

저장 서브프로그램

지금까지 살펴본 PL/SQL 블록은 한번 작성하여 바로 실행하는 방식을 사용했습니다. 이 장에서는 이름을 지정하여 오라클에 저장한 후 필요할 때마다 실행할 수 있는 PL/SQL 저장 서브프로그램에 대해 알아보겠습니다.

19-1 저장 서브프로그램

19-2 프로시저

19-3 함수

19-4 패키지

19-5 트리거

이 장에서 꼭 익혀야 할 것

- 프로시저 작성 및 사용 방법
- 함수 작성 및 사용 방법

19-1 저장 서브프로그램

저장 서브프로그램이란?

앞 장에서 다룬 PL/SQL 블록은 작성한 내용을 단 한 번 실행하는 데 사용했습니다. 이런 블록을 이름이 정해져 있지 않은 PL/SQL 블록이라는 의미로 익명 블록(anonymous block)이라고 부릅니다. 익명 블록은 오라클에 저장되지 않기 때문에 한 번 실행한 뒤에 다시 실행하려면 PL/SQL 블록을 다시 작성하여 실행해야 합니다.

☞ 이렇게 매번 내용을 작성하는 것은 불편하기 때문에 실행할 내용을 따로 파일(.sql)에 저장하여 실행하기도 하지만 오라클에 저장되는 것은 아니죠.

그런데 PL/SQL로 만든 프로그램을 주기적으로 또는 필요할 때마다 여러 번 사용해야 하는 상황이 빈번히 발생합니다. 이럴 경우에 PL/SQL 프로그램을 오라클에 저장해 두면 필요할 때마다 수행할 내용을 작성하지 않고 실행할 수 있으므로 여러모로 편리합니다. 이렇게 여러 번 사용할 목적으로 이름을 지정하여 오라클에 저장해 두는 PL/SQL 프로그램을 저장 서브프로그램(stored subprogram)이라고 합니다.

익명 블록과 달리 저장 서브프로그램은 오라클에 저장하여 공유할 수 있으므로 메모리·성능·재사용성 등 여러 면에서 장점이 있습니다. 다음은 익명 블록과 저장 서브프로그램의 차이점입니다.

	익명 블록	저장 서브프로그램
이름	이름 없음	이름 지정
오라클 저장	저장할 수 없음	저장함
컴파일	실행할 때마다 컴파일	저장할 때 한 번 컴파일
공유	공유할 수 없음	공유하여 사용 가능
다른 응용 프로그램에서의 호출 가능 여부	호출할 수 없음	호출 가능

오라클에서는 용도에 따라 여러 가지 방식으로 저장 서브프로그램을 구현할 수 있는데 대표적인 구현 방식은 프로시저·함수·패키지·트리거입니다. 그러면 각 유형별로 사용법을 알아볼까요?

서브프로그램	용도
저장 프로시저 (stored procedure)	일반적으로 특정 처리 작업 수행을 위한 서브프로그램으로 SQL문에서는 사용할 수 없습니다.
저장 함수 (stored function)	일반적으로 특정 연산을 거친 결과 값을 반환하는 서브프로그램으로 SQL문에서 사용할 수 있습니다.
패키지 (package)	저장 서브프로그램을 그룹화하는 데 사용합니다.
트리거 (trigger)	특정 상황(이벤트)이 발생할 때 자동으로 연달아 수행할 기능을 구현하는 데 사용합니다.

1분 복습

다음은 저장 서브프로그램에 대한 내용입니다. 내용이 참인지 거짓인지 판단해 보세요.

- 저장 서브프로그램은 오라클 내에 저장할 수 있다.
- 실행할 때마다 컴파일을 한다.
- 다른 응용 프로그램에서 호출할 수 없다.

정답 1. 참 2. 거짓 3. 거짓

19-2 프로시저

저장 프로시저(stored procedure)는 특정 처리 작업을 수행하는 데 사용하는 저장 서브프로그램으로 용도에 따라 파라미터를 사용할 수 있고 사용하지 않을 수도 있습니다.

☞ 이후 저장 프로시저는 '프로시저'로 표기하겠습니다.

파라미터를 사용하지 않는 프로시저

프로시저 생성하기

작업 수행에 별다른 입력 데이터가 필요하지 않을 경우에 파라미터를 사용하지 않는 프로시저를 사용합니다. 다음과 같이 CREATE [OR REPLACE] PROCEDURE를 사용해 만들 수 있습니다. 앞에서 다룬 익명 블록과 마찬가지로 프로시저도 선언부, 실행부, 예외 처리부로 이루어져 있음을 눈여겨보세요.

CREATE [OR REPLACE] PROCEDURE 프로시저 이름
IS | AS -③ ① ②
선언부
BEGIN
 실행부
EXCEPTION -④
 예외 처리부
END [프로시저 이름]; -⑤

기본 형식

번호	설명
①	지정한 프로시저 이름을 가진 프로시저가 이미 존재하는 경우에 현재 작성한 내용으로 대체합니다. 즉 덮어 쓴다는 뜻이며 생략 가능한 옵션입니다.
②	저장할 프로시저의 고유 이름을 지정합니다. 같은 스키마 내에서 중복될 수 없습니다.
③	선언부를 시작하기 위해 IS 또는 AS 키워드를 사용합니다. 선언부가 존재하지 않더라도 반드시 명시합니다. DECLARE 키워드는 사용하지 않습니다.
④	예외 처리부는 생략 가능합니다.
⑤	프로시저 생성의 종료를 뜻하며 프로시저 이름은 생략 가능합니다.

그러면 기본 형식에 따라 SQL*PLUS에 SCOTT 계정으로 접속하여 프로시저를 생성해 보죠.

실습 19-1 프로시저 생성하기

• 완성 파일 19-1.sql

```
01 CREATE OR REPLACE PROCEDURE pro_noparam
02 IS
03   V_EMPNO NUMBER(4) := 7788;
04   V_ENAME VARCHAR2(10);
05 BEGIN
06   V_ENAME := 'SCOTT';
07   DBMS_OUTPUT.PUT_LINE('V_EMPNO : ' || V_EMPNO);
08   DBMS_OUTPUT.PUT_LINE('V_ENAME : ' || V_ENAME);
09 END;
10 /
```

:: 결과 화면

프로시저가 생성되었습니다.

SQL*PLUS로 프로시저 실행하기

생성한 프로시저는 SQL*PLUS에서 바로 사용하거나 다른 PL/SQL 블록에서 실행할 수 있습니다. SQL*PLUS에서 실행할 때 다음과 같이 EXECUTE 명령어를 사용합니다.

EXECUTE 프로시저 이름;

EXECUTE 명령어를 사용하여 만든 pro_noparam 프로시저를 실행합니다.

실습 19-2 생성한 프로시저 실행하기

• 완성 파일 19-2.sql

```
01 SET SERVEROUTPUT ON;
02 EXECUTE pro_noparam;
```

:: 결과 화면

V_EMPNO : 7788

V_ENAME : SCOTT

PL/SQL 블록에서 프로시저 실행하기

특정 PL/SQL 블록에서 이미 만들어져 있는 프로시저를 실행한다면 오른쪽과 같이 실행부에 실행할 프로시저 이름을 지정합니다.

BEGIN

 프로시저 이름;

END;

여기에서는 익명 블록에서 pro_noparam 프로시저를 실행해 보겠습니다.

실습 19-3 익명 블록에서 프로시저 실행하기

• 완성 파일 19-3.sql

```
01 BEGIN  
02     pro_noparam;  
03 END;  
04 /
```

:: 결과 화면

```
V_EMPNO : 7788  
V_ENAME : SCOTT
```

☞ 출력 결과가 나오지 않는다면 실행 전 SET SERVEROUTPUT ON 명령어의 실행을 확인해 주세요.

프로시저 내용 확인하기

이미 저장되어 있는 프로시저를 포함하여 서브프로그램의 소스 코드 내용을 확인하려면 USER_SOURCE 데이터 사전에서 조회합니다.

USER_SOURCE의 열	설명
NAME	서브프로그램(생성 객체) 이름
TYPE	서브프로그램 종류(PROCEDURE, FUNCTION 등)
LINE	서브프로그램에 작성한 줄 번호
TEXT	서브프로그램에 작성한 소스 코드

다음은 토드로 만든 pro_noparam 프로시저를 USER_SOURCE에서 조회한 결과입니다.

실습 19-4 USER_SOURCE를 통해 프로시저 확인하기(토드)

• 완성 파일 19-4.sql

```
01 SELECT *  
02     FROM USER_SOURCE  
03    WHERE NAME = 'PRO_NOPARAM';
```

:: 결과 화면

NAME	TYPE	LINE	TEXT
▶ PRO_NOPARAM	PROCEDURE	1	PROCEDURE pro_noparam
PRO_NOPARAM	PROCEDURE	2	IS
PRO_NOPARAM	PROCEDURE	3	V_EMPNO NUMBER(4) := 7788;
PRO_NOPARAM	PROCEDURE	4	V_ENAME VARCHAR2(10);
PRO_NOPARAM	PROCEDURE	5	BEGIN
PRO_NOPARAM	PROCEDURE	6	V_ENAME := 'SCOTT';
PRO_NOPARAM	PROCEDURE	7	DBMS_OUTPUT.PUT_LINE('V_EMPNO : ' V_EMPNO);
PRO_NOPARAM	PROCEDURE	8	DBMS_OUTPUT.PUT_LINE('V_ENAME : ' V_ENAME);
PRO_NOPARAM	PROCEDURE	9	END;

SQL*PLUS에서 사용한다면 TEXT 열을 조회해 주면 됩니다.

