Tibero

tbPSM 안내서

Tibero 7

대한민국 경기도 성남시 분당구 황새울로258번길 29, BS 타워 9층 우)13595

Website

http://www.tmaxtibero.com

기술서비스센터

Tel : +82-1544-8629

E-Mail : info@tmax.co.kr

Open Source Software Notice

Some modules or files of this product are subject to the terms of the following licenses. : OpenSSL, RSA Data Security, Inc., Apache Foundation, Jean-loup Gailly and Mark Adler, Paul Hsieh's hash

Detailed Information related to the license can be found in the following directory : ${INSTALL_PATH}/license/oss_licenses

본 제품의 일부 파일 또는 모듈은 다음의 라이선스를 준수합니다. : OpenSSL, RSA Data Security, Inc., Apache Foundation, Jean-loup Gailly and Mark Adler, Paul Hsieh's hash

관련 상세한 정보는 제품의 다음의 디렉터리에 기재된 사항을 참고해 주십시오. : ${INSTALL_PATH}/license/oss_licenses

안내서 정보

안내서 제목: Tibero tbPSM 안내서

발행일: 2024-08-22

소프트웨어 버전: Tibero 7.2.2

안내서 버전: v7.2.2

내용 목차

안내서에 대하여

안내서의 대상

본 안내서는 Tibero®(이하 Tibero)에서 제공하는 저장 프러시저 모듈 즉, tbPSM(Tibero의 Persistent Stored Module)을 참고하려는 데이터베이스 관리자(Database Administrator, 이하 DBA), 애플리케이션 프로그램 개발자를 대상으로 기술한다.

안내서의 전제 조건

본 안내서를 원활히 이해하기 위해서는 다음과 같은 사항을 미리 알고 있어야 한다.

- 데이터베이스의 이해
- RDBMS의 이해
- SQL의 이해
- PSM의 이해

안내서의 제한 조건

본 안내서는 Tibero를 실무에 적용하거나 운용하는 데 필요한 모든 사항을 포함하고 있지 않다. 따라서 설치, 환경설정 등 운용 및 관리에 대해서는 각 제품 안내서를 참고하기 바란다.

참고

Tibero의 설치 및 환경설정에 관한 내용은 "Tibero 설치 안내서"를 참고한다.

안내서 구성

Tibero tbPSM 안내서는 총 11개의 장과 Appendix로 이루어져 있다. 각 장의 주요 내용은 다음과 같다.

- 제1장: tbPSM 소개

  tbPSM의 기본 개념과 구성요소, 프로그램 구조를 간략히 소개한다.

- 제2장: tbPSM의 문법

  tbPSM에서 사용하는 tbPSM 문장의 기본적인 문법과 데이터 타입을 기술한다.

- 제3장: 제어 구조

  tbPSM에서 제공하는 IF, CASE, LOOP 등의 제어 구조를 기술한다.

- 제4장: 복합 타입

  tbPSM에서 제공하는 구조체 형태의 콜렉션 타입과 레코드를 기술한다.

- 제5장: 서브 프로그램

  tbPSM 프로그램 내에서 호출할 수 있는 프로그램 블록인 서브 프로그램을 기술한다.

- 제6장: 패키지

  tbPSM의 변수나 타입, 서브 프로그램 등을 그룹화하여 모아 놓은 객체인 패키지를 기술한다.

- 제7장: SQL 문장의 실행

  tbPSM 프로그램에서 SQL 문장을 실행하는 방법을 기술한다.

- 제8장: 에러 처리

  tbPSM 프로그램에서 발생하는 에러를 처리하는 방법을 기술한다.

- 제9장: 파이프라인드 테이블 함수

  파이프라인드 방식으로 데이터를 처리할 수 있는 파이프라인드 테이블 함수를 기술한다.

- 제10장: 오브젝트 타입

  추상 데이터 타입인 오브젝트 타입을 기술한다.

- 제11장: 사용자 정의 AGGREGATION 함수

  사용자가 정의 가능한 AGGREGATION 함수를 기술한다.

- 제12장: BULK SQL

  대량의 SQL 처리에 대하여 기술한다.

- Appendix A: 예약어

  tbPSM에서 사용하는 예약어를 기술한다.

- Appendix B: PSM 소스코드 암호화

  tbPSM로 작성된 소스코드를 암호화하여 배포하는 방법을 기술한다.

안내서 규약

표기	의미
<<AaBbCc123>>	프로그램 소스 코드의 파일명
<Ctrl>+C	Ctrl과 C를 동시에 누름
[Button]	GUI의 버튼 또는 메뉴 이름
진하게	강조
" "(따옴표)	다른 관련 안내서 또는 안내서 내의 다른 장 및 절 언급
'입력항목'	화면 UI에서 입력 항목에 대한 설명
하이퍼링크	메일 계정, 웹 사이트
>	메뉴의 진행 순서
+----	하위 디렉터리 또는 파일 있음
|----	하위 디렉터리 또는 파일 없음
참고	참고 또는 주의사항
주의	주의할 사항
[그림 1.1]	그림 이름
[예 1.1]	예제 이름
`AaBbCc123`	Java 코드, XML 문서
[*command argument*]	옵션 파라미터
< xyz >	'<'와 '>' 사이의 내용이 실제 값으로 변경됨
|	선택 사항. 예) A|B: A나 B 중 하나
…	파라미터 등이 반복되어서 나옴
${ }	환경변수

시스템 사용 환경

	요구 사항
Platform	HP-UX 11i v3(11.31)
	Solaris (Solaris 11)
	AIX (AIX 7.1/AIX 7.2/AIX 7.3)
	GNU (X86, 64, IA64)
	Red Hat Enterprise Linux 7 kernel 3.10.0 이상
	Windows(x86) 64bit
Hardware	최소 2.5GB 하드디스크 공간
	1GB 이상 메모리 공간
Compiler	PSM (C99 지원 필요)
	tbESQL/C (C99 지원 필요)

관련 안내서

안내서	설명
Tibero 설치 안내서	설치 과정에 필요한 시스템 요구사항과 설치 및 제거 방법을 기술한 안내서이다.
Tibero tbCLI 안내서	Call Level Interface인 tbCLI의 개념과 구성요소, 프로그램 구조를 소개하고 tbCLI 프로그램을 작성하는 데 필요한 데이터 타입, 함수, 에러 메시지를 기술한 안내서이다.
Tibero 애플리케이션 개발자 안내서	각종 애플리케이션 라이브러리를 이용하여 애플리케이션 프로그램을 개발하는 방법을 기술한 안내서이다.
Tibero External Procedure 안내서	External Procedure를 소개하고 이를 생성하고 사용하는 방법을 기술한 안내서이다.
Tibero JDBC 개발자 안내서	Tibero에서 제공하는 JDBC 기능을 이용하여 애플리케이션 프로그램을 개발하는 방법을 기술한 안내서이다.
Tibero tbESQL/C 안내서	C 프로그래밍 언어를 사용해 데이터베이스 작업을 수행하는 각종 애플리케이션 프로그램을 작성하는 방법을 기술한 안내서이다.
Tibero tbESQL/COBOL 안내서	COBOL 프로그래밍 언어를 사용해 데이터베이스 작업을 수행하는 각종 애플리케이션 프로그램을 작성하는 방법을 기술한 안내서이다.
Tibero tbPSM 참조 안내서	저장 프러시저 모듈인 tbPSM의 패키지를 소개하고, 이러한 패키지에 포함된 각 프러시저와 함수의 프로토타입, 파라미터, 예제 등을 기술한 참조 안내서이다.
Tibero 관리자 안내서	Tibero의 동작과 주요 기능의 원활한 수행을 보장하기 위해 DBA가 알아야 할 관리 방법을 논리적 또는 물리적 측면에서 설명하고, 관리를 지원하는 각종 도구를 기술한 안내서이다.
Tibero 유틸리티 안내서	데이터베이스와 관련된 작업을 수행하기 위해 필요한 유틸리티의 설치 및 환경설정, 사용 방법을 기술한 안내서이다.
Tibero TAS 안내서	Tibero Active Storage(TAS)를 사용해서 Tibero 의 파일을 관리하고자 하는 관리자를 대상으로 기술한 안내서이다.
Tibero 에러 참조 안내서	Tibero를 사용하는 도중에 발생할 수 있는 각종 에러의 원인과 해결 방법을 기술한 안내서이다.

안내서	설명
Tibero 참조 안내서	Tibero의 동작과 사용에 필요한 초기화 파라미터와 데이터 사전, 정적 뷰, 동적 뷰를 기술한 참조 안내서이다.
Tibero SQL 참조 안내서	데이터베이스 작업을 수행하거나 애플리케이션 프로그램을 작성할 때 필요한 SQL 문장을 기술한 참조 안내서이다.
Tibero Spatial 참조 안내서	Tibero에서 Geometry 타입에 대한 설명과 Spatial 기능 관련 프러시저 함수 목록 및 사용 방법 등을 기술한 안내서이다.
Tibero TEXT 참조 안내서	Tibero의 제공하는 Text Index를 소개하고, Text Index를 생성 하고 사용하는 방법을 기술하는 안내서이다.
Tibero TDP.NET 안내서	Tibero Data Provider for .NET 기능을 기술하는 안내서이다.
Tibero IMCS 안내서	Tibero에서 제공하는 In-Memory Column Store(이하 IMCS) 기능을 기술하는 안내서이다.

제1장 tbPSM 소개

본 장에서는 tbPSM의 기본 개념과 구성요소 그리고 tbPSM 프로그램의 구조를 소개한다.

1.1. 개요

SQL은 비절차적(Non-procedural) 언어로 데이터베이스의 모든 작업을 통제한다. SQL은 대체로 간단하고 적은 수의 명령어로 구성되어 있다. 따라서 일반적으로 사용자는 SQL의 기초가 되는 데이터 구조와 알고리즘을 몰라도 자유롭게 SQL 명령을 실행할 수 있다. SQL의 장점에도 불구하고 사용자가 원하는 프로그램을 작성하려고 할 때 여전히 불편함이 따른다.

tbPSM은 Tibero에서 제공하는 PSM 프로그램 언어 및 실행 시스템이다. 순차적으로 원하는 결과를 얻어야 하는 프로그램을 SQL 문장만으로 작성할 수 없다는 제약 때문에 tbPSM이 필요하다.

참고

PSM(Persistent Stored Modules)은 절차적 기능을 갖춘 확장된 SQL를 생성하는 강력한 프로그래밍 언어이다. 그 기능이 매우 단순하고 강력하여 사용자에 의해 널리 사용되고 있다. 또한 PSM은 데이터베이스 프로그래밍을 위한 ISO 표준에 기반을 두고 있으며, SQL 명령을 사용하여 데이터베이스와 상호 연동을 할 수 있다.

---

tbPSM은 SQL 문장을 제어 구조(예: if...then...else..., loop) 등에 추가하여 프로그램을 절차적으로 구성할 수 있다. SQL의 비절차적인 구조의 문제점을 해결하여 사용자가 원하는 프로그램을 작성할 수 있다.

빈번하게 수행되는 작업을 데이터베이스 시스템에 컴파일 된 상태로 저장함으로써, 매번 해당 질의를 다시 컴파일해야 하는 부담을 덜어준다. 뿐만 아니라 tbPSM 내부에 IF, WHILE 등과 같은 제어 문장과 변수 등을 활용할 수 있기 때문에 복잡한 작업의 수행을 위해 애플리케이션 프로그램과 데이터베이스 서버 간의 빈번한 통신이 발생하는 것을 줄여 준다. 이 기능은 널리 사용되고 있는 PL/SQL 기능과 대등하다.

tbPSM은 다음과 같은 절차형 언어(제3세대 언어)의 성격을 SQL에 추가함으로써 사용자에게 편의를 제공한다.

- 변수와 타입
- 제어 구조
- 프러시저와 함수

1.2. 구성요소

본 절에서는 tbPSM 프로그램을 작성하거나 실행하기에 앞서 기본적으로 알아야 할 구성요소를 설명한다.

1.2.1. 변수와 상수

본 절에서는 변수와 상수를 간략히 설명한다.

변수

변수는 프로그램을 작성할 때 값을 나타내는 문자나 문자들의 집합이다.

변수를 선언하는 방법은 다음과 같다.

```
변수이름 변수의 타입[제약조건] [기본값 정의]
```

항목	설명
변수이름	값을 나타내는 문자나 문자 세트의 이름이다. – 변수이름을 먼저 입력한다. – 변수이름 뒤에 변수의 타입과 제약조건이 따른다.
변수의 타입	데이터베이스 고유의 타입뿐만 아니라 tbPSM 프로그램에서 전용으로 사용하는 타입도 포함된다.
제약조건	데이터 타입에 따라 사용 여부가 달라진다.
기본값 정의	정의된 변수에 값이 할당되지 않은 경우에는 정의된 기본값을 사용한다.

