성 지침**
  - 블록 내 **SQL**문을 여러 번 작성할 수 있다.
  - 식별자는 최대 **30**문자로 작성
  - 식별자는 테이블 또는 칼럼 명과 동일할 수 없다.
  - 식별자는 알파켓 문자로 시작해야 한다.
  - 주석은 단일 라인인 경우 **2**개의 대쉬**(--)**, 여러 라인인 경우 **/* ~ */**

STORED PROCEDURE

- **PL/SQL** 언어 중 대표적인 구조
- 개발자가 자주 실행해야 하는 업무를 이 문법에 의해 미리 작성
- 필요할 때마다 호출하여 실행 가능

Syntax

```
Create [or Replace] procedure
( argument1 [mode] data_type1,
….
IS [AS]
BEGIN
    …..
EXCEPTION
    …..
END;

DROP procedure;
```

where	*procedure*	프로시저 명
	argument	변수
	mode	IN, OUT, IN OUT
	data_type	argument 변수의 데이터 타입
	BEGIN ~ END	실행하려는 처리 절차

STORED PROCEDURE

- **PL/SQL에서 SQL문 사용**

SELECT column **INTO** variable FROM table_name WHERE condition;
INSERT INTO table_name VALUES(value1, value2….)
UPDATE table_name SET [column] = value WHERE condition;
DELETE FROM table_name WHERE condition

STORED PROCEDURE

1. O/S 편집기로 파일 생성

```
vi salary_cal.sql
CREATE PROCEDURE SALARY_CAL
BEGIN
        SELECT * INTO :A :B
        FROM EMP
        WHERE ID =  10;
        .........
END;
```

2. 데이터베이스 접속

tbsql sys/tibero

SQL> @salary_cal.sql

3. Compiler 생성

문법에 오류가 없으면 **Procedure Create**라는 메시지와 함께 프로시저 생성

4. SQL> EXECUTE salary_cal

EXECUTE 명령어를 사용해 프로시저 호출

STORED PROCEDURE

EXAMPLE

데이터베이스에 접속한 사용자를 기록하는 프로시저

로그 정보를 저장하기 위한 테이블 생성

```
tbsql sys/tibero
SQL> CREATE TABLE LOG_TABLE
        (USERID VARCHAR2(10),
         LOG_DATE DATE);
```

```
$ vi log_execution.sql
CREATE OR REPLACE PROCEDURE LOG_EXECUTION
IS
BEGIN
INSERT INTO LOG_TABLE (USERID, LOG_DATE)
VALUES (USER, SYSDATE);
END LOG_EXECUTION;
```

STORED PROCEDURE

PROCEDURE 생성.

```
SQL> @log_execution.sql
Procedure 'LOG_EXECUTION' created.

SQL> SHOW ERRORS
No Errors.

SQL> execute log_execution

PSM completed.
```

로그 정보가 기록된 테이블 조회

```
SQL> select * from log_table;

USERID     LOG_DATE
---------- ------------------------------------------------------------
SYS        2022/10/12
```

STORED PROCEDURE

EXAMPLE

사원 삭제하기

로그 정보를 저장하기 위한 테이블 생성

```
$ vi fire_emp.sql
CREATE OR REPLACE PROCEDURE FIRE_EMP
(V_EMP_NO     IN     S_EMP.EMPNO%TYPE)
IS
BEGIN
    DELETE FROM S_EMP
    WHERE EMPNO = V_EMP_NO;

END FIRE_EMP;
/
```

STORED PROCEDURE

PROCEDURE 생성.

```
SQL> @fire_emp.sql

Procedure 'FIRE_EMP' created.

SQL> select ename, empno
    from s_emp
    where empno = 7654;

ENAME          EMPNO
---------- ----------
MARTIN          7654

1 row selected.
```

```
SQL> execute fire_emp(7654)

PSM completed.

SQL> select ename, empno
    from s_emp
    where empno = 7654;
0 row selected.
```

STORED PROCEDURE

매개 변수의 종류

SQL>VARIABLE C NUMBER;

SQL>EXECUTE TEST(1234, 100, :C);