실습 19-5 USER_SOURCE를 통해 프로시저 확인하기(SQL*PLUS)

· 완성 파일 19-5.sql

```
01  SELECT TEXT  
02    FROM USER_SOURCE  
03   WHERE NAME = 'PRO_NOPARAM';
```

:: 결과 화면

```
TEXT
```

```
-----  
PROCEDURE pro_noparam  
IS  
  V_EMPNO NUMBER(4) := 7788;  
  V_ENAME VARCHAR2(10);  
BEGIN  
  V_ENAME := 'SCOTT';  
  DBMS_OUTPUT.PUT_LINE('V_EMPNO : ' || V_EMPNO);  
  DBMS_OUTPUT.PUT_LINE('VENAME : ' || VENAME);  
END;
```

9개의 행이 선택되었습니다.

프로시저 삭제하기

DROP PROCEDURE 명령어로 프로시저를 삭제할 수 있습니다.

실습 19-6 프로시저 삭제하기

· 완성 파일 19-6.sql

```
01  DROP PROCEDURE PRO_NOPARAM;
```

:: 결과 화면

프로시저가 삭제되었습니다.

이미 저장되어 있는 프로시저의 소스 코드를 변경할 때에는 CREATE OR REPLACE PROCEDURE 명령어로 프로시저를 재생성합니다. ALTER PROCEDURE는 프로시저의 소스 코드 내용을 재컴파일하는 명령어이므로 작성한 코드 내용을 변경하지 않습니다.

☞ ALTER PROCEDURE 명령어의 자세한 내용은 오라클 문서(docs.oracle.com/cd/B28359_01/appdev.111/b28370/alter_procedure.htm#LNPLS99997)를 참고하세요.

파라미터를 사용하는 프로시저

프로시저를 실행하기 위해 입력 데이터가 필요한 경우에 파라미터를 정의할 수 있습니다. 파라미터는 여러 개 작성할 수 있으며 다음과 같은 형식으로 사용합니다.

기본 형식

```
CREATE [OR REPLACE] PROCEDURE 프로시저 이름
  ①          ②
  [(파라미터 이름1 [modes] 자료형 [ := | DEFAULT 기본값], -③
    파라미터 이름2 [modes] 자료형 [ := | DEFAULT 기본값],
    ...
    파라미터 이름N [modes] 자료형 [ := | DEFAULT 기본값]
  )]
  IS | AS -④
  선언부
  BEGIN
  실행부
  EXCEPTION -⑤
  예외 처리부
  END [프로시저 이름]; -⑥
```

번호	설명
①	지정한 프로시저 이름을 가진 프로시저가 이미 존재하는 경우에 현재 작성한 내용으로 대체합니다. 즉 덮어 쓴다는 뜻이며 생략 가능한 옵션입니다.
②	저장할 프로시저의 고유 이름을 지정합니다. 같은 스키마 내에서 중복될 수 없습니다.
③	실행에 필요한 파라미터를 정의합니다. 파라미터는 쉼표(,)로 구분하여 여러 개 지정할 수 있습니다. 기본값과 모드(modes)는 생략 가능합니다. 자료형은 자릿수 지정 및 NOT NULL 제약 조건 사용이 불가능합니다.
④	선언부를 시작하기 위해 IS 또는 AS 키워드를 사용합니다. 선언부가 존재하지 않더라도 반드시 명시합니다. DECLARE 키워드는 사용하지 않습니다.
⑤	예외 처리부는 생략 가능합니다.
⑥	프로시저 생성의 종료를 뜻하며 프로시저 이름은 생략 가능합니다.

파라미터를 지정할 때 사용하는 모드는 IN, OUT, IN OUT 세 가지가 있는데요. 각 모드 지정에 따른 파라미터 지정 방식을 살펴볼까요?

파라미터 모드	설명
IN	지정하지 않으면 기본값으로 프로시저를 호출할 때 값을 입력받습니다.
OUT	호출할 때 값을 반환합니다.
IN OUT	호출할 때 값을 입력받은 후 실행 결과 값을 반환합니다.

IN 모드 파라미터

프로시저 실행에 필요한 값을 직접 입력받는 형식의 파라미터를 지정할 때 IN을 사용합니다. IN은 기본값이기 때문에 생략 가능합니다. 다음은 파라미터 네 개를 지정하고 param3, param4에는 기본값을 지정하는 코드입니다.

실습 19-7 프로시저에 파라미터 지정하기

• 완성 파일 19-7.sql

```
01 CREATE OR REPLACE PROCEDURE pro_param_in
02 (
03     param1 IN NUMBER,
04     param2 NUMBER,
05     param3 NUMBER := 3,
06     param4 NUMBER DEFAULT 4
07 )
08 IS
09
10 BEGIN
11     DBMS_OUTPUT.PUT_LINE('param1 : ' || param1);
12     DBMS_OUTPUT.PUT_LINE('param2 : ' || param2);
13     DBMS_OUTPUT.PUT_LINE('param3 : ' || param3);
14     DBMS_OUTPUT.PUT_LINE('param4 : ' || param4);
15 END;
16 /
```



param3과 param4에는 각각 기본값이 지정되어 있습니다.

:: 결과 화면

프로시저가 생성되었습니다.

지정된 파라미터가 네 개이므로 실행할 때 다음과 같이 네 개 값을 지정하여 실행할 수 있습니다.

실습 19-8 파라미터를 입력하여 프로시저 사용하기

• 완성 파일 19-8.sql

```
01 EXECUTE pro_param_in(1,2,9,8);
```

:: 결과 화면

```
param1 : 1
param2 : 2
param3 : 9
param4 : 8
```

파라미터 param3, param4는 기본값이 지정되어 있는 상태이므로 호출할 때 값을 지정하지 않아도 실행이 가능합니다. 다음과 같이 두 개 값만 지정하여 프로시저를 실행하면 기본값이 지정된 param3, param4는 기본값이 출력되고 두 값은 param1, param2 파라미터에 순서대로 입력됩니다.

실습 19-9 기본값이 지정된 파라미터 입력을 제외하고 프로시저 사용하기

· 완성 파일 19-9.sql

```
01 EXECUTE pro_param_in(1, 2);
```

:: 결과 화면

```
param1 : 1  
param2 : 2  
param3 : 3  
param4 : 4
```

param1, param2 파라미터는 기본값이 지정되어 있지 않습니다. 만약 pr_parma_in 프로시저를 호출 할 때 기본값이 지정되지 않은 파라미터 수보다 적은 수의 값을 지정하면 프로시저 실행은 실패하게 되므로 유의하기 바랍니다.

실습 19-10 실행에 필요한 개수보다 적은 파라미터를 입력하여 프로시저 실행하기 · 완성 파일 19-10.sql

```
01 EXECUTE pro_param_in(1);
```

:: 결과 화면

```
BEGIN pro_param_in(1); END;
```

*

1행에 오류:

ORA-06550: 줄 1, 열7:PLS-00306: 'PRO_PARAM_IN' 호출할 때 인수의 개수나 유형이 잘못되었습니다

ORA-06550: 줄 1, 열7:PL/SQL: Statement ignored

만약 위 예와는 달리 기본값이 지정된 파라미터와 정되지 않은 파라미터의 순서가 섞여 있다면 기본값이 지정되지 않은 파라미터까지 값을 입력해 주어야 합니다. 즉 오른쪽과 같이 파라미터가 지정되어 있다면 프로시저를 실행할 때 최소한 세 개 값을 입력해 주어야 합니다.

```
(  
    param1 IN NUMBER,  
    param2 NUMBER := 3,  
    param3 NUMBER,  
    param4 NUMBER DEFAULT 4  
)
```

하지만 파라미터 종류나 개수가 많아지면 이러한 방식은 다소 불편할 수 있습니다. 그래서 다음과 같이 =>를 사용하여 파라미터 이름에 직접 값을 대입하는 방식도 사용합니다.

실습 19-11 파라미터 이름을 활용하여 프로시저에 값 입력하기

• 완성 파일 19-11.sql

```
01 EXECUTE pro_param_in(param1 => 10, param2 => 20);
```

:: 결과 화면

param1 : 10

param2 : 20

param3 : 3

param4 : 4

이와 같이 파라미터에 값을 지정할 때는 다음 세 가지 지정 방식을 사용할 수 있습니다.

종류	설명
위치 지정	지정한 파라미터 순서대로 값을 지정하는 방식
이름 지정	=> 연산자로 파라미터 이름을 명시하여 값을 지정하는 방식
혼합 지정	일부 파라미터는 순서대로 값만 지정하고 일부 파라미터는 => 연산자로 값을 지정하는 방식(11g부터 사용 가능)

OUT 모드 파라미터

OUT 모드를 사용한 파라미터는 프로시저 실행 후 호출한 프로그램으로 값을 반환합니다. 사원 번호를 입력받아 사원 이름과 급여를 반환하는 pro_param_out 프로시저를 생성해 보죠.

실습 19-12 OUT 모드 파라미터 정의하기

• 완성 파일 19-12.sql

```
01 CREATE OR REPLACE PROCEDURE pro_param_out
02 (
03     in_empno IN EMP.EMPNO%TYPE,
04     out_ename OUT EMP.ENAME%TYPE,
05     out_sal OUT EMP.SAL%TYPE
06 )
07 IS
08
09 BEGIN
10     SELECT ENAME, SAL INTO out_ename, out_sal
11     FROM EMP
12     WHERE EMPNO = in_empno;
13 END pro_param_out;
14 /
```

:: 결과 화면

프로시저가 생성되었습니다.