다음은 변수를 선언하는 예이다.

```
id NUMBER;

productname VARCHAR2(20) := 'Tibero';
```

변수를 할당하는 방법은 콜론(:)과 등호(=)를 조합하여 이루어진다.

```
[할당 받는 대상] := [할당하는 대상]
```

항목	설명
할당 받는 대상	반드시 변수가 와야 한다.
:=	이 기호를 기준으로 왼쪽에 위치한 것은 할당 받는 대상이고 오른쪽은 할당하는 대상이다.

항목	설명
할당하는 대상	변수, 표현 식, 함수의 결과 등이 올 수 있으며 대상의 제약이 없다.

다음은 변수를 할당하는 예이다.

```
gravity := 9.8;

force := mass * acceleration;

grade := calculate_the_grade('Darwin');
```

상수

상수는 프로그램이 수행되는 동안은 변하지 않는 값을 나타내는 데이터로 숫자, 문자, 문자열 등이 있다.

상수를 선언하는 방법은 다음과 같다.

```
변수이름 CONSTANT 변수의 타입 := [기본값 정의]
```

항목	설명
변수이름	변수이름이 제일 먼저 위치한다.
CONSTANT	상수의 선언은 CONSTANT라는 예약어를 사용하면 된다.
변수의 타입	변수의 데이터 타입을 설정한다.
기본값 정의	정의된 상수에 값을 할당한다. 상수로 선언된 변수에는 어떠한 값도 다시 할당할 수 없다.

다음은 상수를 선언하고 특정 값을 할당하는 예이다.

```
PI CONSTANT NUMBER := 3.141592;
```

1.2.2. 제어 구조

tbPSM은 프로그램 실행 중에 발생되는 조건에 따라 특정 작업을 수행하거나 반복적인 작업을 수행하는 제어 구조(control structure)를 제공한다. 특정 작업을 수행하는 조건 구조(conditional structure)와 반복 작업을 수행하는 반복 구조(iterative structure)를 위한 다양한 명령어를 제공한다. 자세한 내용은 “제3장 제어 구조”를 참고한다.

tbPSM의 제어 구조는 다음과 같이 구분할 수 있다.

- 선택적(selective) 제어 구조

IF 문과 CASE 문이 포함된다.

- 반복 제어 구조(iterative structure)

  LOOP 문, WHILE 문, FOR 문이 포함된다.

다음은 IF 문의 예이다.

```
DECLARE
    evaluation_score NUMBER;
BEGIN
    SELECT bonus into evaluation_score FROM employee WHERE emp_no = 19963077;
    IF evaluation_score > 80 THEN
        DBMS_OUTPUT.PUT_LINE('This person gets a salary bonus.');
    ELSE
        DBMS_OUTPUT.PUT_LINE('This person does not get a salary bonus.');
    END IF;
END;
```

1.2.3. 서브 프로그램과 패키지

본 절에서는 서브 프로그램과 패키지를 간략히 설명한다.

서브 프로그램

서브 프로그램(subprogram)은 일반적인 프로그래밍에서의 함수와 동일한 기능을 수행한다.

기본적으로 프러시저(procedure)와 함수(function)를 제공하는데, 이 둘의 차이점은 프러시저는 반환값이 없고, 함수는 반환값이 있다는 것이다. 자세한 내용은 “제5장 서브 프로그램”을 참고한다.

서브 프로그램은 tbPSM의 프로그램 구조와 마찬가지로 선언부, 실행부, 예외 처리부로 구성된다.

다음은 서브 프로그램을 작성한 예이다.

```
CREATE [OR REPLACE] PROCEDURE check_sal(name VARCHAR2) IS
    sal NUMBER;
    invalid_id exception;
BEGIN
    SELECT registered_num INTO sal FROM emp WHERE emp_name = name;
    IF sal < 7000 THEN
        raise invalid_sal;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Correct');
EXCEPTION
    WHEN invalid_id THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Invalid registration number');
END;
```

패키지

패키지(package)는 서브 프로그램과 변수, 상수의 집합이라 할 수 있다. 즉, 관련된 기능을 그룹으로 묶어 사용할 수 있어 프로그램을 관리하기가 쉽다. 또한 패키지는 사용하는 시점에 패키지의 구성요소가 일시에 메모리에 로드되므로 특정 작업을 수행할 때 데이터베이스 서버의 성능을 향상하는 데 도움이 되기도 한다. 모듈 간의 의존성 축소화(dependency minimizing)를 위해서도 사용된다. 자세한 내용은 “제6장 패키지”를 참고한다.

패키지는 선언부(specification)와 구현부(body)로 구성된다.

- 선언부

  패키지에서 사용할 함수와 프러시저의 이름 그리고 변수, 상수, 타입, 예외 상황, 커서의 선언 정보가 포함된다.

  다음은 선언부의 작성 예이다.

```
CREATE [OR REPLACE] PACKAGE book_manager IS
    book_cnt NUMBER;
    PROCEDURE add_new_book(author VARCHAR2, name VARCHAR2, publish_year DATE);
    PROCEDURE remove_lost_book(author VARCHAR2, name VARCHAR2);
    FUNCTION search_book_position(author VARCHAR2, name VARCHAR2)
             RETURN NUMBER;
    FUNCTION get_total_book_cnt RETURN NUMBER;
END;
```

- 구현부

  선언된 프러시저나 함수가 실제로 구현되는 부분이다.

  다음은 구현부의 작성 예이다.

```
CREATE [OR REPLACE] PACKAGE BODY book_manager IS
    PROCEDURE add_new_book(v_author VARCHAR2, v_name VARCHAR2,
                           publish_year DATE) IS
    BEGIN
        IF substr(v_name, 1, 1) >= 'a' AND
           substr(v_name, 1, 1) < 'k' THEN
            INSERT INTO books
                VALUES (1, v_author, v_name, publish_year);
        ELSE
            INSERT INTO books
                VALUES (2, v_author, v_name, publish_year);
        END IF;
```

```
        COMMIT;
        book_cnt := book_cnt + 1;
    END;

    PROCEDURE remove_lost_book(v_author VARCHAR2, v_name VARCHAR2) IS
    BEGIN
        DELETE FROM books WHERE author = v_author AND name = v_name;
        COMMIT;
    END;

    FUNCTION search_book_position(v_author VARCHAR2, v_name VARCHAR2)
    RETURN NUMBER IS
        book_position NUMBER;
    BEGIN
        SELECT kind INTO book_position FROM books
            WHERE author = v_author AND name = v_name;
        RETURN book_position;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('NOT EXIST...SORRY');
            RAISE;
    END;

    FUNCTION get_total_book_cnt RETURN NUMBER IS
    BEGIN
        RETURN book_cnt;
    END;
BEGIN
    book_cnt := 0;
END;
```

1.2.4. 커서

커서(cursor)는 SQL 문장을 처리한 후 얻은 결과 집합에 대한 포인터이다.

데이터베이스에서 탐색된 여러 로우를 처리 하기 위해 사용되며, 프로그램은 커서를 사용하여 질의(query)의 결과로 생성된 집합을 한 번에 한 로우씩 검사하고 처리할 수 있다. 커서는 질의의 결과와 반환되는 결과 집합이 매우 크거나 크기를 예상할 수 없는 경우에 유용하게 사용된다.

tbPSM은 커서와 커서의 변수를 제공한다. 자세한 내용은 “7.3. 커서”를 참고한다.

다음은 커서를 사용한 예이다.

```
DECLARE
    EMP_NO VARCHAR(8);
```

```
    CURSOR c1 IS SELECT EMP_NO FROM employee;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO  EMP_NO;
        EXIT WHEN c1%NOTFOUND;
    END LOOP;
END;
```

1.2.5. 에러 처리

tbPSM 프로그램은 실행 중에 에러가 발생할 수 있다. 예를 들어 SELECT INTO 문을 실행했을 때 반환되는 로우가 없는 경우를 들 수 있다. 이러한 에러를 **예외 상황**이라 한다.

tbPSM 프로그램 내에서는 실행 중에 발생할 수 있는 예외 상황(exception)을 처리하기 위한 루틴을 포함시킬 수 있다. 이러한 루틴을 에러 처리 루틴(error handling routine)이라고 한다. 에러가 발생했을 때 그 에러에 대한 처리 루틴이 정의된 경우 해당 루틴을 실행하게 된다.

이러한 루틴을 통해 tbPSM 프로그램은 구조가 명확해지고 관리가 쉬워진다는 장점이 있다. 자세한 내용은 “제8장 에러 처리”를 참고한다.

다음은 에러를 처리하는 예이다.

```
DECLARE
    no_bonus exception;
    the_sales NUMBER := 12000;
BEGIN
    IF the_sales < 20000 THEN
        raise no_bonus;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Your bonus is $' || the_sales * 0.1);
EXCEPTION
    WHEN no_bonus THEN
        DBMS_OUTPUT.PUT_LINE('Sorry, increase the sales');
END;
```

1.3. 프로그램 구조

tbPSM 프로그램은 기본적으로 블록(block) 구조로 이루어져 있다. 하나의 블록은 크거나 작은 하나의 작업을 수행하는 모듈이며, 다른 블록을 포함할 수 있다.

전형적인 tbPSM의 블록은 다음과 같이 세 부분으로 구성된다.

- 선언부

DECLARE와 함께 시작되며, 블록 내에서 사용할 데이터 변수를 선언하는 부분이다.

선언부는 다음과 같은 특징이 있다.

– 생략할 수 있다.

– 선언부는 실행 코드부와 에러 처리부와는 달리 서브 블록을 포함할 수 없다.

– 실행 코드부에서 사용하는 변수가 없으면 선언부는 블록에 포함하지 않아도 된다.

- 실행 코드부

  BEGIN 예약어와 END 예약어 중간에 포함된다.

  실행 코드부는 다음과 같은 특징이 있다.

  – 데이터베이스를 액세스하기 위한 일반적인 SQL 문장과 제어 구조 문장이 포함된다.

  – 실행 코드부 내의 모든 SQL 문장과 코드는 항상 세미콜론(;)으로 끝난다.

- 에러 처리부

  BEGIN 예약어와 END 예약어 중간에 포함되며, 실행 코드부에서 발생한 에러를 처리하는 부분이다. 이러한 에러를 예외 상황이라 한다. 예외 상황에는 시스템이 미리 정의해 둔 것과 사용자가 프로그램에서 정의한 것이 있다.

다음은 tbPSM 프로그램의 구조를 나타내는 예이다.

```
DECLARE
    -- 선언부
    name varchar(32);

BEGIN
    -- 실행 코드부
    SELECT emp_name INTO name FROM employee WHERE emp_no = 100;

EXCEPTION
    -- 에러 처리부
WHEN NO_DATA_FOUND THEN
     dbms_output.put_line('employee not exist');

END;
```

위 예를 기준으로 tbPSM 프로그램이 수행되는 순서는 다음과 같다.

1. 선언부

   DECLARE와 함께 name이라는 변수를 선언한다. 데이터 타입은 varchar이며, 길이는 32이다.

2. 실행 코드부

EMPLOYEE 테이블에서 emp_no 컬럼이 100인 emp_name의 데이터를 검색한다. 검색한 결과는 선언부에서 정의한 name에 할당한다. 만약, 검색된 결과가 없으면 3번 순서로 이동한다.

3. 에러 처리부

   검색된 결과가 없을 때 처리되는 부분으로 메시지 버퍼에 **employee not exist**라는 메시지를 EOF 문자와 함께 저장한다.

제2장 tbPSM 문법

본 장에서는 tbPSM 프로그램을 작성하기 위해 필요한 tbPSM 문장의 기본적인 문법을 설명한다. 이러한 문법은 다른 프로그래밍 언어에서도 공통적으로 정의하고 있는 내용이다. 따라서 이 내용만으로도 충분히 간단한 프로그램을 작성할 수 있다.

우선 먼저 식별자(identifier), 분리자(delimiter), 상수(literal) 등과 같은 tbPSM 문장을 작성하는 데 필요한 기본적인 구성요소를 설명하고, 이를 기초로 변수 선언 및 할당 그리고 연산자와 연산식 등을 설명한다.

2.1. tbPSM 문장 구성요소

tbPSM 프로그램은 tbPSM 문장과 SQL 문장으로 구성된다. 본 절에서는 tbPSM 문장을 기준으로 설명한다.

tbPSM 프로그램에서 사용할 수 있는 문자는 다음과 같다.

- 알파벳 문자 : a~z, A~Z
- 한글 및 기타 유니코드
- 숫자 : 0~9
- 기타 문자 : ( ) [ ] { } + - * / < > = ~ ! @ # $ % ^ & ; : . ’ , ” _ | ? tab space 개행문자

하나의 문자 또는 둘 이상이 결합된 문자들이 기호(symbol)를 구성한다. tbPSM은 이러한 기호를 연산자나 분리자 등으로 사용하기도 한다.