SQL> PRINT C;

```
CREATE PROCEDURE TEST
(a     IN        number,
 b     INOUT     number,
 c     OUT       number)

BEGIN
    C := 1234;


EXCEPTION
.....


END;
```

STORED PROCEDURE

EXAMPLE

사원번호를 입력하고 사원에 대한 정보를 검색하는 프로시저

```
$ vi

CREATE OR REPLACE PROCEDURE query_emp
( v_emp_no      IN      s_emp.empno%TYPE,
v_emp_name      OUT     s_emp.ename%TYPE,
v_emp_sal       OUT     s_emp.sal%TYPE,
v_emp_comm      OUT     s_emp.comm%TYPE)
IS
BEGIN
        SELECT ename, sal, comm
        INTO v_emp_name, v_emp_sal, v_emp_comm
        FROM s_emp
        WHERE empno = v_emp_no;
END query_emp;
/
```

STORED PROCEDURE

EXAMPLE

사원번호를 입력하고 사원에 대한 정보를 검색하는 프로시저

```
SQL> @query_emp.sql

Procedure 'QUERY_EMP' created.

SQL> VARIABLE emp_name varchar2(15)
SQL> VARIABLE emp_sal number
SQL> VARIABLE emp_comm number

SQL> execute query_emp(7900,:emp_name, :emp_sal, :emp_comm)

PSM completed.

SQL> print emp_name

EMP_NAME
------------------------------
JAMES
```

비교문

- **IF문은 조건 제어문을 사용하여 문장의 논리적 흐름을 변경할 때 사용**
- **1) IF ~ THEN ~END IF 문**
  - **IF절에 정의된조건이 TRUE인 경우에만 처리 문장을 실행**
  - 조건을 만족하지 않는다면, 문장은 실행되지 않으며 **END IF** 절을 생략하면 에러가 발생
  - SET SERVEROUTPUT ON로 결과물 확인

```
SQL>SET SERVEROUTPUT ON
SQL>DECLARE
V_CONDITION NUMBER :=1;
BEGIN
 IF V_CONDITION = 1 THEN
DBMS_OUTPUT.PUT_LINE('데이터 값은 1입니다!!');
END IF;
END;
/
```

비교문

- **2) IF ~ THEN ~ELSE ~END IF 문**
  - IF절에 정의된조건이 TRUE인 경우 처리 문장 1을 실행
  - 조건을 만족하지 않으면 처리 문장2를 실행
  - ELSE절은 조건문에서 단 한번만 정의 할 수 있다.
  - IF문 안에 또 다른 IF문을 중첩적으로 포함할 수는 있다.

```
SQL>SET SERVEROUTPUT ON
SQL>DECLARE
V_CONDITION NUMBER :=2;
BEGIN
 IF V_CONDITION = 1 THEN
DBMS_OUTPUT.PUT_LINE('데이터 값은 1입니다!!');
ELSE
DBMS_OUTPUT.PUT_LINE('데이터 값은 1이 아닙니다!!');
END IF;
END;
/
```

비교문

- **3) IF ~ THEN ~ ELSIF ~ ELSE ~ END IF 문**
  - **IF**절에 여러 개의 조건절을 포함할 수 있는 방법
  - **ELSIF** 절을 여러 번 정의할 수 있으며, 조건에 만족하지 않으면 **ELSE**절에 의해 처리

```
SQL>SET SERVEROUTPUT ON
SQL>DECLARE
V_CONDITION NUMBER :=2;
BEGIN
 IF V_CONDITION > 1 THEN
DBMS_OUTPUT.PUT_LINE('데이터 값은 1보다 큽니다!!');
ELSIF V_CONDITION = 1 THEN
DBMS_OUTPUT.PUT_LINE('데이터 값은 1입니다!!');
ELSE
DBMS_OUTPUT.PUT_LINE('데이터 값은 1보다 작습니다!!');
END IF;
END;
/
```