OUT 모드로 지정한 두 파라미터 out_ename, out_sal은 pro_param_out 프로시저를 실행한 후 값이 반환됩니다. 이렇게 반환되는 값을 다음과 같이 또 다른 PL/SQL 블록에서 받아서 처리할 수 있습니다. 변수 두 개를 선언하여 pro_param_out 프로시저의 반환 값을 대입하는 것도 눈여겨보세요.

실습 19-13 OUT 모드 파라미터 사용하기

· 완성 파일 19-13.sql

```
01  DECLARE
02      v_ename EMP.ENAME%TYPE;
03      v_sal EMP.SAL%TYPE;
04  BEGIN
05      pro_param_out(7788, v_ename, v_sal);
06      DBMS_OUTPUT.PUT_LINE('ENAME : ' || v_ename);
07      DBMS_OUTPUT.PUT_LINE('SAL : ' || v_sal);
08  END;
09  /
```

:: 결과 화면

```
ENAME : SCOTT
SAL : 3000
```

IN OUT 모드 파라미터

IN OUT 모드로 선언한 파라미터는 IN, OUT으로 선언한 파라미터 기능을 동시에 수행합니다. 즉 값을 입력받을 때와 프로시저 수행 후 결과 값을 반환할 때 사용합니다.

실습 19-14 IN OUT 모드 파라미터 정의하기

· 완성 파일 19-14.sql

```
01  CREATE OR REPLACE PROCEDURE pro_param_inout
02  (
03      inout_no IN OUT NUMBER
04  )
05  IS
06
07  BEGIN
08      inout_no := inout_no * 2;
09  END pro_param_inout;
10  /
```

:: 결과 화면

```
프로시저가 생성되었습니다.
```

OUT 모드로 선언된 파라미터와 마찬가지로 IN OUT 모드로 선언된 파라미터 역시 다른 PL/SQL 블록에서 데이터를 대입받아 사용할 수 있습니다.

실습 19-15 IN OUT 모드 파라미터 사용하기

· 완성 파일 19-15.sql

```
01  DECLARE
02      no NUMBER;
03  BEGIN
04      no := 5;
05      pro_param_inout(no);
06      DBMS_OUTPUT.PUT_LINE('no : ' || no);
07  END;
08 /
```

:: 결과 화면

no : 10

1분 복습

다음은 파라미터를 갖는 프로시저의 여러 모드와 그 설명입니다. 개념에 알맞은 설명을 연결해 보세요.

- | | | |
|-----------|---|---|
| 1. IN | • | ① 호출할 때 값을 입력받은 후 실행 결과 값을 반환합니다. |
| 2. OUT | • | ② 지정하지 않으면 기본값으로 프로시저를 호출할 때 값을 입력받습니다. |
| 3. IN OUT | • | ③ 호출할 때 값을 반환합니다. |

정답 1. ② 2. ③ 3. ①

프로시저 오류 정보 확인하기

앞의 여러 실습에서 다양한 프로시저를 생성해 보았습니다. 아마도 오타나 문법 실수로 프로시저가 제대로 생성되지 않아 다시 만들어야 했던 독자도 있을 겁니다. 그런 독자들을 위해 프로시저를 생성할 때 발생하는 오류를 확인하는 방법을 살펴보겠습니다. 이 방법은 다른 서브프로그램의 오류에도 똑같이 적용할 수 있습니다. 먼저 다음과 같이 오류가 발생하는 프로시저 생성 문장을 실행해 보죠.

실습 19-16 생성할 때 오류가 발생하는 프로시저 알아보기

· 완성 파일 19-16.sql

```
01  CREATE OR REPLACE PROCEDURE pro_err
02  IS
03      err_no NUMBER;
04  BEGIN
05      err_no = 100;
```

```
06      DBMS_OUTPUT.PUT_LINE('err_no : ' || err_no);
07  END pro_err;
08 /
```

:: 결과 화면

경고 : 컴파일 오류와 함께 프로시저가 생성되었습니다.

'컴파일 오류와 함께 프로시저가 생성되었습니다.'라는 메시지와 함께 프로시저를 만들 때 오류가 발생했음을 알 수 있습니다. 서브프로그램을 만들 때 발생한 오류는 SHOW ERRORS 명령어와 USER_ERRORS 데이터 사전을 조회하여 확인할 수 있습니다.

SHOW ERRORS로 오류 확인

SQL*PLUS에서 발생한 오류를 확인하는 가장 간단한 방법은 SHOW ERRORS 명령어를 사용하는 것입니다. SHOW ERRORS 명령어는 가장 최근에 생성되거나 변경된 서브프로그램의 오류 정보를 출력합니다. 여기에서는 다섯 번째 줄에서 오류가 발생했음을 알 수 있습니다.

실습 19-17 SHOW ERRORS 명령어로 오류 확인하기

• 완성 파일 19-17.sql

```
01 SHOW ERRORS;
```

:: 결과 화면

PROCEDURE PRO_ERR에 대한 오류:

LINE/COL	ERROR
5/8	PLS-00103: 심볼 "="를 만났습니다 다음 중 하나가 기대될 때: := . (@ % ; 심볼이 "=:" 계속하기 위해 "=" 전에 삽입되었음

SHOW ERRORS 명령어는 줄여서 SHOW ERR로 사용할 수도 있습니다. 만약 최근에 발생한 프로그램 오류가 아니라 특정 프로그램의 오류 정보를 확인하고 싶다면 다음과 같이 프로그램 종류와 이름을 추가로 지정하면 됩니다. 이번 예제는 프로시저로 생성했으므로 PROCEDURE를 붙여 줍니다.

```
SHOW ERR 프로그램 종류 프로그램 이름;
SHOW ERR PROCEDURE pro_err;
```

USER_ERRORS로 오류 확인

토드 같은 응용 프로그램을 사용하고 있다면 USER_ERRORS 데이터 사전을 조회하여 오류 정보를 확인할 수 있습니다.

실습 19-18 USER_ERRORS로 오류 확인하기

• 완성 파일 19-18.sql

```
01  SELECT *
02    FROM USER_ERRORS
03   WHERE NAME = 'PRO_ERR';
```

:: 결과 화면(SQL*PLUS)

NAME	TYPE	SEQUENCE	LINE	POSITION
TEXT				

ATTRIBUTE	MESSAGE_NUMBER
PRO_ERR	PROCEDURE
PLS-00103: 심볼 "="를 만났습니다 다음 중 하나가 기대될 때 := . (@ % ;	1 5 8
심볼이 "=:" 계속하기 위해 "=" 전에 삽입되었음	
ERROR	103

:: 결과 화면(토드)

NAME	TYPE	SEQUENCE	LINE	POSITION	TEXT	ATTRIBUTE	MESSAGE_NUMBER
PRO_ERR	PROCEDURE	1	5	11	PLS-00103: 심볼 "="를 만났습니다 다음 중 하나가 기대될 때 := . (@ % ; ...	ERROR	103

19-3 함수

오라클 함수는 크게 내장 함수(built-in function)와 사용자 정의 함수(user defined function)로 분류할 수 있고 그 중 자주 사용하는 내장 함수를 06장과 07장에서 소개했습니다.

이번에는 PL/SQL을 사용하여 함수를 직접 정의하는 방법을 알아보죠. 함수 제작 방식은 먼저 살펴본 프로시저 제작 방식과 비슷한 면이 많아 혼동하는 경우도 많은데 먼저 프로시저와 함수의 차이점을 살펴보겠습니다.

특징	프로시저	함수
실행	EXECUTE 명령어 또는 다른 PL/SQL 서브프로그램 내에서 호출하여 실행	변수를 사용한 EXECUTE 명령어 또는 다른 PL/SQL 서브프로그램에서 호출하여 실행하거나 SQL문에서 직접 실행 가능
파라미터 지정	필요에 따라 지정하지 않을 수도 있고 여러 개 지정할 수도 있으며 IN, OUT, IN OUT 세 가지 모드를 사용할 수 있음	프로시저와 같게 지정하지 않을 수도 있고 여러 개 지정할 수 있지만 IN 모드(또는 생략)만 사용
값의 반환	실행 후 값의 반환이 없을 수도 있고 OUT, IN OUT 모드의 파라미터 수에 따라 여러 개 값을 반환할 수 있음	반드시 하나의 값을 반환해야 하며 값의 반환은 프로시저와 달리 OUT, IN OUT 모드의 파라미터를 사용하는 것이 아니라 RETURN절과 RETURN문을 통해 반환

위 표에서 알 수 있듯이 함수는 프로시저와 달리 SQL문에서도 사용할 수 있다는 특징이 있습니다. RETURN절과 RETURN문을 통해 반드시 하나의 값을 반환해야 한다는 것을 기억하세요.

함수 생성하기

함수 생성은 프로시저와 마찬가지로 CREATE [OR REPLACE] 명령어와 FUNCTION 키워드를 명시하여 생성합니다. 앞에서 다룬 프로시저와 작성 방식이나 문법 면에서는 별 차이가 없습니다. 다만 함수는 반환 값의 자료형과 실행부에서 반환할 값을 RETURN절 및 RETURN문으로 명시해야 하죠. 실행부의 RETURN문이 실행되면 함수 실행은 즉시 종료된다는 것도 기억하세요.

```

CREATE [OR REPLACE] FUNCTION 함수 이름
[(파라미터 이름1 [IN] 자료형1, -①
  파라미터 이름2 [IN] 자료형2,
  ...
  파라미터 이름N [IN] 자료형N
)]
RETURN 자료형 -②
IS | AS
선언부
BEGIN
  실행부
  RETURN (반환 값); -③
EXCEPTION
  예외 처리부
END [함수 이름];

```

번호	설명
①	함수 실행에 사용할 입력 값이 필요하면 파라미터를 지정합니다. 파라미터 지정은 생략 가능하며 필요에 따라 여러 개 정의할 수 있습니다. 프로시저와 달리 IN 모드만 지정합니다. :=, DEFAULT 옵션으로 기본값을 지정할 수도 있습니다.
②	함수의 실행 후 반환 값의 자료형을 정의합니다.
③	함수의 반환 값을 지정합니다.