2.1.1. 식별자

프로그램의 변수, 상수, 레이블, 커서, 함수, 패키지 등과 같은 tbPSM 프로그램의 구성요소는 각각의 이름을 갖고 있다. 이러한 이름을 **식별자**(Identifier)라고 한다. 식별자는 변수, 커서, 서브 프로그램과 같은 tbPSM 객체의 명명을 위해 사용한다.

tbPSM의 식별자는 다음과 같은 특징이 있다.

- 알파벳 문자(대문자와 소문자), 한글, 숫자, $, _, #를 사용하여 만든다.
- 알파벳 문자 또는 한글로 시작해야 한다.
- 대문자와 소문자를 구분하지 않는다.
- 최대 길이는 255bytes이다.

다음은 **유효한 식별자의 예**이다.

```
x
employee_#
room_num__Local
```

반면에 다음은 **유효하지 않은 식별자의 예**이다.

```
abc=xyz             -- 허가되지 않은 기호(예: =)를 포함한 경우
_under_departs      -- 알파벳 문자로 시작하지 않은 경우
Date Year           -- 허가되지 않은 기호(예: 공백 문자)를 포함한 경우
000_name            -- 알파벳 문자로 시작하지 않은 경우
```

또한 tbPSM은 **대문자와 소문자를 구별하지 않으므로 다음의 식별자는 모두 같은 의미**를 갖는다.

```
Employee_id
Employee_ID
employee_id
employee_ID
```

식별자 사용 예제

tbPSM의 예약어는 SQL 표준에서 정의하고 있는 예약어보다 더 많다. tbPSM의 전체 예약어는 “Appendix A. 예약어”를 참고한다.

식별자에 이러한 tbPSM의 **예약어**를 지정하여 사용하는 경우는 다음과 같다.

- 예약어를 식별자로 지정한 경우

  특별한 의미를 갖는 일부 식별자는 예약어로 지정되어 있다. tbPSM은 예약어를 특별한 의미로 사용한다. 따라서 예약어를 변수와 같이 일반적인 식별자로 지정하면 안 된다.

  ```
  DECLARE
      begin PLS_INTEGER;
  ```

  BEGIN 예약어를 식별자로 지정하면 컴파일 에러가 발생한다.

- 예약어를 식별자의 일부로 지정한 경우

  ```
  DECLARE
      original_begin PLS_INTEGER;
  ```

  BEGIN 예약어를 식별자 이름의 일부로 지정하는 경우라면 사용할 수 있다.

다음은 식별자에 **큰따옴표(" ")**를 지정하여 사용하는 예이다.

식별자는 일반적으로 공백 문자나 탭 등의 문자를 포함할 수 없으며 대소문자 구분을 하지 않는다. 그러나 특별히 이런 특징에 예외가 되는 식별자를 사용해야 하는 경우에는 큰따옴표로 식별자를 묶어 사용할 수 있다.

```
"abc=def"
"_under_departs"
"Date Year"
"000_name"
```

큰따옴표로 감싼 식별자의 최대 길이는 일반적인 식별자의 길이와 같은 255bytes이며, 감싼 부분에는 tbPSM의 문자를 모두 사용할 수 있다. 이러한 예는 tbPSM의 예약어를 식별자로 지정하여 사용할 때에도 매우 유용하다.

다음은 큰따옴표를 사용하여 **RECORD** 예약어를 식별자로 사용하는 예이다.

```
DECLARE
    tmp_record PLS_INTEGER;

BEGIN
    SELECT "RECORD" INTO tmp_record FROM record_table;
    -- RECORD는 예약어이다.
END;
```

위의 예에서 보듯이 테이블의 컬럼 이름은 데이터베이스에 저장될 때 대문자로 저장된다. 따라서 큰따옴표로 감싼 식별자를 SQL 질의에서 사용할 때에는 반드시 대문자를 사용해야 한다.

2.1.2. 분리자

분리자는 식별자를 구분하기 위해 사용한다. 일부 분리자는 연산자의 역할도 수행한다.

다음은 tbPSM의 분리자를 요약한 목록이다.

기호	의미
+	덧셈
-	뺄셈
*	곱셈
/	나눗셈
=	등호
<	부등호(보다 작다)
>	부등호(보다 크다)
(	괄호(왼쪽)
)	괄호(오른쪽)

기호	의미
;	문장의 끝 표시
%	속성 표시
,	항목 구분
.	컴포넌트 구분
@	데이터베이스 링크 표시
‘	문자열 분리자
“	인용된 문자열 분리자
:	바인드 변수 표시
**	지수
<>	같지 않다.
!=	같지 않다.
~=	같지 않다.
^=	같지 않다.
<=	크지 않다(보다 작거나 같다).
>=	작지 않다(보다 크거나 같다).
:=	대입
..	범위
||	문자열 결합
<<	레이블(왼쪽)
>>	레이블(오른쪽)
--	단일 라인 주석
/*	다중 라인(왼쪽)
*/	다중 라인(오른쪽)

2.1.3. 상수

하나 또는 그 이상의 문자를 사용하여 어떤 값 자체를 표현하는 경우를 상수(literal)라 한다. 상수는 한 번 표시되면 변하지 않는다는 특성이 있다. 상수는 숫자 상수, 문자 상수, 문자열 상수, 날짜 상수, 진리(boolean) 상수 등으로 분류된다.

2.1.4. 주석

주석(comment)은 tbPSM이 인식하지 않는 문자열로, 단일 라인 주석과 다중 라인 주석으로 설정이 가능하다. 단일 라인과 다중 라인의 주석은 한 프로그램에서 함께 사용할 수 있다.

- 단일 라인 주석

  기본적인 tbPSM의 주석은 **'--'**로 시작하고, 최초로 만나는 개행문자로 끝난다.

  다음은 단일 라인으로 설정한 주석의 예이다.

```
DECLARE
    guest_ID         BINARY_INTEGER;     -- 고객을 식별하기 위한 ID
    guest_name       VARCHAR2(100);      -- 고객의 이름, 길이는 VARCHAR2 타입의 100자리

BEGIN
    INSERT INTO add_books (ID, NAME) VALUES (guest_ID, guest_name);
    -- guest_ID와 guest_name으로 입력된 정보를
    -- "add_books" 테이블에 삽입한다.
END;
```

- 다중 라인 주석

  C 프로그래밍 언어에서 사용하는 주석도 사용할 수 있다. 이 주석은 여러 줄에 걸쳐 사용할 수 있으며, **/*...*/**의 형식으로 사용한다. 단, 중첩하여 사용할 수 없다.

  다음은 다중 라인으로 설정한 주석의 예이다.

```
DECLARE
    guest_ID         BINARY_INTEGER;    -- 고객을 식별하기 위한 ID
    guest_name       VARCHAR2(100);     -- 고객의 이름, 길이는 VARCHAR2 타입의 100자리
BEGIN
    INSERT INTO add_books (ID, NAME) VALUES (guest_ID, guest_name);
    /* guest_ID와 guest_name으로 입력된 정보를
       add_books 테이블에 삽입한다. */
END;
```

2.2. tbPSM 데이터 타입

tbPSM의 데이터 타입은 스칼라(scalar), 복합(composite), 참조(reference), 대용량 객체형(LOB) 등의 타입으로 분류할 수 있다.

복합 타입은 몇 개의 필드로 이루어지며, 각 필드는 스칼라 타입에 속하는 데이터이다. 참조 타입은 다른 타입에 대한 포인터이며, 대용량 객체형은 대용량 객체(Large object)를 지원하기 위한 타입이다.

tbPSM의 데이터 타입 중에는 특정 데이터 타입을 기반으로 정의되는 타입이 있다. 이를 **서브 타입**(subtype)이라 한다. 서브 타입은 기반이 되는 데이터 타입에 대한 연산(operation)은 변경하지 않으며, 다만 기반이 되는 데이터 타입의 값에 제약을 부여하여 생성한다.

또한 사용자는 필요에 따라 기존의 데이터 타입에 기반하여 새로운 서브 타입을 정의할 수 있다. 이를 칭할 때 사용자 정의 서브 타입이라 한다. 자세한 내용은 “2.2.6. 사용자 정의 서브 타입”을 참고한다.

다음은 타입 별 tbPSM의 데이터 타입이다.

- 스칼라 타입

그룹	서브 타입
NUMERIC	NUMBER, DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NATURAL, NATURALN, NUMERIC, REAL, POSITIVE, POSITIVEN, SMALLINT, SIGNTYPE, PLS_INTEGER, BINARY_INTEGER, BINARY_FLOAT, BINARY_DOUBLE
CHARACTER / STRING	VARCHAR2, VARCHAR, CHAR, CHARACTER, LONG, STRING, RAW, ROWID, LONG RAW, NCHAR, NVARCHAR
DATETIME / INTERVAL	DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND
BOOLEAN	BOOLEAN

- 복합 타입

그룹	서브 타입
-	RECORD, VARRAY, TABLE

- 참조 타입

그룹	서브 타입
-	REF CURSOR

- 대용량 객체형 타입

그룹	서브 타입
-	CLOB, BLOB, XMLTYPE, GEOMERTY, BFILE, JSON

- 기타 타입

그룹	서브 타입
-	%TYPE, %ROWTYPE

2.2.1. 스칼라 타입

스칼라 타입은 데이터 그대로 사용하거나 또는 간단한 타입의 변화를 통해 데이터베이스 테이블의 컬럼에 저장할 수 있는 데이터 타입을 말한다. 스칼라 타입은 데이터의 종류에 따라 NUMERIC, CHARACTER, DATETIME, BOOLEAN 그룹으로 나뉜다.

NUMERIC 그룹

NUMERIC 그룹의 데이터 타입은 정수나 실수 값을 표현한다.

- NUMBER

  이 타입은 부동소수점 숫자를 표현하며, 데이터베이스에서 사용하는 NUMBER 타입과 동일하다.

  선언하는 문법은 다음과 같다.

```
NUMBER[(P, S)];
```

옵션	설명
P	정밀도를 의미한다. 다음은 정밀도의 특징이다. – 값의 자릿수이다. – 단독으로 사용할 수 있다. – 최대 자릿수는 38이다.
S	스케일을 의미한다. 다음은 스케일의 특징이다. – 소수점 오른쪽 자릿수이다. – 단독으로 사용할 수 없다. – 최대 자릿수는 127이다.

NUMBER의 서브 타입은 다음과 같다.

서브 타입	설명
DEC	NUMBER 타입과 동일하다.
DECIMAL	NUMBER 타입과 동일하다.
DOUBLE PRECISION	정밀도와 스케일을 명시할 수 없다.
FLOAT	스케일을 명시할 수 없다.
INTEGER	최대 38자리의 정수만 허용한다.
INT	최대 38자리의 정수만 허용한다.
NUMERIC	NUMBER 타입과 동일하다.
REAL	스케일을 명시할 수 없다.
SMALLINT	최대 38자리의 정수만 허용한다.

- BINARY_INTEGER

  정수만을 표현하기 위해 지원되는 데이터 타입이다. BINARY_INTEGER는 -2^31 ~ 2^31 - 1사이의 값을 갖는 부호를 포함하는 정수 값을 표현하기 위해 사용된다. BINARY_INTEGER는 바이너리 형식으로 저장된다.

  BINARY_INTEGER의 서브 타입은 다음과 같다.

서브 타입	설명
NATURAL	N >= 0인 정수만 허용한다.
NATURALN	NULL을 대입할 수 없다.
POSITIVE	POSITIVE는 N > 0인 정수만 허용한다.
POSITIVEN	NULL을 대입할 수 없다.
SIGNTYPE	-1, 0, 1의 값만 허용한다.

- BINARY_FLOAT / BINARY_DOUBLE

  부동소수점 실수를 표현하고, 빠른 계산을 위해 사용된다.

  선언하는 문법은 다음과 같다.

```
x BINARY_FLOAT := 1.5[f|F];

y BINARY_DOUBLE := 1.5[d|D];
```

- PLS_INTEGER

  BINARY_INTEGER 타입과 동일한 범위(-2^31 ~ 2^31 - 1)를 가진다.

CHARACTER / STRING 그룹

CHARACTER 그룹의 데이터 타입은 문자나 문자열 값을 표현한다.

- VARCHAR2

  데이터베이스에서 사용하는 VARCHAR2 타입과 유사하다.

  선언하는 문법은 다음과 같다.

```
VARCHAR2(L)
```

항목	설명
L	변수의 최대 길이이다. – 가변 길이 문자열을 저장한다. – 항상 표시되어야 한다.