반복문

- 일련의 SQL문을 반복적으로 여러 번 실행할 때 사용
- 문법에 의해 설정된 반복 횟수만큼 반복적으로 SQL 실행
- 1) LOOP 문
  - LOOP ~ END LOOP문 안에 정의된 SQL문이 반복적으로 실행 되다가 EXIT WHEN절에 만족되는 조건을 만나면 반복 작업 중단
  - EXIT절이 없으면 무한루프가 될 수 있다.

```
SQL>SET SERVEROUTPUT ON
SQL>DECLARE
        CNT       NUMBER := 0;
    BEGIN
        LOOP
        CNT :=  CNT + 1;
        DBMS_OUTPUT.PUT_LINE(CNT);
        EXIT WHEN CNT = 10;
    END LOOP;
    END;
    /
```

반복문

- **2) FOR ~ LOOP 문**
  - **FOR~ LOOP**문에 의해 반복 실행되는 횟수를 정확히 아는 경우 사용할 수 있다.
  - 정의된 변수가 최솟 값과 최댓값 범위 내에서 반복적으로 실행
  - 변수가 최댓값을 만나는 순간 반복 작업은 종료

```
SQL>SET SERVEROUTPUT ON
SQL>DECLARE
        I           NUMBER;
    BEGIN
        FOR I IN 1..10 LOOP
                IF (MOD (I, 2) = 1) THEN
                DBMS_OUTPUT.PUT_LINE(I);
                END IF;
        END LOOP;
    END;
    /
```

반복문

- **3) WHILE ~ END LOOP문**
  - **WHILE ~ END LOOP** 문은 **WHILE** 절에 정의된 조건이 만족할 때까지 **SQL**문을 반복 실행
  - 조건이 참일 경우 작업 중단

```
SQL>SET SERVEROUTPUT ON
SQL>DECLARE
            V_CNT       NUMBER :=1 ;
            V_STR       VARCHAR2(10) := NULL;
      BEGIN
            WHILE V_CNT <= 10 LOOP
                        V_STR := V_STR || '#';
                         DBMS_OUTPUT.PUT_LINE(V_STR);
                        V_CNT := V_CNT + 1;
             END LOOP;
      END;
      /
```

STORED FUNCTION

- **STORED PROCEDURE**와 동일한 개념, 기능을 가지고 있다.
- **STORED PROCEDURE** 은 로직을 처리하고 끝난다.
- **STORED FUNCTION**은 그 처리 결과를 사용자에게 돌려주는 기능이 있다.

Syntax

```
Create [or Replace] function
( argument1 [mode] data_type1,
….
   Return data_type;
IS [AS]
BEGIN
     …..
     Return 변수
EXCEPTION
     …..
END;

DROP Function;
```

where	*function*	함수 명
	argument	변수
	mode	IN, OUT, IN OUT
	data_type	argument 변수의 데이터 타입
	BEGIN ~ END	실행하려는 처리 절차

STORED FUNCTION

1. O/S 편집기로 파일 생성

```
vi test_check.sql
CREATE Function test_check
 ( v_test       number;)
   Return number;
BEGIN
        v_chk := v_test * 0.01;
        return (v_chk)
END;
```

2. 데이터베이스 접속

tbsql sys/tibero

SQL> @test_check.sql

3. Compiler 생성

문법에 오류가 없으면 **Function Create**라는 메시지와 함께 함수 생성

4. SQL> VARIABLE v_test number

SQL> EXECUTE :test_check := test_check(7900)

EXECUTE 명령어를 사용해 함수 호출

STORED FUNCTION

EXAMPLE

입력 값에 세금을 매겨주는 함수 생성

```
$ vi tax.sql
CREATE OR REPLACE FUNCTION tax
(v_value       IN      NUMBER)

      RETURN  NUMBER
IS
BEGIN
      RETURN (v_value * 0.07);
END tax;
/
```

STORED FUNCTION

EXAMPLE

```
$ tbsql sys/tibero
SQL> @tax.sql

Function 'TAX' created.

SQL> variable x number;
SQL> EXECUTE :x := tax(100);

PSM completed.

SQL> print x

         X
----------
         7
```

STORED FUNCTION

EXAMPLE

DML문장에서도 실행 가능 (UPDATE, INSERT, SELECT)

```
SELECT sal, tax(sal)
FROM s_emp
WHERE empno = 7900;

 SAL   TAX(SAL)
---------- ----------
      950       66.5
```