▣ 한 발 더 나가기! 함수의 파라미터에 OUT, IN OUT 모드 사용

함수에서 파라미터를 지정하는 방식은 프로시저와 달리 IN 모드만을 사용한다고 했는데요. 엄밀히 말하면 함수에도 OUT, IN OUT 모드의 파라미터를 지정할 수 있습니다. 하지만 함수에 OUT, IN OUT 모드의 파라미터가 지정되어 있으면 SQL문에서는 사용할 수 없는 함수가 됩니다. 즉 프로시저와 달리 바 없는 함수가 되는 것이죠. 이러한 이유와 부작용 때문에 오라클에서도 함수 파라미터에 OUT, IN OUT 모드를 사용하지 말라고(avoid) 권장합니다.

비슷한 맥락으로 SQL문에 사용할 함수는 반환 값의 자료형을 SQL문에서 사용할 수 없는 자료형으로 지정할 수 없으며 트랜잭션을 제어하는 명령어(TCL, DDL) 또는 DML 명령어도 사용할 수 없습니다.

그러면 함수를 생성해 볼까요? 다음은 급여 값을 입력받아 세금을 제한 실수령액을 계산하는 함수입니다.

실습 19-19 함수 생성하기

· 완성 파일 19-19.sql

```
01 CREATE OR REPLACE FUNCTION func_aftertax(
02     sal IN NUMBER
03 )
04 RETURN NUMBER
05 IS
06     tax NUMBER := 0.05;
07 BEGIN
08     RETURN (ROUND(sal - (sal * tax)));
09 END func_aftertax;
10 /
```

:: 결과 화면

함수가 생성되었습니다.

함수 실행하기

생성된 함수는 익명 블록 또는 프로시저 같은 저장 서브프로그램, SQL문에서 사용할 수 있습니다. PL/SQL로 실행할 때는 함수 반환 값을 대입받을 변수가 필요합니다.

PL/SQL로 함수 실행하기

함수는 실행 후 하나의 값을 반환하므로 PL/SQL로 구현한 프로그램 안에 반환 값을 받기 위한 변수를 선언하여 사용합니다.

실습 19-20 PL/SQL에서 함수 사용하기

· 완성 파일 19-20.sql

```
01 DECLARE
02     aftertax NUMBER;
03 BEGIN
04     aftertax := func_aftertax(3000);
05     DBMS_OUTPUT.PUT_LINE('after-tax income : ' || aftertax);
06 END;
07 /
```

:: 결과 화면

after-tax income : 2850

SQL문에서 함수 실행하기

SQL문에서 제작한 함수를 사용하는 방식은 기존 오라클의 내장 함수와 같은데요. DUAL 테이블에 다음과 같이 값을 직접 입력하여 사용할 수 있습니다.

실습 19-21 SQL에서 함수 사용하기

• 완성 파일 19-21.sql

```
01  SELECT func_aftertax(3000)
02    FROM DUAL;
```

:: 결과 화면

```
FUNC_AFTERTAX(3000)
```

```
-----  
2850
```

함수에 정의한 파라미터와 자료형이 일치한다면 내장 함수와 마찬가지로 특정 열 또는 열 데이터 간에 연산 가공된 데이터를 입력하는 것도 가능합니다.

실습 19-22 함수에 테이블 데이터 사용하기

• 완성 파일 19-22.sql

```
01  SELECT EMPNO, ENAME, SAL, func_aftertax(SAL) AS AFTERTAX
02    FROM EMP;
```

:: 결과 화면

EMPNO	ENAME	SAL	AFTERTAX
7369	SMITH	800	760
7499	ALLEN	1600	1520
7521	WARD	1250	1188
7566	JONES	2975	2826
7654	MARTIN	1250	1188
7698	BLAKE	2850	2708
7782	CLARK	2450	2328
7788	SCOTT	3000	2850
7839	KING	5000	4750
7844	TURNER	1500	1425
7876	ADAMS	1100	1045
7900	JAMES	950	903
7902	FORD	3000	2850
7934	MILLER	1300	1235

14개의 행이 선택되었습니다.

함수 삭제하기

다른 객체와 마찬가지로 DROP FUNCTION 명령어를 사용하여 함수를 삭제합니다.

실습 19-23 함수 삭제하기

• 완성 파일 19-23.sql

```
01  DROP FUNCTION func_aftertax;
```

:: 결과 화면

함수가 삭제되었습니다.

1분
복습

다음은 함수의 대한 설명입니다. 틀린 내용을 골라 보세요.

1. SQL문에서 직접 실행이 가능하다.
2. IN, OUT, IN OUT모드를 모두 사용할 수 있다.
3. 반드시 하나의 값을 반환해야 한다.
4. 값을 반환할 때는 RETURN절 또는 RETURN문을 사용해야 한다.

2. 정답

19-4 패키지

패키지(package)는 업무나 기능 면에서 연관성이 높은 프로시저, 함수 등 여러 개의 PL/SQL 서브프로그램을 하나의 논리 그룹으로 묶어 통합·관리하는 데 사용하는 객체를 뜻합니다. 앞에서 다룬 프로시저나 함수 등은 각각 개별 기능을 수행하기 위해 제작 후 따로 저장했습니다. 패키지를 사용하여 서브프로그램을 그룹화하면 다음과 같은 장점이 있습니다.

장점	설명
모듈성	서브프로그램을 포함한 여러 PL/SQL 구성 요소를 모듈화할 수 있습니다. 모듈성은 잘 묶어 둔다는 뜻으로 프로그램의 이해를 쉽게 하고 패키지 사이의 상호 작용을 더 간편하고 명료하게 해 주는 역할을 합니다. 즉 PL/SQL로 제작한 프로그램의 사용 및 관리에 큰 도움을 줍니다.
쉬운 응용 프로그램 설계	패키지에 포함될 서브프로그램은 원격하게 완성되지 않아도 정의가 가능합니다. 이 때문에 전체 소스 코드를 다 작성하기 전에 미리 패키지에 저장할 서브프로그램을 지정할 수 있으므로 설계가 수월해집니다.
정보 은닉	제작 방식에 따라 패키지에 포함하는 서브프로그램의 외부 노출 여부 또는 접근 여부를 지정할 수 있습니다. 즉 서브프로그램을 사용할 때 보안을 강화할 수 있습니다.
기능성 향상	패키지 내부에는 서브프로그램 외에 변수·커서·예외 등도 각 세션이 유지되는 동안 선언해서 공용(public)으로 사용할 수 있습니다. 예를 들어 특정 커서 데이터는 세션이 종료되기 전까지 보존되므로 여러 서브프로그램에서 사용할 수 있습니다.
성능 향상	패키지를 사용할 때 패키지에 포함한 모든 서브프로그램이 메모리에 한 번에 로딩되는데 메모리에 로딩된 후의 호출은 디스크 I/O를 일으키지 않으므로 성능이 향상됩니다.

장점으로 기술한 내용이 다소 어렵게 느껴질 수도 있지만 PL/SQL 서브프로그램의 제작·사용·관리·보안·성능 등에 좋은 영향을 끼친다는 정도로 이해해도 괜찮습니다.

패키지 구조와 생성

패키지는 프로시저, 함수와 달리 보통 두 부분으로 나누어 제작하는데요. 하나는 명세(specification) 또 하나는 본문(body)이라고 부릅니다.

패키지 명세

패키지 명세는 패키지에 포함할 변수, 상수, 예외, 커서 그리고 PL/SQL 서브프로그램을 선언하는 용도로 작성합니다. 패키지 명세에 선언한 여러 객체는 패키지 내부뿐만 아니라 외부에서도 참조할 수 있습니다.

```
CREATE [OR REPLACE] PACKAGE 패키지 이름
IS | AS
    서브프로그램을 포함한 다양한 객체 선언
END [패키지 이름];
```

기본 형식

그러면 패키지 명세를 작성해 볼까요? 다음 패키지 명세는 변수 한 개, 함수 한 개, 프로시저 두 개를 선언합니다.

실습 19-24 패키지 생성하기

· 완성 파일 19-24.sql

```
01 CREATE OR REPLACE PACKAGE pkg_example
02 IS
03     spec_no NUMBER := 10;
04     FUNCTION func_aftertax(sal NUMBER) RETURN NUMBER;
05     PROCEDURE pro_emp(in_empno IN EMP.EMPNO%TYPE);
06     PROCEDURE pro_dept(in_deptno IN DEPT.DEPTNO%TYPE);
07 END;
08 /
```

:: 결과 확인

패키지가 생성되었습니다.

이미 생성되어 있는 패키지 명세의 코드를 확인하거나 선언한 서브프로그램을 확인하려면 USER_SOURCE 데이터 사전을 조회하거나 DESC 명령어를 활용할 수 있습니다.