항목	설명
	– 길이는 문자가 아니라 Byte로 지정된다. – 최대 길이는 32767bytes이다. – 실제 데이터는 ASCII나 EUR-KR, UTF-8 등 데이터베이스의 문자 집합으로 변경되어 저장된다.

데이터베이스에서 사용하는 VARCHAR2 타입의 컬럼은 4000bytes만을 저장할 수 있다. 따라서 4000bytes 이상인 tbPSM 변수를 VARCHAR2 타입의 데이터베이스 컬럼에 저장할 경우 에러가 발생한다.

VARCHAR2의 서브 타입은 다음과 같다.

서브 타입	설명
STRING	VARCHAR2와 같은 크기의 값을 표현할 수 있으며, 타입의 호환을 위해 사용하는 서브 타입이다.
VARCHAR	VARCHAR2와 같은 크기의 값을 표현할 수 있으며, 타입의 호환을 위해 사용하는 서브 타입이다.

- CHAR

  고정 길이 문자열로 문자 데이터를 저장하고 남는 공간을 빈 칸(blank)으로 채운다. 빈 칸(blank)으로 채워지기 때문에 같은 문자열이 저장된 변수라도 서로 길이가 다르다면, 이 두 변수를 비교할 때 서로 일치하지 않는다는 점에 주의해야 한다.

  선언하는 문법은 다음과 같다.

```
CHAR[(L)]
```

옵션	설명
L	변수의 최대 길이이다. – 고정 길이 문자열을 저장한다. – VARCHAR2와는 달리 길이를 지정하는 것은 옵션이다. (기본값: 1) – 길이는 문자가 아니라 Byte로 지정된다. – 최대 길이는 32767bytes이다.

데이터베이스에서 사용하는 CHAR 타입의 컬럼은 2000bytes만을 저장할 수 있다.

CHAR의 서브 타입은 다음과 같다.

서브 타입	설명
CHARACTER	CHAR과 같은 크기의 값을 표현할 수 있으며, 타입의 호환을 위해 사용하는 서브 타입이다.

- NVARCHAR2

  데이터베이스에서 사용하는 NVARCHAR2 타입과 유사하다.

  선언하는 문법은 다음과 같다.

```
NVARCHAR2(L)
```

항목	설명
L	변수의 최대 길이이다. – 가변 길이 문자열을 저장한다. – 항상 표시되어야 한다. – 길이는 항상 문자 단위로 지정된다. – 최대 길이는 32767bytes이다. – 실제 데이터는 UTF8이나 UTF16 등 데이터베이스의 다국어 문자 집합으로 변경되어 저장된다.

데이터베이스에서 사용하는 NVARCHAR2 타입의 컬럼은 4000bytes만을 저장할 수 있다. 따라서 4000bytes 이상인 tbPSM 변수를 NVARCHAR2 타입의 데이터베이스 컬럼에 저장할 경우 에러가 발생한다.

NVARCHAR2의 서브 타입은 다음과 같다.

서브 타입	설명
NVARCHAR	NVARCHAR2와 같은 크기의 값을 표현할 수 있으며, 타입의 호환을 위해 사용하는 서브 타입이다.

- NCHAR

  고정 길이 문자열로 문자 데이터를 저장하고 남는 공간을 빈 칸(blank)으로 채운다. 빈 칸(blank)으로 채워지기 때문에 같은 문자열이 저장된 변수라도 서로 길이가 다르다면, 이 두 변수를 비교할 때 서로 일치하지 않는다는 점에 주의해야 한다.

  선언하는 문법은 다음과 같다.

```
NCHAR[(L)]
```

옵션	설명
L	변수의 최대 길이이다. – 고정 길이 문자열을 저장한다. – NVARCHAR2와는 달리 길이를 지정하는 것은 옵션이다. (기본값: 1)

옵션	설명
	– 길이는 항상 문자 단위로 지정된다. – 최대 길이는 32767bytes이다.

데이터베이스에서 사용하는 NCHAR 타입의 컬럼은 2000bytes만을 저장할 수 있다.

- LONG

  2GB(2^31)까지의 데이터를 저장할 수 있는 데이터베이스의 LONG 타입과는 달리 32760bytes의 최대 길이를 갖는 가변 길이 문자열이다.

  LONG 타입은 VARCHAR2 타입과 매우 유사하다. LONG 타입의 데이터베이스의 컬럼은 32760bytes 이상의 데이터를 저장할 수 있기 때문에 tbPSM의 LONG 타입으로는 LONG 타입의 데이터베이스 컬럼 값을 저장하는데 제약이 따른다. 하지만 tbPSM의 LONG 타입의 최대 길이는 데이터베이스에서 사용하는 LONG 타입보다 작으므로 제한 없이 데이터베이스 컬럼에 삽입할 수 있다.

- RAW

  이 타입은 바이너리 데이터를 저장하기 위한 고정길이 문자열이다.

  선언하는 문법은 다음과 같다.

```
RAW(L)
```

항목	설명
L	변수의 최대 길이이다. – 고정 길이 문자열을 저장한다. – 항상 표시되어야 한다. – 길이는 문자가 아니라 Byte로 지정된다. – 최대 길이는 32767bytes이다.

데이터베이스에서 사용하는 RAW 타입의 컬럼은 2000bytes만을 저장할 수 있다. 따라서 만일 데이터의 길이가 2000bytes 이상이면 RAW 타입의 데이터베이스 컬럼에 저장될 수 없다. 그러나 최대 길이가 2GB(2^31)인 LONG RAW 타입의 데이터베이스 컬럼에는 삽입될 수 있다.

LONG RAW 타입의 데이터베이스 컬럼에 저장된 데이터의 길이가 32767bytes 이상이라면 tbPSM의 RAW 타입의 변수에 저장될 수 없다.

- LONG RAW

  이 타입은 바이너리 데이터 또는 Byte로 된 문자열을 저장하기 위해 사용하는 자료 구조이다.

  선언하는 문법은 다음과 같다.

```
LONG RAW(L)
```

항목	설명
L	변수의 최대 길이이다. – 항상 표시되어야 한다. – 길이는 문자가 아니라 Byte로 지정된다. – 최대 길이는 32760bytes이다.

데이터베이스에서 사용하는 LONG RAW 타입의 컬럼은 최대 길이가 2GB이기 때문에 컬럼 데이터의 실제 길이가 32760bytes 이상이라면 tbPSM의 LONG RAW 타입의 변수에 저장될 수 없다. 반면에 tbPSM의 LONG RAW 타입의 컬럼의 최대 길이보다 작으므로 저장에 제한이 없다.

- ROWID

  이 타입은 기본적으로 데이터베이스의 ROWID 타입과 같다. ROWID는 내부적으로 운영체제에 따라 길이가 다른 고정 길이 2진수로 저장된다. 일반적으로 ROWID는 tbPSM 프로그램에 의해 구성되지 않는다.

  ROWID는 ROWIDTOCHAR 내장 함수를 통해 문자열로 변환할 수 있다. 이 함수를 통해 출력된 결과 값은 총 18자리 문자열로 나타난다.

```
SSSSSS.FFF.BBBBBB.RRR
```

항목	설명
SSSSSS	데이터베이스의 세그먼트이다.
FFF	데이터 파일의 번호이다.
BBBBBB	데이터베이스의 블록이다.
RRR	블록 내의 로우이다.

ROWID의 각 구성요소는 16진수로 표현될 수 있다.

다음은 파일 1의 1번 세그먼트의 첫 번째 블록의 첫 번째 열을 의미하는 ROWID의 예이다.

```
000001.001.00001.001
```

DATETIME / INTERVAL 그룹

년, 월, 일, 시, 분, 초를 포함하는 날짜 및 시간 정보를 저장하기 위해 사용한다.

- DATE

  변수는 고정 길이의 날짜 정보를 저장한다. DATE 타입의 포맷은 초기화 파라미터인 NLS_DATE_FORMAT에 의해 결정된다. (기본값: 'YYYY/MM/DD')

다음은 DATE 타입을 사용한 예이다.

```
DECLARE
    today DATE := '2009/04/30';
```

Tibero에서 제공하는 **SYSDATE** 함수는 현재 날짜를 반환한다.

- TIMESTAMP

  변수는 DATE 타입을 확장하여 시간 정보까지 저장한다.

  선언하는 방법은 다음과 같다.

```
TIMESTAMP[(P)]
```

옵션	설명
P	소수점 초 단위의 정밀도이다. – 0 ~ 9사이의 값을 사용할 수 있다. (기본값: 6) – TIMESTAMP 타입의 포맷인 초기화 파라미터 NLS_TIMESTAMP_FORMAT에 의해 결정된다.

  다음은 TIMESTAMP 타입을 사용한 예이다.

```
DECLARE
    today TIMESTAMP := '2009/04/30 15:38:53.000';
```

- TIMESTAMP WITH TIME ZONE

  TIMESTAMP 타입을 확장하여 시간대 정보까지 저장한다.

  선언하는 방법은 다음과 같다.

```
TIMESTAMP[(P)] WITH TIME ZONE
```

옵션	설명
P	소수점 초 단위의 정밀도이다. – 0 ~ 9사이의 값을 사용할 수 있다. (기본값: 6) – TIMESTAMP WITH TIME ZONE 타입의 포맷인 초기화 파라미터 NLS_TIMESTAMP_TZ_FORMAT에 의해 결정된다.

  다음은 TIMESTAMP WITH TIME ZONE 타입을 사용한 예이다.

```
DECLARE
    today TIMESTAMP WITH TIME ZONE := '2009/04/30 15:38:53.000 Asia/Seoul';
```

- TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP 타입을 UTC(Coordinated Universal Time)으로 정규화해서 저장한다.

선언하는 방법은 다음과 같다.

```
TIMESTAMP[(P)] WITH LOCAL TIME ZONE
```

옵션	설명
P	소수점 초 단위의 정밀도이다. – 0 ~ 9사이의 값을 사용할 수 있다. (기본값: 6) – TIMESTAMP 타입의 포맷인 초기화 파라미터 NLS_TIMESTAMP_FORMAT에 의해 결정된다.

다음은 TIMESTAMP WITH LOCAL TIME ZONE 타입을 사용한 예이다.

```
DECLARE
    today TIMESTAMP WITH LOCAL TIME ZONE := '2009/04/30 15:38:53.000';
```

- INTERVAL YEAR TO MONTH

연도와 월의 차이를 저장한다.

선언하는 방법은 다음과 같다.

```
INTERVAL YEAR[(P)] TO MONTH
```

옵션	설명
P	연도의 자릿수를 결정한다. – 0 ~ 4 사이의 값을 사용할 수 있다. (기본값: 2)

다음은 INTERVAL YEAR TO MONTH 타입을 사용한 예이다.

```
DECLARE
    remained_months INTERVAL YEAR(3) TO MONTH;
BEGIN
    remained_months := INTERVAL '3-6' YEAR TO MONTH;
    remained_months := '3-6';
    remained_months := INTERVAL '3' YEAR;
    remained_months := INTERVAL '6' MONTH;
END;
```

- INTERVAL DAY TO SECOND

날짜와 초의 차이를 저장한다.

선언하는 방법은 다음과 같다.

```
INTERVAL DAY[(P)] TO SECOND[(F)]
```

옵션	설명
P	날짜의 자릿수를 결정한다. – 0 ~ 9 사이의 값을 사용할 수 있다. (기본값: 2)
F	초의 자릿수를 결정한다. – 0 ~ 9 사이의 값을 사용할 수 있다. (기본값: 6)

다음은 INTERVAL DAY TO SECOND 타입을 사용한 예이다.

```
DECLARE
    remained_days INTERVAL DAY(3) TO SECOND(3);
BEGIN
    remained_days := INTERVAL '3 6:02:05.9' DAY TO SECOND;
    remained_days := '3 6:02:05.9';
    remained_days := INTERVAL '3' DAY;
    remained_days := INTERVAL '12.902' SECOND;
END;
```

BOOLEAN 그룹

TRUE, FALSE, NULL 만을 저장할 수 있다.

다음은 유효하지 않은 BOOLEAN를 사용한 예이다.

```
DECLARE
    flag BOOLEAN := 0;
```

2.2.2. 복합 타입

tbPSM에서 사용할 수 있는 복합 타입(composite type)은 컬렉션 타입(테이블, 배열)과 레코드가 있다. 복합 타입은 내부에 하나 이상의 스칼라 타입을 포함해야 한다. 자세한 내용은 “제4장 복합 타입”을 참고한다.

2.2.3. 참조 타입

프로그램에서 변수가 일단 스칼라나 복합 타입으로 선언되면 참조 타입의 변수에 메모리 저장소가 할당된다. 변수에 할당된 메모리 저장소는 나중에 프로그램에서 참조를 위해 사용된다. 그러나, 일단 변수에 메모리 저장소가 할당이 되면 해제할 방법은 없으며, 계속 그 변수를 사용할 수 밖에 없다. 즉, 메모리 저장소는 변수가 소멸되기 전까지는 해제되지 않는다.