CURSOR AND EXCEPTION

- **CURSOR는 실행한 SQL문의 단위를 의미**
- **DBMS는 모든 문장을 CURSOR 단위로 처리하고, 그 정보를 저장 관리한다.**
- **1) IMPLICT CURSOR(암시적 커서)**
  - 한 번 실행에 하나의 결과를 리턴하는 SQL 문

```
SELECT empno, ename
INTO :v_no, :v_ename
FROM s_emp
WHERE deptno = 10;
```

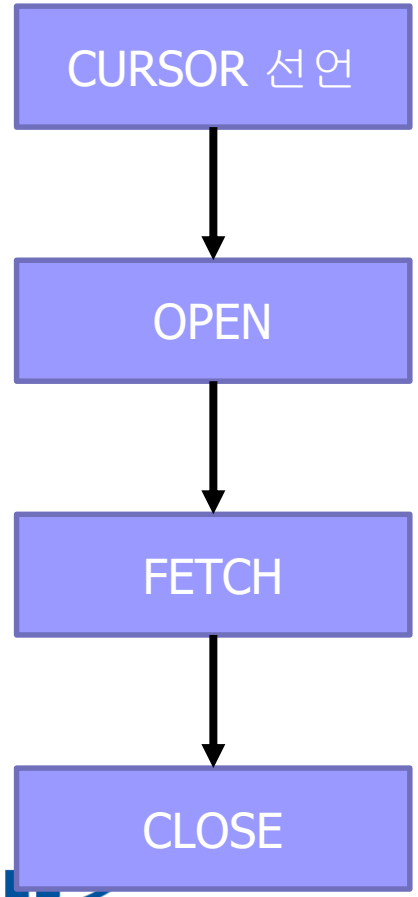
EMPNO	ENAME	DEPTNO
7369	SMITH	20
7499	ALLEN	30
7521	WARD	30
7566	JONES	20
7698	BLAKE	30
7788	SCOTT	20
7839	KING	10
7844	TURNER	30
7876	ADAMS	20
7900	JAMES	30
7902	FORD	20

- **2) EXPLICT CURSOR(명시적 커서)**
  - SQL문을 실행시켰을 때 결과가 여러 개인 경우
  - 결과가 여러 개인데 암시적 커서를 사용했을 경우 에러 발생
  - 암시적 커서에 사용되는 스칼라 변수는 한 번에 하나의 값만 저장

```
Cusor C1 IS
SELECT empno, ename
FROM s_emp
WHERE deptno = 20;
OPEN C1;
LOOP
 FETCH C1 INTO v_no, v_ename
END LOOP;
CLOSE C1;
```

CURSOR AND EXCEPTION

- 명시적 커서는 기본적으로 **CURSOR ~ OPEN ~ FETCH ~CLOSE** 문법으로 구성



CURSOR cursor_name IS [SELECT …];	실행하고자 하는 SELECT문 작성
OPEN cursor_name	CURSOR 선언에서 선언된 SELECT문 실행을 의미
LOOP FETCH cursor_name INTO 변수; EXIT WHEN [처리 내용]; END LOOP;	OPEN된 SELECT문에 의해 검색된 하나의 행 정보를 읽어오고, 결과가 여러 개 인 경우 반복문 실행
CLOSE cursor_name	선언된 SELECT 해제

CURSOR AND EXCEPTION

EXAMPLE

해당 부서에 근무하는 모든 사원의 정보를 조회하는 프로시저

```
CREATE OR REPLACE PROCEDURE emp_process
IS
      v_empno       s_emp.empno%TYPE;
      v_ename       s_emp.ename%TYPE;
      v_sal         NUMBER(7,2);
      CURSOR emp_cursor IS
      SELECT empno, ename, sal
      FROM s_emp
      WHERE deptno = 20;
BEGIN
      OPEN emp_cursor;
      LOOP
FETCH emp_cursor INTO v_empno, v_ename, v_sal;

      EXIT WHEN emp_cursor%ROWCOUNT > 3 OR
      emp_cursor%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE(v_empno||' '||v_ename||' '||v_sal);

      END LOOP;
      CLOSE emp_cursor;
END emp_process;
/
```