실습 19-25 패키지 명세 확인하기(USER_SOURCE 데이터 사전으로 조회)

· 완성 파일 19-25.sql

```
01 SELECT TEXT
02     FROM USER_SOURCE
03     WHERE TYPE = 'PACKAGE'
04     AND NAME = 'PKG_EXAMPLE';
```

:: 결과 확인

TEXT

```
-----  
PACKAGE pkg_example
```

```

IS
spec_no NUMBER := 10;
FUNCTION func_aftertax(sal NUMBER) RETURN NUMBER;
PROCEDURE pro_emp(in_empno IN EMP.EMPNO%TYPE);
PROCEDURE pro_dept(in_deptno IN DEPT.DEPTNO%TYPE);
END;

```

7개 행이 선택되었습니다.

실습 19-26 패키지 명세 확인하기(DESC 명령어로 조회)

• 완성 파일 19-26.sql

01 DESC pkg_example;

:: 결과 화면

FUNCTION FUNC_AFTERTAX RETURNS NUMBER		
인수명	유형	기본 내부/외부?
SAL	NUMBER	IN
PROCEDURE PRO_DEPT		
인수명	유형	기본 내부/외부?
IN_DEPTNO	NUMBER(2)	IN
PROCEDURE PRO_EMP		
인수명	유형	기본 내부/외부?
IN_EMPNO	NUMBER(4)	IN

패키지 본문

패키지 본문에는 패키지 명세에서 선언한 서브프로그램 코드를 작성합니다. 그리고 패키지 명세에 선언하지 않은 객체나 서브프로그램을 정의하는 것도 가능합니다. 이때 패키지 본문에만 존재하는 프로그램은 패키지 내부에서만 사용할 수 있습니다. 패키지 본문 이름은 패키지 명세 이름과 같게 지정해야 합니다.

CREATE [OR REPLACE] PACKAGE BODY 패키지 이름

기본 형식

IS | AS

패키지 명세에서 선언한 서브프로그램을 포함한 여러 객체를 정의

경우에 따라 패키지 명세에 존재하지 않는 객체 및 서브프로그램도 정의 가능

END [패키지 이름];

실습 19-24에서 생성한 패키지 명세 pkg_example의 패키지 본문을 작성해 보죠. 본문에서는 패키지 명세에 선언한 함수와 프로시저를 기술하고 body_no 변수를 선언합니다. body_no 변수는 pkg_example 패키지 안에서만 사용할 수 있습니다.

실습 19-27 패키지 본문 생성하기

· 완성 파일 19-27.sql

```
01 CREATE OR REPLACE PACKAGE BODY pkg_example
02 IS
03     body_no NUMBER := 10;
04
05     FUNCTION func_aftertax(sal NUMBER) RETURN NUMBER
06     IS
07         tax NUMBER := 0.05;
08     BEGIN
09         RETURN (ROUND(sal - (sal * tax)));
10     END func_aftertax;
11
12     PROCEDURE pro_emp(in_empno IN EMP.EMPNO%TYPE)
13     IS
14         out_ename EMP.ENAME%TYPE;
15         out_sal EMP.SAL%TYPE;
16     BEGIN
17         SELECT ENAME, SAL INTO out_ename, out_sal
18             FROM EMP
19             WHERE EMPNO = in_empno;
20
21         DBMS_OUTPUT.PUT_LINE('ENAME : ' || out_ename);
22         DBMS_OUTPUT.PUT_LINE('SAL : ' || out_sal);
23     END pro_emp;
24
25     PROCEDURE pro_dept(in_deptno IN DEPT.DEPTNO%TYPE)
26     IS
27         out_dname DEPT.DNAME%TYPE;
28         out_loc DEPT.LOC%TYPE;
29     BEGIN
30         SELECT DNAME, LOC INTO out_dname, out_loc
31             FROM DEPT
32             WHERE DEPTNO = in_deptno;
33
34         DBMS_OUTPUT.PUT_LINE('DNAME : ' || out_dname);
35         DBMS_OUTPUT.PUT_LINE('LOC : ' || out_loc);
```

```
36      END pro_dept;
37  END;
38 /
```

:: 결과 화면

패키지 본문이 생성되었습니다.

서브프로그램 오버로드

기본적으로 서브프로그램 이름은 중복될 수 없습니다. 하지만 같은 패키지에서 사용하는 파라미터의 개수, 자료형, 순서가 다를 경우에 한해서만 이름이 같은 서브프로그램을 정의할 수 있습니다. 이를 서브프로그램 오버로드(subprogram overload)라고 합니다. 서브프로그램 오버로드는 보통 같은 기능을 수행하는 여러 서브프로그램이 입력 데이터를 각각 다르게 정의 할 때 사용합니다. 또한 서브프로그램 종류가 같아야 오버로드가 가능합니다. 즉 특정 프로시저를 오버로드할 때 반드시 이름이 같은 프로시저로 정의해야 합니다. 프로시저와 이름이 같은 함수를 정의할 수는 없습니다.

CREATE [OR REPLACE] PACKAGE 패키지 이름

기본 형식

IS | AS

서브프로그램 종류 서브프로그램 이름(파라미터 정의);

서브프로그램 종류 서브프로그램 이름(개수나 자료형, 순서가 다른 파라미터 정의);

END [패키지 이름];

다음 실습은 사원 번호 또는 사원 이름을 입력받아 사원 이름과 급여를 출력하기 위해 pro_emp 프로시저를 오버로드하는 패키지와 패키지 본문을 생성합니다.

실습 19-28 프로시저 오버로드하기

· 완성 파일 19-28.sql

```
01 CREATE OR REPLACE PACKAGE pkg_overload
02 IS
03     PROCEDURE pro_emp(in_empno IN EMP.EMPNO%TYPE);
04     PROCEDURE pro_emp(in_ename IN EMP.ENAME%TYPE);
05 END;
06 /
```

:: 결과 화면

패키지가 생성되었습니다.

실습 19-29 패키지 본문에서 오버로드된 프로시저 작성하기

• 완성 파일 19-29.sql

```
01 CREATE OR REPLACE PACKAGE BODY pkg_overload
02 IS
03     PROCEDURE pro_emp(in_empno IN EMP.EMPNO%TYPE)
04     IS
05         out_ename EMP.ENAME%TYPE;
06         out_sal EMP.SAL%TYPE;
07     BEGIN
08         SELECT ENAME, SAL INTO out_ename, out_sal
09             FROM EMP
10            WHERE EMPNO = in_empno;
11
12         DBMS_OUTPUT.PUT_LINE('ENAME : ' || out_ename);
13         DBMS_OUTPUT.PUT_LINE('SAL : ' || out_sal);
14     END pro_emp;
15
16     PROCEDURE pro_emp(in_ename IN EMP.ENAME%TYPE)
17     IS
18         out_ename EMP.ENAME%TYPE;
19         out_sal EMP.SAL%TYPE;
20     BEGIN
21         SELECT ENAME, SAL INTO out_ename, out_sal
22             FROM EMP
23            WHERE ENAME = in_ename;
24
25         DBMS_OUTPUT.PUT_LINE('ENAME : ' || out_ename);
26         DBMS_OUTPUT.PUT_LINE('SAL : ' || out_sal);
27     END pro_emp;
28
29 END;
30 /
```

:: 결과 화면

패키지 본문이 생성되었습니다.

**1분
복습**

다음 설명의 패키지의 장점 중 어떤 내용에 대한 설명인지 골라 보세요.

제작 방식에 따라 패키지에 포함하는 서브프로그램의 외부 노출 여부 또는 접근 여부를 지정할 수 있습니다. 즉 서브프로그램을 사용할 때 보안을 강화할 수 있습니다.

1. 모듈성 2. 성능 향상 3. 쉬운 응용 프로그램 설계 4. 정보 은닉

패키지 사용하기

패키지를 통해 그룹화된 변수, 상수, 예외, 커서 그리고 PL/SQL 서브프로그램은 패키지 이름과 마침표(.)와 사용할 객체 이름으로 사용할 수 있습니다.

패키지 이름.객체 이름:

다음은 pkg_example, pkg_overload 패키지의 서브프로그램을 실행한 결과입니다.

실습 19-30 패키지에 포함된 서브프로그램 실행하기

• 완성 파일 19-30.sql

```
01 BEGIN
02   DBMS_OUTPUT.PUT_LINE('--pkg_example.func_aftertax(3000)--');
03   DBMS_OUTPUT.PUT_LINE('after-tax: ' || pkg_example.func_aftertax(3000));
04
05   DBMS_OUTPUT.PUT_LINE('--pkg_example.pro_emp(7788)--');
06   pkg_example.pro_emp(7788);
07
08   DBMS_OUTPUT.PUT_LINE('--pkg_example.pro_dept(10)--' );
09   pkg_example.pro_dept(10);
10
11  DBMS_OUTPUT.PUT_LINE('--pkg_overload.pro_emp(7788)--' );
12  pkg_overload.pro_emp(7788);
13
14  DBMS_OUTPUT.PUT_LINE('--pkg_overload.pro_emp(''SCOTT'')--' );
15  pkg_overload.pro_emp('SCOTT');
16 END;
17 /
```

:: 결과 화면

```
--pkg_example.func_aftertax(3000)-
after-tax:2850
--pkg_example.pro_emp(7788)--
ENAME : SCOTT
SAL : 3000
--pkg_example.pro_dept(10)--
DNAME : ACCOUNTING
LOC : NEW YORK
--pkg_overload.pro_emp(7788)--
ENAME : SCOTT
SAL : 3000
--pkg_overload.pro_emp('SCOTT')--
ENAME : SCOTT
SAL : 3000
```

패키지 삭제하기

두 가지 방식을 사용하여 패키지를 삭제할 수 있습니다. 패키지 명세와 본문을 한 번에 삭제하거나 패키지 본문만 삭제할 수도 있습니다. 하지만 패키지에 포함된 서브프로그램을 따로 삭제하는 것은 불가능합니다. CREATE OR REPLACE문을 활용하여 패키지 안의 객체 또는 서브프로그램을 수정 및 삭제할 수 있습니다.