그러나 참조 타입은 이러한 제한이 없다. tbPSM에서 참조 타입은 C 언어의 포인터와 같다. 참조 타입으로 선언된 변수는 다른 저장소의 위치를 가리킨다. 현재 tbPSM에서 사용할 수 있는 참조 타입은 REF CURSOR 이다. 자세한 내용은 “7.3.3. 커서 변수”를 참고한다.

2.2.4. 대용량 객체형 타입

대용량 객체형 타입은 CLOB, BLOB, XMLTYPE, GEOMETRY, BFILE, JSON이 있다.

- CLOB

  대용량의 문자열 데이터를 저장한다. CLOB 타입으로 문자열이 저장될 때는 항상 고정길이의 문자 집합으로 변환된다. 최댓값은 4GB(2^64bytes)까지 가능하며, DBMS_LOB 패키지를 이용하여 조작할 수 있다.

- BLOB

  대용량의 바이너리 데이터를 저장한다. 최댓값은 4GB(2^64bytes)까지 가능하며, DBMS_LOB 패키지를 이용하여 조작할 수 있다.

- XMLTYPE

  XML형태의 데이터를 저장한다. 물리적으로 CLOB 타입과 동일하다.

- GEOMERY

  GEOMERY 형태의 데이터를 저장한다. 물리적으로 BLOB 타입과 동일히다.

- BFILE

  File Locator로써, 파일 접근을 위한 파일 경로와 파일이름을 저장하고 있는 타입니다. 데이터베이스 외부의 파일을 접근하기 위함이며, 최대 4GB 크기의 파일을 접근 가능하다.

- JSON

  JSON 형태의 데이터를 저장한다. 물리적으로 BLOB 타입과 동일하다.

2.2.5. 기타 타입

데이터베이스 테이블에 저장된 데이터를 조작하기 위해 사용하는 tbPSM 변수는 테이블 컬럼과 같은 타입을 가지고 있어야 하며, 해당 컬럼의 타입이 변경되더라도 프로그램에 영향을 미치지 않아야 한다.

다음과 같은 기타 타입의 변수가 사용된다.

- %TYPE

  어떤 테이블 컬럼 하나와 동일한 타입을 갖는다. 예를 들어 students 테이블의 first_name 컬럼은 VARCHAR2(20) 타입을 가지고 있다고 가정하면 다음과 같이 변수를 선언할 수 있다.

```
DECLARE
    v_first_name VARCHAR2(20);
```

만약 first_name 컬럼의 정의가 VARCHAR2(25) 타입으로 변경되면 이 컬럼을 사용하는 모든 코드는 변경되어야 한다. 이때 %TYPE을 사용할 수 있다. 테이블 컬럼과 연동되어 해당 타입을 반환한다.

예를 들면 다음과 같다.

```
DECLARE
    v_fisrt_name students.first_name%TYPE;
```

%TYPE를 사용하면 v_first_name 변수는 students 테이블의 first_name 컬럼의 타입과는 관계 없이 항상 동일하게 동작한다. 만일 %TYPE이 NOT NULL로 제한된 변수나 컬럼에 적용되는 경우 반환되는 타입은 이러한 제한을 갖지 않는다.

- %ROWTYPE

  테이블의 컬럼 전체와 같은 타입을 갖는 레코드 변수처럼 동작한다. 기본 속성은 %TYPE과 동일하다.

2.2.6. 사용자 정의 서브 타입

사용자는 필요에 따라 기존의 데이터 타입에 기반하여 새로운 서브 타입을 정의할 수 있다. 이를 칭할 때 사용자 정의 서브 타입이라고 한다.

선언하는 방법은 다음과 같다.

```
SUBTYPE new_type IS orig_type;
```

옵션	설명
new_type	새롭게 선언할 서브 타입의 이름이다.
orig_type	미리 정의된 타입이거나 서브 타입 또는 %TYPE일 수 있다. 새롭게 선언된 서브 타입(new_type)은 orig_type에 따라 해당 그룹에 속하게 된다.

예를 들면 orig_type를 %TYPE으로 선언하는 경우 tbPSM 블록의 선언부에 정의될 수 없다. 하지만 다음과 같이 더미(dummy) 변수를 선언한 후 이 변수를 이용하여 서브 타입을 선언할 수 있다.

```
DECLARE
    dummy NUMBER(4);
    SUBTYPE counter IS dummy%TYPE;
```

다음은 사용자 정의 서브 타입의 예이다.

```
DECLARE
    SUBTYPE Single IS NUMBER(1, 0);
    count Single;
```

위 예에서는 NUMBER 타입에 기반한 새로운 서브 타입 즉 **Single**이 선언되었다.

서브 타입의 선언은 항상 tbPSM 블록의 선언부에 포함되어야 하며, 서브 타입을 사용하기 전에 선언되어야 한다. 만약 서브 타입의 변수가 갖지 못하는 값을 할당하면 예외 상황이 발생한다.

서브 타입은 이미 존재하는 데이터 타입에 기초한 tbPSM의 타입이므로 애플리케이션 프로그램 개발자의 편의나 프로그램의 이해를 높이기 위한 별칭(Alias)으로도 사용할 수 있다. 또한 미리 정의된 서브 타입 외에 새로운 서브 타입을 정의할 수 있다.

예를 들면 다음과 같다.

```
DECLARE
    SUBTYPE T_Numeric is NUMBER;
    SUBTYPE V_Counter is T_Numeric(5);
```

2.3. 데이터 타입 변환

tbPSM은 스칼라 타입 중에서 다른 그룹 사이의 변환을 처리할 수 있다. 그룹 내에서의 변수의 제한을 제외하면 제한 없이 데이터 타입을 변경할 수 있다.

예를 들면 CAHR(10) 데이터 타입은 충분한 공간이 없기 때문에 VARCHAR(1)로 변환할 수 없다. 마찬가지로 정밀도와 스케일의 제약은 NUMBER(3)를 NUMBER(3,2)로 변환을 방해한다. 이렇게 변수에 제약이 있으면 컴파일러는 에러를 발생시키지 않는다. 그 대신 런타임 에러가 발생한다.

일반적으로 혼합된 데이터 타입은 변환 자체가 금지된다. 그러나 변환이 필요하면 변환 함수를 작성하여 프로그램에서 사용할 수 있다.

데이터 타입을 변환하는 방법에는 다음과 같이 두 가지가 있다.

- 명시적(explicit) 변환
- 묵시적(implicit) 변환

2.3.1. 명시적 변환

명시적 변환은 시스템 변환 함수와 CAST 구문을 사용하여 변환하는 것을 말한다.

다음은 시스템에서 제공하는 변환 함수이다.

구분	설명
TO_CHAR	입력 데이터를 VARCHAR2 타입으로 변환한다.

구분	설명
	변환이 가능한 타입은 다음과 같다. – NUMERIC 그룹 – RAW, LONG RAW를 제외한 CHARACTER / STRING 그룹 – DATETIME / INTERVAL 그룹 – ROWID – CLOB
TO_DATE	입력 데이터를 DATE 타입으로 변환한다. 단, 입력 데이터가 DATE 형식으로 되어 있지 않은 경우에는 예외 상황이 발생한다. 변환이 가능한 타입은 다음과 같다. – RAW, LONG RAW를 제외한 CHARACTER / STRING 그룹 – CLOB
TO_CLOB	입력 데이터를 CLOB 타입으로 변환한다. 변환이 가능한 타입은 다음과 같다. – NUMERIC 그룹 – RAW, LONG RAW를 제외한 CHARACTER / STRING 그룹 – DATETIME / INTERVAL 그룹 – ROWID
TO_TIMESTAMP	입력 데이터를 TIMESTAMP 타입으로 변환한다. 단, 입력 데이터가 TIMESTAMP 형식으로 되어 있지 않은 경우에는 예외 상황이 발생한다. 변환이 가능한 타입은 다음과 같다. – RAW, LONG RAW를 제외한 CHARACTER / STRING 그룹 – CLOB
TO_NUMBER	입력 데이터를 NUMBER 타입으로 변환한다. 단, 입력 데이터가 NUMBER 형식으로 되어 있지 않은 경우에는 예외 상황이 발생한다. 변환이 가능한 타입은 다음과 같다. – RAW, LONG RAW를 제외한 CHARACTER / STRING 그룹 – CLOB

구분	설명
RAWTOHEX	바이너리 데이터를 16진수 표현으로 변환한다. 변환이 가능한 타입은 다음과 같다. – RAW, LONG RAW
HEXTORAW	16진수 표현을 동일한 바이너리 데이터로 변환한다. 변환이 가능한 타입은 다음과 같다. – 16진수 표현으로 되어 있으며 VARCHAR2로 변환될 수 있는 타입
CHARTOROWID	ROWID 형식으로 된 문자열 변수를 ROWID 타입으로 변환한다. 변환이 가능한 타입은 다음과 같다. – ROWID 형식으로 되어 있으며, VARCHAR2로 변환될 수 있는 타입
ROWIDTOCHAR	ROWID 타입의 변수를 문자열로 변환한다. 변환이 가능한 타입은 다음과 같다. – ROWID

CAST 구문의 사용법은 다음과 같다.

```
CAST (expression AS type)
```

2.3.2. 묵시적 변환

묵시적 변환은 변수 사이의 대입 등에서 필요하다고 판단될 경우 자동으로 일어나는 변환을 말한다.

예를 들면 다음과 같다.

```
SQL> CREATE TABLE emp (id NUMBER, current_credits NUMBER(3));
SQL> INSERT INTO emp VALUES (1004, 2);
...

SQL> DECLARE
    cur_cred VARCHAR2(5);
BEGIN
    SELECT current_credits INTO cur_cred
    FROM emp WHERE id = 1004;
END;
```

위의 예는 emp 테이블에서 ID가 1004번인 직원의 현재 신용 번호를 조회하는 SELECT 문이다. 그러나 컬럼인 current_credits의 데이터 타입은 NUMBER(3)인 반면에 cur_cred는 VARCHAR2(5)이다. 이러한 경우 tbPSM은 NUMBER 타입의 데이터를 자동으로 VARCHAR2 타입으로 변환하여 INTO 절에 할당된 cur_cred에 저장한다.

다음은 묵시적 변환이 허용되는 타입이다.

	BIN_INT	**PLS_INT**	**NUMBER**	**CHAR**	**VARCHAR2**	**LONG**	**DATE**	**RAW**	**CLOB**	**BLOB**	**ROWID**
BIN_INT	O	O	O	O	O	O					
PLS_INT	O	O	O	O	O	O					
NUMBER	O	O	O	O	O	**O**					
CHAR	O	O	O	O	O	O	O	O	O		O
VARCHAR2	O	O	O	O	O	O	O	O	O		O
LONG	O	O	O	O	O	O	O	O	O		O
DATE				O	**O**	O					
RAW				O	O	O				O	
CLOB				O	O	O					
BLOB								O			
ROWID				O	O	O					

위 표에서 세로로 표현된 열은 원본 타입이고, 가로로 표현된 행은 대상 타입이다.

묵시적 변환에서 주의할 점은 다음과 같다.

- VARCHAR2 타입이 DATE 타입으로 변환되는 경우 위 표에서는 변환이 가능한 것으로 되어 있지만, 실제로는 VARCHAR2 타입의 변수가 DATE 형식으로 되어 있지 않으면 예외 상황이 발생한다.
- LONG 타입의 변수가 NUMBER 타입으로 변환되는 경우 위 표에서는 변환이 가능한 것으로 되어 있지만, 실제로는 LONG 타입의 변수가 NUMBER 형식으로 되어 있지 않으면 예외 상황이 발생한다.

특정한 형식을 갖고 있는 타입으로 변환하기 위해서는 원본 변수 역시 같은 형식을 가져야 한다.

예를 들면 다음과 같다.

```
DECLARE
    a VARCHAR2(20) := '2009/04/30';
    b VARCHAR2(20) := 'Tibero';
    c VARCHAR2(20) := '1';
    d NUMBER;
```

```
    e DATE;
BEGIN
    d := b;     -- 예외 상황이 발생한다.
    d := c;     -- 변환에 성공한다.

    e := a;     -- 변환에 성공한다.
    e := b;     -- 예외 상황이 발생한다.
END;
```

2.4. 데이터 변수의 선언과 참조 영역

본 절에서는 데이터 변수의 선언과 참조 영역에 대해서 설명한다.

2.4.1. 변수 선언

데이터 변수를 선언하려면 먼저 변수 이름을 입력하고 그 다음 변수의 데이터 타입을 정의한다. 이때 **변수에 초기 값을 할당할 수 있으며, 상수로 초기 값을 선언**할 수도 있다.