명시적 커서명 선언
실행 될 SELECT 문

선언된 SELECT 실행

SELECT 실행 후 하나의 행을 선언된 지역변수에 저장

리턴된 결과가 3행 이상 혹은 만족하는 행을 발견하지 못하면 반복문 종료

검색된 결과를 사용자 화면에 출력

선언된 커서 해제

CURSOR AND EXCEPTION

EXAMPLE

해당 부서에 근무하는 모든 사원의 정보를 조회하는 프로시저

종류	설명	
%ROWCOUNT	커서의 현재 Row Count 리턴	실행된 커서 문장에서 행 수를 알아야 하는 경우
%FOUND	커서가 현재 조건을 만족하는지 리턴	검색된 행이 발견되었는지 알 수 있는 속성
%NOTFOUND	커서가 현재 조건을 만족하지 않는지 리턴	검색된 행이 발견되지 않았는지 알 수 있는 속성
%ISOPEN	커서가 현재 OPEN되어 있는지 리턴	선언된 커서 OPEN 상태인지 아닌지 확인

CURSOR AND EXCEPTION

- **PL/SQL** 블록 내 **SQL**문이 정상적으로 실행되지 못할 때 에러 발생
- 에러가 발생하면 **EXCEPTION**에 의해 처리할 수 있다.
- 자주 발생하는 에러 처리는 각 데이터베이스에서 제공하며, 자주 발생하지 않는 에러는 사용자가 직접 정의 가능.

```
DECLARE
…
BEGIN
    SELECT empno INTO v_empno FROM s_emp WHERE
    deptno = 99;
…
EXCEPTION
…
END;
```

PL/SQL블록 내에 사용된 **SQL** 에러 처리는 **EXCEPTION**절에 정의하면 된다.

CURSOR AND EXCEPTION

- 에러를 처리해주는 방법은 4가지 방법이 있다.
- 1) 미리 정의된 에러 처리 방법
  - 자주 발생하기 쉬운 에러를 시스템에 미리 정의하는 것을 '시스템 정의 예외'라고 한다.

시스템 정의 예외는 다음과 같은 예외 상황이 존재한다.

예외 상황	설명
CASE_NOT_FOUND	CASE 문의 WHEN 절 중에서 조건을 만족하는 것이 없고 ELSE 절도 없는 경우이다.
CURSOR_ALREADY_OPEN	이미 열려 있는 커서를 또 다시 여는 경우이다.
DUP_VAL_ON_INDEX	유일 키(UNIQUE) 제약조건이 선언되어 있는 컬럼에 중복된 값을 삽입하려는 경우이다.
INVALID_CURSOR	열려 있지 않은 커서를 닫는 경우이다.
NO_DATA_FOUND	SELECT INTO에 의한 질의에서 결과 로우가 하나도 없는 경우이다.
TOO_MANY_ROWS	SELECT INTO에 의한 질의에서 결과 로우가 둘 이상인 경우이다.
VALUE_ERROR	데이터 값의 변환(conversion), 절단(truncation), 제약조건 등과 관련된 에러가 발생한 경우이다.
ZERO_DIVIDE	0으로 나눗셈 연산을 수행하는 경우이다.

예외 상황	설명
COLLECTION_IS_NULL	초기화되지 않은 컬렉션 변수의 요소에 값을 대입하려는 경우이다. 초기화되지 않은 컬렉션 변수에 EXISTS를 제외한 서브 프로그램을 사용하는 경우이다.
SUBSCRIPT_BEYOND_COUNT	컬렉션 변수에 있는 요소의 개수보다 큰 인덱스를 사용하는 경우이다.
SUBSCRIPT_OUTSIDE_LIMIT	컬렉션 변수를 접근하는 인덱스가 유효하지 않은 경우이다. (예: -1)
ROWTYPE_MISMATCH	커서 변수의 타입과 PSM 내부에서 사용한 커서 변수의 타입이 맞지 않은 경우이다.
INVALID_NUMBER	주어진 문자열이 number로 적절한 문자열 형태가 아닌 경우이다.