패키지 명세와 본문을 한 번에 삭제하기

DROP PACKAGE 패키지 이름;

패키지의 본문만을 삭제

DROP PACKAGE BODY 패키지 이름;

❷ 한 발 더 나가기!! 오라클 제공 패키지란?

오라클 함수는 오라클에서 기본으로 제공하는 내장 함수와 필요에 따라 직접 정의하는 사용자 정의 함수로 구분합니다. 패키지 역시 앞서 살펴본 실습처럼 사용자가 직접 그룹화한 서브프로그램을 정의하여 제작할 수도 있고, 오라클에서 기본으로 제공하는 패키지를 사용할 수도 있습니다. 오라클에서 제공하는 패키지 종류는 무척 많은데요. 이 책은 오라클에서 제공하는 패키지를 소개하고 있지 않으므로 자세한 내용이 필요하다면 오라클 공식 문서(docs.oracle.com/cd/B28359_01/appdev.111/b28419/intro.htm#ARPLS001)를 참고하세요.

19-5 트리거

트리거란?

오라클에서 트리거(trigger)는 데이터베이스 안의 특정 상황이나 동작, 즉 이벤트가 발생할 경우에 자동으로 실행되는 기능을 정의하는 PL/SQL 서브프로그램입니다.

☞ trigger를 사전에서 찾아보면 총의 방아쇠, 계기 등 연속성을 가지는 어떤 동작의 시작점이라는 뜻입니다.

예를 들어 어떤 테이블의 데이터를 특정 사용자가 변경하려 할 때 해당 데이터나 사용자 기록을 확인한다든지 상황에 따라 데이터를 변경하지 못하게 막는 것이 가능합니다. 오라클 데이터베이스가 가동하거나 종료할 때 데이터베이스 관리자 등 관련 업무자에게 메일을 보내는 기능도 구현할 수 있습니다. 이러한 트리거를 통해 연관 데이터 작업을 잘 정의해 두면 다음과 같은 장점이 있습니다.

1. 데이터와 연관된 여러 작업을 수행하기 위해 여러 PL/SQL문 또는 서브프로그램을 일일이 실행해야 하는 번거로움을 줄일 수 있습니다. 즉 데이터 관련 작업을 좀 더 간편하게 수행할 수 있습니다.
2. 제약 조건(constraints)만으로 구현이 어렵거나 불가능한 좀 더 복잡한 데이터 규칙을 정할 수 있어 더 수준 높은 데이터 정의가 가능합니다.
3. 데이터 변경과 관련된 일련의 정보를 기록해 둘 수 있으므로 여러 사용자가 공유하는 데이터 보안성과 안정성 그리고 문제가 발생했을 때 대처 능력을 높일 수 있습니다.

하지만 트리거는 특정 작업 또는 이벤트 발생으로 다른 데이터 작업을 추가로 실행하기 때문에 무분별하게 사용하면 데이터베이스 성능을 떨어뜨리는 원인이 되므로 주의가 필요합니다. 트리거는 테이블·뷰·스키마·데이터베이스 수준에서 다음과 같은 이벤트에 동작을 지정할 수 있습니다.

- 데이터 조작어(DML) : INSERT, UPDATE, DELETE
- 데이터 정의어(DDL) : CREATE, ALTER, DROP
- 데이터베이스 동작 : SERVERERROR, LOGON, LOGOFF, STARTUP, SHUTDOWN

그리고 이렇게 트리거가 발생할 수 있는 이벤트 종류에 따라 오라클은 트리거를 다음과 같이 구분합니다.

종류	설명
DML 트리거	INSERT, UPDATE, DELETE와 같은 DML 명령어를 기점으로 동작함
DDL 트리거	CREATE, ALTER, DROP과 같은 DDL 명령어를 기점으로 동작함
INSTEAD OF 트리거	뷰(View)에 사용하는 DML 명령어를 기점으로 동작함
시스템(system) 트리거	데이터베이스나 스키마 이벤트로 동작함
단순(simple) 트리거	다음 각 시점(timing point)에 동작함 <ul style="list-style-type: none">• 트리거가 작동시킬 문장이 실행되기 전 시점• 트리거가 작동시킬 문장이 실행된 후 시점• 트리거가 작동시킬 문장이 행에 영향을 미치기 전 시점• 트리거가 작동시킬 문장이 행에 영향을 준 후 시점
복합(compound) 트리거	단순 트리거의 여러 시점에 동작함

이 책에서는 가장 사용 빈도가 높은 DML 트리거를 예로 간단한 트리거를 제작해 보겠습니다.

☞ 좀 더 다양한 트리거의 사용법은 오라클 공식 문서(docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#LNPLS020)를 참고하세요.

DML 트리거

DML 트리거 형식

DML 트리거는 특정 테이블에 DML 명령어를 실행했을 때 작동하는 트리거입니다. 기본 형식은 다음과 같습니다.

```
CREATE [OR REPLACE] TRIGGER 트리거 이름 -①
BEFORE | AFTER -②
INSERT | UPDATE | DELETE ON 테이블 이름 -③
REFERENCING OLD as old | New as new -④
FOR EACH ROW WHEN 조건식 -⑤
FOLLOWS 트리거 이름2, 트리거 이름3 ... -⑥
ENABLE | DISABLE -⑦
```

기본 형식

```
DECLARE
  선언부
BEGIN
  실행부
```

EXCEPTION

예외 처리부

END;

번호	설명
①	트리거 이름을 명시하고 트리거를 생성합니다. 트리거 내용을 간신하려면 OR REPLACE 키워드를 함께 명시합니다.
②	트리거가 작동할 타이밍을 지정합니다. BEFORE는 DML 명령어가 실행되기 전 시점, AFTER는 DML 명령어가 실행된 후 시점에 트리거가 작동합니다.
③	지정한 테이블에 트리거가 작동할 DML 명령어를 작성합니다. 여러 종류의 DML 명령어를 지정할 경우에는 OR로 구분합니다.
④	DML로 변경되는 행의 변경 전 값과 변경 후 값을 참조하는 데 사용합니다(생략 가능).
⑤	트리거를 실행하는 DML 문장에 한 번만 실행할지 DML 문장에 의해 영향받는 행별로 실행할지를 지정합니다. 생략하면 트리거는 DML 명령어가 실행될 때 한 번만 실행합니다. 생략하지 않고 사용할 경우, DML 명령어에 영향받는 행별로 트리거를 작동하되 WHEN 키워드를 함께 사용하면 DML 명령어에 영향받는 행 중 트리거를 작동시킬 행을 조건식으로 지정할 수 있습니다.
⑥	오라클 11g부터 사용 가능한 키워드로서 여러 관련 트리거의 실행 순서를 지정합니다(생략 가능).
⑦	오라클 11g부터 사용 가능한 키워드로서 트리거의 활성화·비활성화를 지정합니다(생략 가능).

DML 트리거의 제작 및 사용(BEFORE)

트리거를 작성해 볼까요? 먼저 DML문을 실행하기 전에 트리거가 작동하는 BEFORE 트리거를 작성해 보죠. 트리거를 적용할 테이블을 다음과 같이 EMP 테이블을 복사하여 생성합니다.

실습 19-31 EMP_TRG 테이블 생성하기

• 완성 파일 19-31.sql

```
01 CREATE TABLE EMP_TRG
02     AS SELECT * FROM EMP;
```

:: 결과 확인

테이블이 생성되었습니다.

생성된 EMP_TRG 테이블에 DML 명령어를 실행하기 직전에 작동할 트리거를 다음과 같이 만들어 보죠. trg_emp_nodml_weekend 트리거는 주말에 EMP_TRG 테이블에 DML 명령어를 사용하면 오류를 일으키고 DML 명령어 실행을 취소합니다.

```

01 CREATE OR REPLACE TRIGGER trg_emp_nodml_weekend
02 BEFORE
03 INSERT OR UPDATE OR DELETE ON EMP_TRG
04 BEGIN
05   IF TO_CHAR(sysdate, 'DY') IN ('토', '일') THEN
06     IF INSERTING THEN
07       raise_application_error(-20000, '주말 사원정보 추가 불가');
08     ELSIF UPDATING THEN
09       raise_application_error(-20001, '주말 사원정보 수정 불가');
10     ELSIF DELETING THEN
11       raise_application_error(-20002, '주말 사원정보 삭제 불가');
12     ELSE
13       raise_application_error(-20003, '주말 사원정보 변경 불가');
14     END IF;
15   END IF;
16 END;
17 /

```

:: 결과 화면

트리거가 생성되었습니다.

01행 트리거 이름(trg_emp_nodml_weekend)을 지정합니다.

02행 DML 명령어를 사용하기 전 실행할 BEFORE 트리거를 정의합니다.

03행 EMP_TRG 테이블을 대상으로 INSERT, UPDATE, DELETE 명령어가 실행될 때 트리거가 작동합니다.

05행 DML을 실행하는 시점이 토요일이나 일요일일 경우를 의미합니다.

06행 INSERT 명령어가 실행되었을 경우를 의미합니다.