예를 들면 다음과 같다.

```
radius REAL := 1.0;

pi CONSTANT REAL := 3.141592654;
```

위의 예에서 변수 radius는 초기 값으로 1.0을 할당하였으며, 변수 pi는 항상 3.141592654 값을 갖도록 초기 값을 상수로 선언하였다.

데이터 변수에 **NOT NULL** 제약조건을 설정할 수도 있다. NOT NULL 제약조건을 설정하면 변수 값이 NULL이 되어서는 안 된다.

```
area REAL NOT NULL := 0.0;
```

만약 NULL이 변수 값으로 할당되면 예외 상황이 발생한다.

사용자가 필요에 의해 정의한 서브 타입에 대해서도 **NOT NULL** 제약조건을 설정할 수 있다. 이러한 서브 타입의 변수에 NULL를 할당하면 예외 상황이 발생한다.

```
SUBTYPE Single IS NUMBER(1, 0) NOT NULL;
```

2.4.2. 변수 참조 영역

변수 참조 영역(scope)은 프로그램의 일부로서, 한 프로그램 내에서 해당 변수에 접근할 수 있는 영역을 의미한다.

tbPSM 변수의 참조 영역은 변수가 선언된 블록이다. 즉, 변수의 선언에서 시작되고 변수가 선언된 블록이 끝날 때 참조 영역도 끝난다. 변수가 참조 영역을 벗어나면 해당 변수를 저장하기 위해 할당된 메모리는 시스템에 반환된다. 따라서 프로그램이 종료되지 않았어도 참조 영역을 벗어난 변수라면 접근할 수 없다.

예를 들면 다음과 같다.

<employee_num 변수의 참조 영역>

```
DECLARE
  employee_num   NUMBER;
BEGIN
  DECLARE
     employee_name  VARCHAR2(100);
  BEGIN
     employee_num  := '100';
     employee_name := 'Peter';
     ...
  END;
END;
```

각 변수의 참조 영역 내에서는 해당 변수와 동일한 이름을 사용할 수 없다. 그러나 참조 영역의 내부에 서브 블록이 사용되는 경우는 다르다.

아래 예와 같이 서브 블록 안에서는 외부 블록에서처럼 동일한 이름의 변수를 선언하고 사용할 수 있다.

```
DECLARE
  employee   PLS_INTEGER;
BEGIN                              ... ⓐ ...
  employee := 100;
  DECLARE
      employee  VARCHAR2(255);     ... ⓑ ...

      BEGIN                        ... ⓒ ...
          employee := 'John';
      END;                         ... ⓓ ...

  employee := employee + 10;       ... ⓔ ...
END;                               ... ⓕ ...
```

위 예를 기준으로 employee 변수의 참조 영역을 설명하면 다음과 같다.

ⓐ ~ ⓑ 외부 블록에서 선언된 employee가 사용되는 영역이다.

ⓒ ~ ⓓ 서브 블록에서 선언된 employee가 사용되는 영역이다.

ⓔ ~ ⓕ 외부 블록에서 선언된 employee가 사용되는 영역이다.

서브 블록 내에서는 외부 블록에서 선언된 employee 변수를 사용할 수 없다. 사용하고 싶다면 다음 예에서처럼 외부 블록에 레이블을 붙여서 사용해야 한다.

```
<<first_block>>
DECLARE
    employee PLS_INTEGER;
BEGIN
    employee := 100;

    DECLARE
        employee VARCHAR2(255);
    BEGIN
        employee := 'Susan';
        first_block.employee := 200;
    END;

    employee := employee + 10;
END;
```

서브 프로그램의 내부에서 선언된 변수의 경우에도 위와 같은 방법으로 접근한다.

2.5. 연산식

tbPSM 프로그램에서는 기본적으로 제공되는 Tibero의 연산식을 그대로 사용한다. 단, Tibero의 기본 연산식에 포함되지 않는 **CASE 연산자**가 존재한다.

CASE 연산자

CASE 연산자는 여러 조건 중에서 만족되는 조건에 연관된 값을 반환하는 연산자이다.

다음은 CASE 연산자를 사용한 예이다.

```
name := CASE order
WHEN 1 THEN 'Mercury'
WHEN 2 THEN 'Venus'
WHEN 3 THEN 'Earth'
...
ELSE 'No Such Planet'
END;
```

위 예에서 데이터 변수 order에 할당된 값에 따라 변수 name의 값은 달라진다. 여기에서 변수 order를 **선택자**(selector)라고 한다.

변수 order의 값이 WHEN 문 뒤의 값과 일치하면 THEN 뒤의 값을 CASE 연산자가 반환한다. 이와는 반대로 변수 order의 값이 WHEN 뒤의 값 중에 어떤 것과도 일치하지 않으면 ELSE 뒤의 값을 반환한다.

위의 예를 기준으로 설명하면, order의 값이 1이라면 CASE 연산자는 Mercury를 반환하고, 따라서 name에는 Mercury가 저장된다. 이와 마찬가지로 order의 값이 2라면 name에는 Venus가 저장될 것이다.

또는 선택자 없이 CASE 연산자를 사용할 수도 있다. 선택자가 있을 때는 선택자와 WHEN 다음의 값을 비교할 때 동등 연산자(=)로 비교한다. 하지만 선택자가 없는 경우에는 WHEN 다음에 임의의 논리 연산식이 올 수 있다. 예를 들면 다음과 같다.

```
name := CASE
WHEN 1 <= order and order <= 2 THEN 'Inner Planet'
WHEN order = 3 THEN 'Earth'
WHEN 4 <= order and order <= 9 THEN 'Outer Planet'
ELSE 'No Such Planet'
END;
```

이처럼 선택자가 없으면 변수 order에 좀 더 복잡한 논리 조건을 사용할 수 있다.

NULL을 포함하는 연산식의 계산

논리 또는 산술 연산식의 피연산자로 NULL 값이 포함되어 있을 때에는 일반적인 경우와 다르게 연산식을 계산할 수 있다.

다음은 **논리 연산자의 피연산자가 NULL 값인 경우**에 대한 결과이다.

X	Y	X and Y	X or Y	not X
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	NULL
NULL	NULL	NULL	NULL	NULL

비교 연산자의 피연산자가 NULL 값인 경우에는 항상 NULL를 반환한다. IF 문과 같은 제어 구조에서는 논리 연산식이 NULL을 반환하면 FALSE가 반환된 것처럼 처리된다. 만약 특정 변수의 값이 NULL인지를 확인하려면 IS NULL 연산자를 사용한다.

2.5.1. 연산자

가장 기본적인 연산자는 대입(assignment) 연산자이다.

대입 연산자

대입 연산자의 문법은 다음과 같다.

```
variable := expression;
```

variable은 tbPSM의 변수이고 expression은 tbPSM의 표현식이다.

항목	설명
variable	tbPSM의 변수이다. – 대입 연산자의 왼쪽에는 lvalue가 위치한다.
expression	tbPSM의 표현식이다. – 대입 연산자의 오른쪽에는 rvalue가 위치한다. – rvalue는 실제 저장될 값이다. – 변수나 상수가 올 수 있다.

다음은 대입 연산자의 예이다.

```
DECLARE
    local_01 VARCHAR2(100);
    local_02 VARCHAR2(100);
    employee_num NUMBER;
BEGIN
   local_01 := 'Seoul';
   local_02 := local_01;
   employee_num := '1024';
END;
```

위의 예에서는 문자 상수인 ‘Seoul’과 숫자 상수인 ‘1024’가 각각 해당 변수에 대입된다.

연산자의 우선 순위

tbPSM의 표현식은 rvalue이다. 표현식은 본질적으로 SQL 문장의 하위 구조이기 때문에 문장의 일부분으로 표현되어야 한다. 예를 들어 표현식은 대입 연산자의 오른쪽 또는 SQL 문장의 일부분으로 표현될 수 있다. 이때 피연산자의 타입과 함께 표현식을 구성하는 연산자는 타입을 결정한다.

피 연산자는 연산자에 대한 인수이다. tbPSM의 연산자는 하나의 인수(unary)나 두 개의 인수(binary) 또는 그 이상의 인수를 가질 수 있다. 예를 들어 덧셈 연산자(+)는 단항 피연산자를 갖고, 곱셈 연산자(*)는 이항 피연산자를 갖는다.

다음은 연산자의 우선 순위에 따라 tbPSM의 연산자를 분류한 것이다. 아래 표에서는 가장 높은 우선 순위를 가진 연산자가 먼저 기술된다.

연산자	타입	설명
**	이항	지수(exponentiation)
+, -	단항	부호

연산자	타입	설명
*, /	이항	곱셈, 나눗셈
+, -, ||	이항	덧셈, 뺄셈, 연결(concatenation)
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	이항	논리 비교
NOT	단항	논리 부정
AND	이항	논리 결합
OR	이항	논리 포함

tbPSM의 표현식에서 연산자의 우선 순위는 계산의 순서를 결정한다.

```
3 + 5 * 7
```

예를 들어 위 예와 같이 곱셈이 덧셈보다 더 높은 우선 순위를 가지므로 이 식의 계산 결과는 56(= 8 * 7)이 아니라 38(= 3 + 35)이 된다.

기본적으로 부여되는 연산자의 우선 순위를 무시하기 위해서는 표현식에 **괄호**를 사용하면 된다. 예를 들어 다음 식은 56(= 8 * 7)으로 계산된다.

```
(3 + 5) * 7
```

유일한 문자 연산자는 **문자열 결합**을 할 수 있는 접합 연산자(||)이다. 이 연산자는 두 개의 문자열을 하나의 문자열로 연결한다. 예를 들어 다음 식은 'SeoulKoreaAsia'로 계산된다.

```
'Seoul' || 'Korea' || 'Asia'
```

문자열 결합을 하는 표현식에서 모든 피연산자가 VARCHAR2 타입이라면 해당 결과 식은 VARCHAR2 타입이 된다. 예를 들면 다음과 같다. 문자열 상수는 CHAR 타입으로 간주되지만, 결과 식은 VARCHAR2 타입이 된다.

```
DECLARE
    Local  VARCHAR2(100) := 'Seoul';
    Result VARCHAR2(255);
BEGIN
    Result := local || 'Korea';
END;
```

진리식

진리식은 진리 값(TRUE, FALSE, NULL)으로 계산되는 식이다.

예를 들어 다음은 진리식이다.

```
X > Y
NULL
(23 > 17 ) AND (2873 <= X)
```

진리 상수나 진리 변수를 피연산자로 갖고, 진리 값을 결과로 반환하는 연산자는 AND, OR, NOT이 있다. 각 연산자의 결과 값은 다음과 같은 진리표를 따른다.

X	Y	X and Y	X or Y	not X
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	NULL
NULL	NULL	NULL	NULL	NULL

위 표에서 NULL은 소실된 값이거나 알지 못하는 값을 의미한다.

다음 식은 두 번째 피연산자가 알지 못하는 값이므로 결과 값은 NULL을 반환한다.

```
TRUE AND NULL
```

비교 및 관계 연산자

비교나 관계 연산자는 숫자, 문자, 데이터 피연산자를 가질 수 있으며, 진리 값을 반환한다.

연산자는 다음과 같이 정의된다.

연산자	문법	정의
=	A = B	A와 B가 같다
!=	A != B	A와 B가 같지 않다
<	A < B	A가 B보다 작다
>	A > B	A가 B보다 크다
<=	A <= B	A가 B보다 작거나 A와 B가 같다
>=	A >= B	A가 B보다 크거나 A와 B가 같다

- **IS NULL 연산자**

  IS NULL 연산자는 피연산자가 NULL일 때만 TRUE를 반환한다. IS NULL 이외의 연산자로는 NULL을 판단할 수 없다.

- **LIKE 연산자**

  LIKE 연산자는 문자열의 패턴을 매칭하는데 사용된다. 언더바(_)는 한 문자에 대응되고, 퍼센트(%)는 0개 이상의 문자에 대응한다.

  예를 들어 다음 식은 모두 TRUE를 반환한다.

```
'Peter' LIKE 'Pet_r'
'Peter' LIKE 'P%r'
'Peter' LIKE '%'
```

- **BETWEEN 연산자**

  BETWEEN 연산자는 3개의 피연산자를 갖는 연산자로서 어떤 값이 주어진 두 값 사이에 존재하는지를 판단한다. A BETWEEN B AND C와 같이 사용되며, 이는 A가 [B, C] 사이의 값인지 아닌지를 확인한다.

  예를 들어 다음 식은 FALSE를 반환한다.

```
10 BETWEEN 100 AND 200
```

- **IN 연산자**

  IN 연산자는 주어진 값이 해당 집합에 포함하는지 여부를 판단한다.