출처 : Tibero tbPSM 안내서

CURSOR AND EXCEPTION

EXAMPLE

급여를 입력 후 조건에 맞는 사원을 출력하는 프로시저

```
CREATE OR REPLACE PROCEDURE test
(v_sal        IN     s_emp.sal%TPYE)
IS
      v_ename        s_emp.ename%TYPE;
BEGIN
      SELECT ENAME
      INTO v_ename
      FROM S_EMP
      WHERE SAL = v_sal;
      DBMS_OUTPUT.PUT_LINE('해당 사원은 '|| v_ename|| '입니다.');

      EXCEPTION
            WHEN NO_DATA_FOUND      THEN
                  RAISE_APPLICATION_ERROR(-20002, 'DATA NOT FOUND');
            WHEN TOO_MANY_ROWS      THEN
                  RAISE_APPLICATION_ERROR(-20003, 'TOO MANY ROWS');
            WHEN OTHERS     THEN
                  RAISE_APPLICATION_ERROR(-20004, 'OTHERS ERROR');

END;
/
```

CURSOR AND EXCEPTION

- **2) 미리 정의되지 않은 에러 처리 방법**
  - 개발자는 미리 정의된 에러 처리 방법 이외의 에러에 대해 직접 에러 처리를 할 수 있다.
  - 이러한 방법을 미리 정의되지 않은 에러 처리 방법이라고 한다.

S_EMP 테이블의 **deptno** 칼럼은 **FK** 칼럼이고, **S_DEPT** 테이블의 **deptno**는 **PK**다. 이런 경우 **S_DEPT** 테이블에 존재하지 않는 부서를 **S_EMP** 테이블의 **deptno** 칼럼에 입력하면 에러가 발생한다.

```
CREATE OR REPLACE PROCEDURE hire_emp
( v_emp_name    IN      s_emp.ename%TYPE,
 v_emp_job    IN      s_emp.job%TYPE,
 v_mgr_no    IN      s_emp.mgr%TYPE,
 v_emp_hiredate    IN      s_emp.hiredate%TYPE,
 v_emp_sal    IN      s_emp.sal%TYPE,
 v_emp_comm    IN      s_emp.comm%TYPE,
 v_dept_no    IN      s_emp.deptno%TYPE)
IS
e_invalid_mgr   EXCEPTION;

PRAGMA EXCEPTION_INIT(e_invalid_mgr, -10008);
BEGIN
      INSERT INTO S_EMP (empno, ename, job, mgr, hiredate, sal, comm, deptno)
      VALUES(emp_id.NEXTVAL, v_emp_name, v_emp_job, v_mgr_no, v_emp_hiredate,
            v_emp_sal, v_emp_comm, v_dept_no);
      COMMIT WORK;
      EXCEPTION
            WHEN e_invalid_mgr THEN
                  RAISE_APPLICATION_ERROR(-20201, 'DEPTNO IS NOT A VALID DEPARTMENT.');
END hire_emp;
/
```

CURSOR AND EXCEPTION

```
SQL>  @test2.sql

Procedure 'HIRE_EMP' created.

SQL>  execute hire_emp('STONE','CLERK',9000,SYSDATE,950,300,44);
TBR-20201: DEPTNO IS NOT A VALID DEPARTMENT..
TBR-15163: Unhandled exception at SYS.HIRE_EMP,  line 21.
TBR-15163: Unhandled exception at  line 1.
```

CURSOR AND EXCEPTION

- **3)** 사용자가 정의하는 에러 처리 방법
  - 미리 정의된 에러 처리 방법과 미리 정의되지 않은 에러 처리 방법은 모두 **DB** 서버에서 발생하는 에러를 처리하는 방법
  - 서버에러라고 부름.
  - 사용자가 정의하는 에러 처리 방법은 서버 에러 처리 방법과는 달리 개발자가 미리 에러에 대한 정의를 한 다음 임의로 에러를 발생 시키는 방법
  - **EXCEPTION** 키워드에 의해 에러 조건명이 정의되고 **[RAISE]** 명령어에 의해 에러가 발생되며 **EXCEPTION**절에 의해 에러가 처리 된다.