07행 주말에 INSERT가 발생했을 경우 raise_application_error 프로시저를 사용하여 사용자 정의 예외를 발생시킵니다. 사용자 정의 예외의 첫 번째 파라미터는 예외 코드를 의미하며 사용자 정의 예외 코드는 -20000 ~ 20999 범위를 사용할 수 있습니다.

08행 UPDATE 명령어가 실행되었을 경우를 의미합니다.

10행 DELETE 명령어가 실행되었을 경우를 의미합니다.

이제 트리거가 작동하도록 EMP_TRG 테이블에 DML 명령어를 사용해 봅시다. 트리거는 특정 이벤트 발생할 때 자동으로 작동하는 서브프로그램이므로 프로시저나 함수와 같이 EXECUTE 또는 PL/SQL 블록에서 따로 실행하지는 못한다는 점도 기억해 주세요. 먼저 오

라클 데이터베이스가 설치된 OS의 날짜를 평일로 변경해 준 후 다음과 같이 EMP_TRG 테이블에 UPDATE문을 사용해 보죠. 평일의 경우 DML 명령어가 문제없이 잘 실행되는 것을 확인할 수 있습니다.

☞ 만약에 윈도우 사용자라면 작업 표시줄의 날짜를 변경해 주면 됩니다.

실습 19-33 평일 날짜로 EMP_TRG 테이블 UPDATE하기

· 완성 파일 19-33.sql

```
01 UPDATE emp_trg SET sal = 3500 WHERE empno = 7788;
```

:: 결과 화면

1행이 갱신되었습니다.

이번에는 날짜를 토요일이나 일요일로 변경해서 실행해 보겠습니다. 트리거 내부에서 주말에 DML 명령어가 실행되면 오류를 발생시키고 있으므로 오류 메시지와 함께 DML 명령어의 실행이 취소됨을 알 수 있습니다.

실습 19-34 주말 날짜에 EMP_TRG 테이블 UPDATE하기

· 완성 파일 19-34.sql

```
01 UPDATE emp_trg SET sal = 3500 WHERE empno = 7788;
```

:: 결과 화면

```
UPDATE emp_trg SET sal = 3500 WHERE empno = 7788
```

```
*
```

1행에 오류:

ORA-20001: 주말 사원정보 수정 불가

ORA-06512: "SCOTT.TRG_EMP_NODML_WEEKEND", 6행

ORA-04088: 트리거 'SCOTT.TRG_EMP_NODML_WEEKEND'의 수행시 오류

DML 트리거의 제작 및 사용(AFTER)

이번에는 DML 명령어가 실행된 후 작동하는 AFTER 트리거를 제작해 봅시다. 앞에서 생성한 EMP_TRG 테이블에 DML 명령어가 실행될 경우 테이블에 수행된 DML 명령어의 종류, DML을 실행시킨 사용자, DML 명령어가 수행된 날짜와 시간을 저장할 EMP_TRG_LOG 테이블을 다음과 같이 생성해 주세요.

실습 19-35 EMP_TRG_LOG 테이블 생성하기

· 완성 파일 19-35.sql

```
01 CREATE TABLE EMP_TRG_LOG(
02     TABLENNAME VARCHAR2(10),    -- DML이 수행된 테이블 이름
03     DML_TYPE VARCHAR2(10),      -- DML 명령어의 종류
```

```
04    EMPNO NUMBER(4),          -- DML 대상이 된 사원 번호
05    USER_NAME VARCHAR2(30),   -- DML을 수행한 USER 이름
06    CHANGE_DATE DATE        -- DML이 수행된 날짜
07  );
```

:: 결과 화면

테이블이 생성되었습니다.

그리고 EMP_TRG 테이블에 DML 명령어를 수행한 후 EMP_TRG_LOG 테이블에 EMP_TRG 테이블 데이터의 변경 사항을 기록하는 트리거를 생성합니다.

실습 19-36 DML 실행 후 수행할 트리거 생성하기

· 완성 파일 19-36.sql

```
01  CREATE OR REPLACE TRIGGER trg_emp_log
02  AFTER
03  INSERT OR UPDATE OR DELETE ON EMP_TRG
04  FOR EACH ROW
05
06  BEGIN
07
08  IF INSERTING THEN
09      INSERT INTO emp_trg_log
10      VALUES ('EMP_TRG', 'INSERT', :new.empno,
11              SYS_CONTEXT('USERENV', 'SESSION_USER'), sysdate);
12
13  ELSIF UPDATING THEN
14      INSERT INTO emp_trg_log
15      VALUES ('EMP_TRG', 'UPDATE', :old.empno,
16              SYS_CONTEXT('USERENV', 'SESSION_USER'), sysdate);
17
18  ELSIF DELETING THEN
19      INSERT INTO emp_trg_log
20      VALUES ('EMP_TRG', 'DELETE', :old.empno,
21              SYS_CONTEXT('USERENV', 'SESSION_USER'), sysdate);
22  END IF;
23 END;
24 /
```

:: 결과 화면

트리거가 생성되었습니다.

- 02행** DML 명령어를 사용한 후 실행할 AFTER 트리거를 정의합니다.
- 03행** EMP_TRG 테이블을 대상으로 INSERT, UPDATE, DELETE 명령어를 실행한 후 트리거가 작동합니다.
- 04행** FOR EACH ROW 키워드를 통해 DML 명령어의 대상이 되는 행별로 트리거가 작동합니다.
- 08행** INSERT 명령어를 실행했을 때를 뜻합니다.
- 09행** EMP_TRG_LOG 테이블에 EMP_TRG 테이블의 INSERT 실행 내역을 저장합니다. :new. empno는 새로 추가된 empno 값을 뜻합니다. SYS_CONTEXT('USERENV', 'SESSION_USER')는 현재 데이터베이스에 접속 중인 사용자를 의미합니다.
- 13행** UPDATE 명령어가 실행되었을 때를 뜻합니다.
- 14행** EMP_TRG_LOG 테이블에 EMP_TRG 테이블의 UPDATE 실행 내역을 저장합니다. :old. empno는 변경 전 empno 값을 의미합니다.
- 18행** DELETE 명령어가 실행되었을 때를 뜻합니다.
- 19행** EMP_TRG_LOG 테이블에 EMP_TRG 테이블의 DELETE 실행 내역을 저장합니다.

이제 EMP_TRG 테이블에 DML 명령어를 사용한 후 EMP_TRG_LOG 테이블의 변화를 살펴볼까요? 먼저 EMP_TRG 테이블에 INSERT문을 사용하여 새로운 사원을 추가해 보죠.

실습 19-37 | EMP_TRG 테이블에 INSERT 실행하기

• 완성 파일 19-37.sql

```
01  INSERT INTO EMP_TRG  
02    VALUES(9999, 'TestEmp', 'CLERK', 7788,  
03          TO_DATE('2018-03-03', 'YYYY-MM-DD'), 1200, null, 20);
```

:: 결과 확인

1개 행이 만들어졌습니다.

☞ trg_emp_nodml_weekend 트리거가 등록되어 있는 상태라면 주말 날짜에는 위 INSERT문이 실행되지 않습니다.

실습 19-38 | EMP_TRG 테이블에 INSERT 실행하기(COMMIT하기)

• 완성 파일 19-38.sql

```
01  COMMIT;
```

:: 결과 확인

커밋이 완료되었습니다.

결과를 살펴보면 EMP_TRG 테이블에 사원이 추가되었음을 확인할 수 있고 EMP_TRG_LOG 테이블에는 EMP_TRG 테이블에 INSERT가 실행된 내용이 기록되어 있습니다.

☞ 가독성을 위해 이번 예제는 SELECT문을 토드에서 실행한 결과를 사용합니다.

실습 19-39 EMP_TRG 테이블의 INSERT 확인하기

· 완성 파일 19-39.sql

```
01  SELECT *
02    FROM EMP_TRG;
```

:: 결과 화면

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	1980-12-17	800		20
7499	ALLEN	SALESMAN	7698	1981-02-20	1600	300	30
7521	WARD	SALESMAN	7698	1981-02-22	1250	500	30
7566	JONES	MANAGER	7839	1981-04-02	2975		20
7654	MARTIN	SALESMAN	7698	1981-09-28	1250	1400	30
7698	BLAKE	MANAGER	7839	1981-05-01	2850		30
7782	CLARK	MANAGER	7839	1981-06-09	2450		10
7788	SCOTT	ANALYST	7566	1987-04-19	3000		20
7839	KING	PRESIDENT		1981-11-17	5000		10
7844	TURNER	SALESMAN	7698	1981-09-08	1500	0	30
7876	ADAMS	CLERK	7788	1987-05-23	1100		20
7900	JAMES	CLERK	7698	1981-12-03	950		30
7902	FORD	ANALYST	7566	1981-12-03	3000		20
7934	MILLER	CLERK	7788	1982-01-23	1300		10
9999	TestEmp	CLERK	7788	2018-03-03	1200		20

실습 19-40 EMP_TRG_LOG 테이블의 INSERT 기록 확인하기

· 완성 파일 19-40.sql

```
01  SELECT *
02    FROM EMP_TRG_LOG;
```

:: 결과 화면

TABLENAME	DML_TYPE	EMPNO	USER_NAME	CHANGE_DATE
EMP_TRG	INSERT	9999	SCOTT	2018-07-13 오후 10:50:49

이번에는 UPDATE문을 사용해 보죠. MGR이 7788로 지정된, 즉 SCOTT이 상급자인 사원들을 변경해 보죠.