  예를 들어 다음 식은 FALSE를 반환한다.

```
'Seoul' IN ('NewYork', 'Tokyo', 'Beijing')
```

  해당 집합이 NULL을 포함하는 경우 비교는 항상 NULL을 반환하므로 연산은 무시된다.

제3장 제어 구조

본 장에서는 tbPSM이 제공하는 제어 구조를 설명한다.

3.1. 개요

제어 구조는 내부에 조건식을 포함하는 문장의 집합으로써, 조건식이 갖는 값에 의해 SQL 문장의 실행 순서가 결정된다. **조건식**은 진리식으로써 결과 값은 TRUE, FALSE, NULL 중 하나를 반환한다.

제4세대 언어인 SQL과는 달리 절차적인 언어에서는 SQL 문장의 실행 순서가 프로그램의 결과에 영향을 미친다. tbPSM은 이러한 절차적인 언어의 중요한 특성 중의 하나인 제어 구조를 SQL 문장에서 사용하게 함으로써 보다 효율적이고 유연한 프로그램을 작성할 수 있다.

tbPSM이 제공하는 제어 구조는 특정 작업을 수행하는 조건 구조와 반복적인 작업을 수행하는 반복 구조를 제공한다. 뿐만 아니라 단순 구조도 제공한다.

- **조건 구조 :** IF, CASE
- **반복 구조 :** LOOP
- **단순 구조 :** GOTO

3.2. IF 문

IF 문은 다음과 같이 3가지 형태를 지원한다.

형태	설명
IF-THEN	한 가지 경우만을 선택하여 사용하는 가장 단순한 형태이다.
IF-THEN-ELSE	두 가지 중에서 하나를 선택하여 사용한다.
IF-THEN-ELSEIF	다양한 경우에서 하나를 선택하여 사용한다.

3.2.1. IF-THEN 문

IF-THEN 문은 여러 IF 문 중 가장 단순한 구조로서 조건식의 결과 값이 TRUE인 경우 THEN 절의 실행문(또는 실행문의 집합)을 수행한다.

사용 방법은 다음과 같다.

```
IF 조건식 THEN
    실행문
END IF;
```

IF-THEN 문은 조건식의 결과 값이 TRUE일 때만 실행문이 실행된다. 조건식의 결과 값이 FALSE이거나 NULL인 경우에는 실행문은 실행되지 않고 제어가 다음 문장으로 이동한다.

다음은 IF-THEN 문의 예이다.

```
IF employee_num > 100 THEN
    pay_bonus(employee_num);
END IF;
```

위 예에서 보듯이 employee_num이 100보다 큰 경우에만 pay_bonus 프러시저가 실행된다.

3.2.2. IF-THEN-ELSE 문

IF-THEN-ELSE 문은 조건식의 결과 값이 TRUE 값을 가질 경우 THEN 절의 실행문-1을 수행하고, 나머지의 경우에는 ELSE 절의 실행문-2를 수행한다.

사용 방법은 다음과 같다.

```
If 조건식 THEN
    실행문-1
ELSE
    실행문-2
END IF;
```

IF-THEN 문과는 달리 IF-THEN-ELSE 문에서는 조건식의 결과 값이 FALSE이거나 NULL인 경우 ELSE 절의 실행문-2를 실행한다.

다음은 IF-THEN-ELSE 문의 예이다.

```
IF employee_num > 100 THEN
    up_grade(employee_num);
ELSE
    down_grade(employee_num);
END IF;
```

위 예에서 보듯이 employee_num이 100보다 큰 경우 조건식의 결과 값이 TRUE이므로, up_grade 프러시저가 실행되고, employee_num이 100보다 작거나 같은 경우에는 조건식의 결과 값이 FALSE이므로 down_grade 프러시저가 실행된다.

3.2.3. IF-THEN-ELSIF 문

IF-THEN-ELSIF 문은 다양한 조건과 그에 따른 실행문을 제시하는 복잡한 구조의 조건 구조이다.

사용방법은 다음과 같다.

```
If 조건식 THEN
    실행문-1
ELSIF 조건식 THEN
    실행문-2
...
ELSIF 조건식 THEN
    실행문-n
END IF;
```

IF-THEN-ELSEIF 문은 조건식의 결과 값이 TRUE인 경우 조건식에 속하는 THEN 절의 실행문을 수행한다.

조건식은 순서대로 수행되며, 조건식의 결과 값이 TRUE일 때까지 제어가 이동된다. 만약 조건식의 결과 값이 TRUE인 조건식이 한 개 이상인 경우에는 최초로 만나는 실행문 하나만을 실행한다. 그 외 나머지 실행문은 무시된다.

참고

IF-THEN-ELSE 문을 여러 번 중첩하면 IF-THEN-ELSIF 문과 동일하게 구현할 수 있다. 그러나 보다 이해하기 쉽고 명확한 프로그램의 작성과 디버깅 과정의 편의를 위해 IF-THEN-ELSIF 문을 사용할 것을 권장한다.

다음은 IF-THEN-ELSEIF 문의 예이다.

```
IF employee_num < 100 THEN
    up_grade(employee_num);
ELSIF employee_num > 200 THEN
    pay_salary(employee_num);
ELSIF employee_num > 300 THEN
    down_grade(employee_num);
END IF;
```

위 예에서 employee_num의 값이 380이라면, 두 번째, 세 번째 조건식이 모두 TRUE 값을 갖게 된다. 앞서 말했지만, 조건식의 결과 값이 TRUE인 조건식이 다수이면 최초로 만나는 실행문 하나만 실행된다. 따라서 두 번째 조건식의 실행문인 pay_salary 프러시저만 실행되고, 세 번째 조건식의 실행문인 down_grade 프러시저는 실행되지 않는다.

3.3. CASE 문

CASE 문은 여러 개의 실행문 중에서 하나를 선택하여 실행한다. CASE 문은 IF-THEN-ELSIF 문과 비슷한 성격을 가지며, 실제로 조건식을 포함하는 CASE 문은 IF-THEN-ELSIF 문으로 동일하게 다시 작성할 수 있다.

CASE 문은 동등 비교에 사용된다. 표현식의 결과 값과 일치하는 연산식에 속하는 THEN 절의 실행문을 실행한다. 동등 비교 이외의 경우에는 조건식을 포함하는 CASE 문을 사용한다.

사용 방법은 다음과 같다.

```
CASE 표현식
  WHEN 연산식 THEN 실행문-1
  WHEN 연산식 THEN 실행문-2
  ...
  WHEN 연산식 THEN 실행문-n
END CASE;
```

IF 문과 마찬가지로, 조건을 만족하는 실행문이 있는 경우 해당 실행문을 수행한 후 자동적으로 CASE 문 이후의 문장을 수행한다. 따라서 일반적인 프로그래밍 언어와는 달리 BREAK 문을 명시할 필요가 없다.

또한 IF 문과 마찬가지로 조건을 만족하는 연산식이 하나 이상 있는 경우에는 첫 번째 연산식의 실행문만 수행하고, 그 외 나머지 실행문은 무시된다.

다음은 CASE 문의 예이다.

```
CASE employee_grade
  WHEN 'A' THEN pay_bonus_a(employee_num);
  WHEN 'B' THEN pay_bonus_b(employee_num);
  WHEN 'C' THEN pay_bonus_c(employee_num);
  WHEN 'D' THEN pay_bonus_d(employee_num);
  WHEN 'E' THEN pay_bonus_e(employee_num);
  ELSE check_grade(employee_num);
END CASE;
```

위 예에서 보듯이 CASE 문에서 ELSE 절은 선택적으로 사용할 수 있다. 만약 사용자가 ELSE 절을 정의하지 않은 경우 표현식의 결과 값과 동일한 연산식이 없으면 tbPSM은 예외 상황을 발생시키고 예외 처리 루틴으로 제어를 이동한다.

3.3.1. 조건식을 포함하는 CASE 문

표현식이 없거나 조건식을 계산한 후의 결과 값이 TRUE인 경우에는 해당 절의 실행문을 수행한다.

사용 방법은 다음과 같다.

```
CASE
  WHEN 조건식 THEN 실행문-1
```

```
  WHEN 조건식 THEN 실행문-2
  ...
  WHEN 조건식 THEN 실행문-n
END CASE;
```

다음은 조건을 포함하는 CASE 문의 예이다.

```
DECLARE
  value PLS_INTEGER := 0;
  result VARCHAR2(10);
BEGIN
CASE
  WHEN value = 0 THEN  result := 'true';
  WHEN value != 0 THEN  result := 'false';
END CASE;
DBMS_OUTPUT.PUT_LINE( 'Is the value Zero? ' || result );
END;
/
```

3.4. LOOP 문

LOOP 문을 사용하면 반복 구조를 작성할 수 있다.

LOOP 문은 다음과 같이 세 가지 형태를 지원한다.

형태	설명
단순 LOOP	단순히 반복을 계속하는 LOOP 문이다.
FOR-LOOP	조건을 부여할 수 있는 LOOP 문이다.
WHILE-LOOP	조건을 부여할 수 있는 LOOP 문이다.

LOOP 문은 주로 EXIT 문과 결합하여 사용된다.

3.4.1. 단순 LOOP 문

단순 LOOP 문은 LOOP와 END LOOP 사이에 생성된 블록을 반복해서 수행한다.

사용 방법은 다음과 같다.

```
LOOP
    실행문
END LOOP;
```

EXIT 문의 사용

EXIT 문을 사용할 때 LOOP 문의 내부에서 EXIT를 만나면 무조건 LOOP 문을 빠져 나오게 된다. 그러나 정의된 EXIT 문이 없으면 LOOP 문은 무한히 반복한다.

따라서 다음 예와 같이 **IF 문과 EXIT 문을 함께 사용**하여 해당 조건식을 만족하는 경우 LOOP를 빠져 나오도록 지정할 수 있다.

```
LOOP
    v_order := v_order + 1;
    IF v_order > 9 THEN
        EXIT;
    END IF;
END LOOP;
```

EXIT WHEN 문의 사용

IF 문과 EXIT 문을 함께 사용하는 대신에 **EXIT WHEN** 문을 사용하면 일정한 조건을 만족할 때까지 EXIT 문을 수행할 수 있다.

사용 방법은 다음과 같다.

```
EXIT WHEN 조건식;
```

위 방법은 아래와 동일한 의미를 갖는다.

```
IF 조건식 THEN
    EXIT;
END IF;
```

다음은 EXIT WHEN 문의 예이다.

```
LOOP
    v_order := v_order + 1;
    EXIT WHEN v_order > 9;
END LOOP;
```

레이블의 사용

LOOP 문이 시작하기 전에 LOOP 문 앞에 레이블을 붙일 수 있다. 예를 들어 아래와 같이 레이블이 정의된 경우에는 EXIT 문에 레이블을 명시하여 여러 개의 LOOP 문을 한꺼번에 빠져나올 수 있다.

```
<<LOOP_OUT>>
LOOP
   <<LOOP_IN>>
    LOOP
```

```
        v_order := v_order + 1;
        EXIT LOOP_OUT WHEN v_order > 9;   ... ⓐ ...
    END LOOP LOOP_IN;
END LOOP LOOP_OUT;  ... ⓑ ...
```

ⓐ LOOP 문의 조건이 맞는 경우에 해당된다.

ⓑ 두 개의 LOOP 문에서 동시에 빠져 나와 LOOP 문 다음의 SQL 문장으로 이동한다.

3.4.2. FOR-LOOP 문

FOR-LOOP 문은 실행문을 정해진 수만큼 반복하고 싶은 경우에 사용한다. FOR-LOOP 문은 SQL 문장 내부에 반복 카운터(loop_counter)를 가지고 있어서 일정한 수만큼 LOOP 문을 반복할 수 있다.

사용 방법은 다음과 같다.

```
FOR loop_counter IN low_bound..high_bound LOOP
    실행문
END LOOP;
```

항목	설명
loop_counter	일반적으로 변수를 사용한다. 시스템이 내부적으로 선언하기 때문에 명시적으로 선언할 필요가 없다. loop_counter를 볼 수 있는 범위는 FOR-LOOP 문의 내부이다.
low_bound, high_bound	LOOP 문의 범위를 지정한다. 주로 숫자 상수를 사용한다. 그러나 반드시 숫자 상수일 필요는 없으며, 숫자 상수로 변환될 수 있는 임의의 식을 사용할 수 있다.

다음은 FOR-LOOP 문의 예이다.

```
FOR box_num IN 1..100 LOOP
    box_weight := box_num * 10;
END LOOP;
```

위 예에서 보듯이 loop_counter는 box_num 변수로 정의하고, low_bound와 high_bound는 1부터 시작하여 100에 도달할 때까지 값을 1씩 증가시키며 LOOP 문을 반복한다. low_bound와 high_bound는 반드시 숫자 상수일 필요는 없다.