S_EMP 테이블로부터 어떤 조건을 만족하는 행의 **COUNT**를 계산하여 **0** 값이면 **'There is are no employee salary'** 메시지를 출력하고, **0**값이 아니면 **'There is a rows employee'** 메시지를 출력

CURSOR AND EXCEPTION

```
CREATE OR REPLACE PROCEDURE TEST3
( v_sal IN      s_emp.sal%TYPE)
IS
      v_low_sal       s_emp.sal%TYPE:= v_sal - 100;
      v_high_sal      s_emp.sal%TYPE:= v_sal + 100;
      v_no_emp        NUMBER(7);

      e_no_emp_returned       EXCEPTION;
      e_more_than_one_emp     EXCEPTION;

BEGIN
      SELECT count(ename)
      INTO v_no_emp
      FROM s_emp
      WHERE sal BETWEEN (v_sal - 100) AND (v_sal + 100);
      IF v_no_emp = 0 then
            RAISE e_no_emp_returned;
      ELSIF v_no_emp > 0 then
            RAISE e_more_than_one_emp;
      END IF;

      EXCEPTION
            WHEN e_no_emp_returned THEN
                  DBMS_OUTPUT.PUT_LINE('There is are no employee salary');
            WHEN e_more_than_one_emp THEN
                  DBMS_OUTPUT.PUT_LINE('There is a rows employee');
            WHEN others THEN
                  DBMS_OUTPUT.PUT_LINE('Som other error occurred');
END;
/
```

CURSOR AND EXCEPTION

```
SQL> @test3.sql

Procedure 'TEST3' created.

SQL> EXECUTE TEST3(9000)
There is are no employee salary

SQL> EXECUTE TEST3(3000)
There is a rows employee

PSM completed.
```

CURSOR AND EXCEPTION

- **4) 예외 트래핑 함수**
  - 사용자가 실행한 **SQL**문이 실행될 때 어떤 에러 코드와 에러 메시지가 발생하는지를 개발자가 직접 참조하여 처리하는 방법
  - **SQL**문을 실행한 후 **SQLCODE** 함수를 참조해 보면 **SQL** 문의 실행 결과를 알 수 있다.
  - **SQLCODE** 값이 **0** 이면 에러 없이 정상적으로 실행
  - **1**이면 사용자가 정의한 에러가 발생,
  - **100**이면 **NO_DATA_FOUND**
  - **SQLERRM** 함수에는 **SQLCODE**에 대한 메시지가 저장되어 있다.

예외 상황	SQLCODE	SQLERRM
에러가 발생하지 않은 경우	0	NORMAL, SUCCESSFUL COMPLETION
사용자 정의 에러	1	user-defined exception
NO_DATA_FOUND	100	no data found
기타 예외 상황	해당 에러 코드	해당 에러 메시지

CURSOR AND EXCEPTION

```
CREATE OR REPLACE PROCEDURE TEST4
(v_sal  IN      s_emp.sal%TYPE)
IS

v_ename         s_emp.ename%TYPE;
v_err_code      NUMBER;
v_err_message   VARCHAR2(255);

BEGIN
      SELECT ename
      INTO   v_ename
      FROM   s_emp
      WHERE  sal = v_sal;
      DBMS_OUTPUT.PUT_LINE('해당 사원은 ' || v_ename|| '입니다');

EXCEPTION
      WHEN NO_DATA_FOUND THEN
            v_err_code := SQLCODE;
            v_err_message := SQLERRM;
            DBMS_OUTPUT.PUT_LINE (v_err_code || ' ' || v_err_message);

      WHEN TOO_MANY_ROWS THEN
            RAISE_APPLICATION_ERROR(-20003, 'TOO MANY ROWS');

      WHEN OTHERS THEN
            RAISE_APPLICATION_ERROR(-20004, 'Others Error');
END;
/
```

CURSOR AND EXCEPTION

```
SQL> @test4.sql

Procedure 'TEST4' created.

SQL> EXECUTE TEST4(9000)
100 TBR-15104: No matching data found.

PSM completed.

SQL> EXECUTE TEST4(3000)
TBR-20003: TOO MANY ROWS.
TBR-15163: Unhandled exception at SYS.TEST4,  line 23.
TBR-15163: Unhandled exception at  line 1.
```