실습 19-41 EMP_TRG 테이블에 UPDATE 실행하기

· 완성 파일 19-41.sql

```
01  UPDATE EMP_TRG
02    SET SAL = 1300
03  WHERE MGR = 7788;
```

:: 결과 화면

2행이 갱신되었습니다.

☞ 실습 19-38과 마찬가지로 trg_emp_nodml_weekend 트리거가 등록되어 있는 상태이고 실습한 날짜가 주말이라면 UPDATE문 역시 실행되지 않습니다.

실습 19-41 EMP_TRG 테이블에 UPDATE 실행하기(COMMIT하기)

· 완성 파일 19-41.sql

01 COMMIT;

:: 결과 화면

커밋이 완료되었습니다.

SCOTT이 상급인 사원은 바로 전에 INSERT문을 통해 추가한 TestEmp와 기존에 존재하고 있던 ADAMS이므로 두 사원의 급여 데이터가 변경되었습니다. 두 개 행이 DML문에 영향을 받았으므로 트리거에 지정한 FOR EACH ROW 옵션으로 트리거는 두 번 실행합니다. 즉 변경된 TestEmp, ADAMS의 UPDATE 정보가 각각 EMP_TRG_LOG 테이블에 저장되는 것 이죠.

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
▶	7369	SMITH	CLERK	7902	1980-12-17	800		20
	7499	ALLEN	SALESMAN	7698	1981-02-20	1600	300	30
	7521	WARD	SALESMAN	7698	1981-02-22	1250	500	30
	7566	JONES	MANAGER	7839	1981-04-02	2975		20
	7654	MARTIN	SALESMAN	7698	1981-09-28	1250	1400	30
	7698	BLAKE	MANAGER	7839	1981-05-01	2850		30
	7782	CLARK	MANAGER	7839	1981-06-09	2450		10
	7788	SCOTT	ANALYST	7566	1987-04-19	3000		20
	7839	KING	PRESIDENT		1981-11-17	5000		10
	7844	TURNER	SALESMAN	7698	1981-09-08	1500	0	30
	7876	ADAMS	CLERK	7788	1987-05-23	1300		20
	7900	JAMES	CLERK	7698	1981-12-03	950		30
	7902	FORD	ANALYST	7566	1981-12-03	3000		20
	7934	MILLER	CLERK	7782	1982-01-23	1300		10
	9999	TestEmp	CLERK	7788	2018-03-03	1300		20

	TABLENAME	DML_TYPE	EMPNO	USER_NAME	CHANGE_DATE
▶	EMP_TRG	INSERT	9999	SCOTT	2018-07-13 오후 10:50:49
	EMP_TRG	UPDATE	7876	SCOTT	2018-07-13 오후 10:54:43
	EMP_TRG	UPDATE	9999	SCOTT	2018-07-13 오후 10:54:43

비교적 간단한 형태의 트리거를 구현해 보았는데 이 예제를 통해 트리거가 어떤 역할을 하는 서브프로그램인지 이해했다면 여기에서는 충분합니다. 오라클에서는 예제에서 소개한 방식 외에도 여러 가지 방식의 트리거를 정의할 수 있으므로 오라클 공식 문서나 인터넷 검색 그리고 PL/SQL 전문 서적 등의 자료를 보고 익혀 가길 바랍니다.

트리거 관리

트리거 정보 조회

트리거 정보를 확인하려면 USER_TRIGGERS 데이터 사전을 조회합니다.

실습 19-42 USER_TRIGGERS로 트리거 정보 조회하기

· 완성 파일 19-42.sql

```
01  SELECT TRIGGER_NAME, TRIGGER_TYPE, TRIGGERING_EVENT, TABLE_NAME, STATUS  
02      FROM USER_TRIGGERS;
```

:: 결과 화면

TRIGGER_NAME	TRIGGER_TYPE	TRIGGERING_EVENT	TABLE_NAME	STATUS
▶ TRG_EMP_LOG	AFTER EACH ROW	INSERT OR UPDATE OR DELETE	EMP_TRG	ENABLED
TRG_EMP_NODML_WEEKEND	BEFORE STATEMENT	INSERT OR UPDATE OR DELETE	EMP_TRG	ENABLED

트리거 변경

ALTER TRIGGER 명령어로 트리거 상태를 변경할 수 있습니다. 특정 트리거를 활성화 또는 비활성화하려면 ALTER TRIGGER 명령어에 ENABLE 또는 DISABLE 옵션을 지정합니다.

ALTER TRIGGER 트리거 이름 ENABLE | DISABLE;

특정 테이블과 관련된 모든 트리거의 상태를 활성화하거나 비활성화하는 것도 가능한데요. 이 경우는 ALTER TABLE 명령어를 사용합니다.

특정 테이블과 관련된 모든 트리거의 상태 활성화

ALTER TABLE 테이블 이름 ENABLE ALL TRIGGERS;

특정 테이블과 관련된 모든 트리거의 상태 비활성화

ALTER TABLE 테이블 이름 DISABLE ALL TRIGGERS;

트리거 삭제

다른 오라클 객체와 마찬가지로 DROP문을 사용하여 트리거를 삭제할 수 있습니다.

DROP TRIGGER 트리거 이름;

Q1 다음과 같은 결과가 나오도록 내용을 작성해 보세요.

- ① DEPT 테이블의 부서 번호(DEPTNO)를 입력 값으로 받은 후 부서 번호(DEPTNO), 부서 이름(DNAME), 지역(LOC)을 출력하는 프로시저 pro_dept_in을 작성해 보세요.

② pro_dept_in 프로시저를 통해 출력된 부서 번호(DEPTNO), 부서 이름(DNAME), 지역(LOC)을 다음과 같이 출력하는 PL/SQL 프로그램을 작성해 보세요.

```
SQL> SET SERVEROUTPUT ON;
SQL> CREATE OR REPLACE PROCEDURE pro_dept_in
  (
    ...
    ... [변수 선언]
  )
IS
BEGIN
  ...
  ... [PL/SQL 작성]
END pro_dept_in;
/
```

프로시저가 생성되었습니다.

```
SQL> DECLARE
      ...
      ... [변수 선언]
    BEGIN
      ...
      ... [pro_dept_in 프로시저를 사용하는 PL/SQL 작성]
    END;
  /
부서 번호 : 10
부서 이름 : ACCOUNTING
지역 : NEW YORK
```

● 이 장에서 배운 내용을 실습하며 정리하세요.

Q2 다음과 같은 결과가 나오도록 내용을 작성해 보세요.

SELECT문에서 사용할 수 있는 함수 func_date_kor를 작성합니다. func_date_kor 함수는 DATE 자료형 데이터를 입력받아 다음과 같이 YYYY년MM월DD일 형태의 데이터를 출력합니다.

```
SQL> SET SERVEROUTPUT ON;
SQL> CREATE OR REPLACE FUNCTION func_date_kor(
    ... [변수 선언]
)
    ... [반환 값 설정]
IS
BEGIN
    ... [PL/SQL 작성]
END func_date_kor;
/
```

함수가 생성되었습니다.

```
SQL> SELECT ENAME, func_date_kor(HIREDATE) AS HIREDATE
2      FROM EMP
3     WHERE EMPNO = 7369;
```

ENAME

HIREDATE

SMITH

1980년12월17일

Q3 다음과 같은 결과가 나오도록 내용을 작성해 보세요.

- ① DEPT 테이블과 같은 열 구조 및 데이터를 가진 DEPT_TRG 테이블을 작성해 보세요.
- ② DEPT_TRG 테이블에 DML 명령어를 사용한 기록을 저장하는 DEPT_TRG_LOG 테이블을 다음과 같이 작성해 보세요.

DEPT_TRG_LOG 테이블

열이름	자료형	길이	설명
TABLENAME	가변형 문자열	10	DML을 수행한 테이블 이름
DML_TYPE	가변형 문자열	10	DML 명령어 종류
DEPTNO	정수형 숫자	2	DML 대상 부서 번호
USER_NAME	가변형 문자열	30	DML을 수행한 USER 이름
CHANGE_DATE	날짜	-	DML을 수행한 날짜

- ③ DEPT_TRG 테이블에 DML명령 수행 기록을 DEPT_TRG_LOG에 저장하는 트리거 TRG_DEPT_LOG를 작성해 보세요.

TRG_DEPT_LOG 트리거 작성

```
SQL> SET SERVEROUTPUT ON;
SQL> CREATE OR REPLACE TRIGGER trg_dept_log
      ... [트리거 작성]
      END;
      /

```

트리거가 생성되었습니다.

```
SQL>
```

DEPT_TRG 테이블의 DML 명령어 사용 및 DEPT_TRG_LOG 테이블 결과 확인

```
SQL> INSERT INTO DEPT_TRG VALUES(99, 'TEST_DNAME', 'SEOUL');
```

1개 행이 만들어졌습니다.

```
SQL> UPDATE DEPT_TRG SET LOC='TEST_LOC' WHERE DEPTNO = 99;
```

1행이 갱신되었습니다.

```
SQL> DELETE FROM DEPT_TRG WHERE DEPTNO = 99;
```

1행이 삭제되었습니다.

```
SQL> SELECT * FROM DEPT_TRG;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

```
SQL> SELECT * FROM DEPT_TRG_LOG;
```

TABLENAME	DML_TYPE	DEPTNO	USER_NAME	CHANGE_D
DEPT_TRG	INSERT	99	SCOTT	18/03/27
DEPT_TRG	UPDATE	99	SCOTT	18/03/27
DEPT_TRG	DELETE	99	SCOTT	18/03/27

정답 이지스파클리싱 홈페이지에서 확인하세요.