예를 들면 다음과 같다.

```
DECLARE
    low_value BINARY_INTEGER := 10;
    high_value BINARY_INTEGER := 50;
    box_weight NUMBER;
BEGIN
```

```
    FOR box_num IN low_value..high_value LOOP
        box_weight := box_num * 10;
    END LOOP;
END;
```

동일한 이름의 변수 사용

만약 외부에서 loop_counter와 동일한 이름을 갖는 변수를 선언한 상태라면, 다음 예와 같이 FOR-LOOP 문의 내부 블록에서는 사용자가 외부에 선언한 변수는 loop_counter에 의해 영향을 받지 않는다.

```
DECLARE
    box_num VARCHAR2(255);
BEGIN
    box_num := 'BOX_A';
    FOR box_num IN 1..100 LOOP
        --이 부분에서 box_num은 loop_counter이다.
        INSERT INTO BOX_INFO (ID) VALUES (box_num);
    END LOOP;
    --이 부분에서 box_num은 VARCHAR2 데이터 타입의 변수이며, 값은 BOX_A 문자열를 갖는다.
END;
```

REVERSE 예약어 사용

FOR LOOP 문 안에서 REVERSE 예약어를 사용하면 loop_counter가 high_bound에서 low_bound로 감소하면서 LOOP 문을 반복하게 된다.

사용 방법은 다음과 같다.

```
FOR loop_counter IN REVERSE low_bound...high_bound LOOP
    실행문
END LOOP;
```

만약 REVERSE 예약어를 사용하지 않는 경우라면 low_bound와 high_bound는 FOR-LOOP 문과 같은 순서로 LOOP 문을 수행한다. 단, high_bound가 low_bound보다 먼저 나오는 경우 RESERVE 예약어의 사용과 관계없이 tbPSM 프로그램은 에러를 발생시킨다.

다음은 REVERSE 예약어를 사용한 FOR-LOOP 문의 예이다.

```
FOR box_num IN REVERSE 1..100 LOOP
    box_weight := box_num * 10;
END LOOP;
```

위 예에서 보듯이 100에서 시작해서 1에 도달할 때까지 1씩 감소한다.

3.4.3. WHILE-LOOP 문

WHILE-LOOP 문은 WHILE 다음에 나오는 조건식을 계산하여 결과 값이 TRUE인 동안에만 실행문을 반복적으로 실행한다. 조건식의 결과 값이 TRUE가 아닌 경우에는 LOOP 이후 다음 문장으로 제어가 이동된다.

사용하는 문법은 다음과 같다.

```
WHILE 조건식 LOOP
    실행문
END LOOP;
```

즉, LOOP 문 자체가 조건식을 포함하고 있다. 따라서 EXIT WHEN 문을 별도로 사용하지 않아도 조건에 따라 LOOP 문에서 빠져나올 수 있다.

다음 문장은 단순 LOOP 문과 동일한 의미의 예이다.

```
WHILE TRUE LOOP
    실행문 집합
END LOOP;
```

3.5. GOTO 문

GOTO 문은 실행문이나 블록에 추가한 레이블과 함께 동작한다. GOTO 문이 실행되면 GOTO 문에 명시된 레이블을 찾아서 해당 실행문이나 블록으로 제어를 이동한다. 일반적인 프로그래밍 언어와는 달리 LABEL 문이 GOTO 문보다 먼저 올 수 있다.

사용하는 방법은 다음과 같다.

```
GOTO lable;
```

IF 문이나 LOOP 문의 내부에서 외부로 분기하는 경우처럼 제어 구조에서 GOTO 문을 사용할 때가 있지만, GOTO 문은 사용할 때에는 다음과 같은 주의가 필요하다. 예를 들어 다음과 같이 IF 문이나 LOOP 문의 중간으로 분기하는 것은 적합하지 않으므로 주의해야 한다.

```
BEGIN
    GOTO inner_loop;
    FOR x IN 1..100 LOOP
        <<inner_loop>>
        EXIT WHEN x > 50;
    END LOOP;
END;
```

또한 IF 문의 중간에서 다른 IF 문의 내부로 분기하는 것도 허용하지 않으며, 외부 블록을 수행하는 도중에 내부 블록의 중간으로 분기하는 것도 맞지 않다. 마지막으로 예외 처리 루틴에서 실행 블록으로 분기하는 것도 허용하지 않는다.

3.6. EXIT 문

EXIT 문은 LOOP 문의 내부에 사용하면 자신을 포함하고 있는 LOOP 문에서 빠져 나올 수 있다. 또한 LOOP 문에 레이블을 붙일 수 있기 때문에 EXIT 문에 레이블을 명시하면 여러 개의 LOOP 문을 동시에 빠져나올 수 있다.

EXIT 문에 레이블을 명시하는 방법은 다음과 같다.

```
EXIT [레이블];
```

다음은 레이블을 사용하여 두 개의 LOOP 문을 동시에 빠져 나오는 예이다.

```
BEGIN
    <<outer>>
    FOR out_index IN 1..100 LOOP
        <<inner>>
        FOR inner_index IN 1..1000 LOOP
            IF inner_index > out_index THEN
                EXIT outer;
            END IF;
        END LOOP inner;
    END LOOP outer;
END;
```

EXIT 문을 수행하기 전에 제시한 조건식을 만족하는지 확인한다. 조건식을 만족하는 경우에는 해당 LOOP 문을 빠져 나올 수 있다.

3.7. NULL 문

NULL 문은 프로그램상에 명시되어도 실제로는 아무런 일도 발생하지 않는다.

프로그램을 설계할 때 **서브 프로그램의 내용을 일시적으로 NULL 문으로 대체**하거나, 또는 제어 구조를 작성하면서 **실행문을 NULL 문으로 대체**하는 경우라면 NULL 문은 매우 유용하다.

예를 들면 다음과 같다.

```
DECLARE
    tmp_current_guest BINARY_INTEGER;
    tmp_max_guest     BINARY_INTEGER;

    PROCEDURE raise_guest_max (current_guest BINARY_INTEGER) IS
    BEGIN
        NULL;
    END;
BEGIN
    SELECT current_guest, max_guest
        INTO tmp_current_guest, tmp_max_guest
```

```
        FROM guest WHERE local = 'SEOUL';

    IF tmp_current_guest > tmp_max_guest THEN
        raise_guest_max (tmp_current_guest);
    ELSE
        NULL;
    END IF;
END;
```

형식적으로 실행문이 반드시 필요한 경우(레이블을 사용하는 경우)에도 다음과 같이 NULL 문을 사용할 수 있다.

```
BEGIN
    FOR x IN 1..100 LOOP
        <<inner_loop>>
       NULL;
    END LOOP;
END;
```

3.8. CONTINUE 문

CONTINUE 문은 LOOP 내에서 해당 LOOP를 건너뛰고 다음 LOOP로 진행할 때 사용된다. LOOP를 작성할 때 **특정 조건에서 해당 LOOP를 수행하지 않을 경우** CONTINUE 문을 사용하면 편리하다.

예를 들면 다음과 같다.

```
DECLARE
    x pls_integer := 0;
BEGIN
    for i in 1..100 loop
        if mod(i,2) = 0 then
          continue;
        END IF;
        x := x + i;
    END LOOP;
END;
```

CONTINUE WHEN 문 사용

IF 문과 CONTINUE 문을 함께 사용하는 대신에 CONTINUE WHEN 문을 사용하면 일정한 조건을 만족할 때 해당 LOOP를 건너뛰고 다음 LOOP로 진행할 수 있다.

사용 방법은 다음과 같다.

```
CONTINUE WHEN 조건식;
```

위 방법은 아래와 동일한 의미를 갖는다.

```
IF 조건식 THEN
    CONTINUE;
END IF;
```

다음은 CONTINUE WHEN 문의 예이다.

```
DECLARE
    x pls_integer := 0;
BEGIN
    for i in 1..100 loop
        CONTINUE WHEN mod(i,2) = 0;
        x := x + i;
    END LOOP;
END;
```

레이블 사용

LOOP 문에 레이블이 정의된 경우에는 CONTINUE 문에 레이블을 명시하여 상위 LOOP로 돌아갈 수 있다.

CONTINUE 문에 레이블을 명시하는 방법은 다음과 같다.

```
CONTINUE [레이블];
```

다음은 레이블을 사용하여 inner LOOP를 빠져 나와 outer LOOP로 돌아가는 예이다.

```
BEGIN
    <<outer>>
    FOR out_index IN 1..100 LOOP
        <<inner>>
        FOR inner_index IN 1..1000 LOOP
            IF inner_index > out_index THEN
                CONTINUE outer;
            END IF;
        END LOOP inner;
    END LOOP outer;
END;
```

CONTINUE 문을 수행하기 전에 제시한 조건식을 만족하는지 확인한다. 조건식을 만족하는 경우에는 inner LOOP의 다음 iteration이 아닌 outer LOOP의 다음 iteration으로 돌아갈 수 있다.

제4장 복합 타입

본 장에서는 tbPSM에서 제공하는 구조체 형태의 콜렉션 타입과 레코드를 설명한다.

4.1. 개요

복합 타입(composite type)이란 tbPSM이 제공하는 스칼라 타입의 집합이다. 일반적인 프로그래밍 언어의 구조체에 해당하며, 그 종류는 다음과 같다.

- 컬렉션(collection) 타입 : 테이블(nested table), 인덱스 테이블(indexed by table), 배열(varray)
- 레코드(record)

4.2. 컬렉션 타입

컬렉션이란 같은 타입을 갖는 구성요소의 집합이다. 일반적인 프로그래밍 언어에서 사용하는 배열이나 리스트와 비슷한 개념이다.

tbPSM에서 지원하는 컬렉션 타입에는 세 가지가 있다. 즉, 테이블, 인덱스 테이블, 그리고 배열이다.

4.2.1. 테이블

테이블은 구성요소의 길이 제한이 없고, 각각의 구성요소에 접근할 때는 배열과 마찬가지로 인덱스로 접근한다. 그러나 배열과 다르게 구성요소가 연속적으로 존재하지 않을 수 있다. 즉, 어떤 구성요소의 다음 인덱스에 값이 반드시 존재한다는 보장이 없다.

테이블의 구성요소는 항상 같은 타입을 가지며, REF CURSOR를 제외한 모든 타입이 될 수 있다.

tbPSM에서 테이블을 선언하는 방법은 다음과 같다.

```
TYPE name IS TABLE OF type [NOT NULL];
```

테이블의 초기화

테이블은 초기화라는 과정을 거치게 되는데, 이 과정을 거치지 않은 변수는 항상 NULL을 가지며, 구성요소도 존재하지 않게 된다. 만약 초기화되지 않은 테이블에 접근할 경우 **COLLECTION_IS_NULL** 예외 상황이 발생한다.

테이블을 초기화하는 방법은 다음과 같다.

```
DECLARE
    TYPE kind_tab IS TABLE OF VARCHAR2(10) NOT NULL;
    kinds kind_tab;
BEGIN
    kinds := kind_tab('math', 'physics', 'history', 'science');
END;
```

테이블의 구성요소

테이블은 최대 크기의 제한이 없기 때문에 초기 값으로 주어진 구성요소가 현재의 최대 길이가 된다. 만약 테이블에서 NOT NULL 제약조건이 없을 경우 NULL을 구성요소로 사용할 수 있다.

```
DECLARE
    TYPE kind_tab IS TABLE OF VARCHAR2(10);
    kinds kind_tab := kind_tab('math', NULL, 'history', 'science');
BEGIN
    NULL;
END;
```

테이블의 구성요소는 하나의 독립적인 변수처럼 사용할 수 있으며, 해당 변수의 값을 추출할 수 있다. 또한 대입도 가능하다.

예를 들면 다음과 같다.

```
DECLARE
    TYPE class_tab IS TABLE OF VARCHAR2(10);
    classes class_tab := class_tab('math', 'physics', 'science');
BEGIN
    classes(3) := 'history';
    IF classes(1) IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('I hate this class');
    END IF;
    DBMS_OUTPUT.PUT_LINE(classes(3) || ' is my favorite class');
END;
/
history is my favorite class

PSM completed
```

테이블의 구성요소를 참조하기 위해서는 다음과 같이 사용해야 한다.

```
table_name(index)
```

index에는 1 ~ 2^31 - 1사이의 값만을 사용할 수 있다.

MULTISET 연산자

테이블 형태의 변수에는 MULTISET 연산자의 결과를 대입할 수 있다. MULTISET 연산자로는 UNION, INTERSECT, EXCEPT 세 가지가 있다.

UNION은 MULTISET 연산자의 두 테이블을 합쳐 준다. 합칠 때 중복값의 제거 여부를 옵션으로 결정할 수 있다. ALL과 DISTINCT의 두 가지 옵션이 있으며, ALL 옵션은 중복되는 값을 제거하지 않는다. 반면 DISTINCT 옵션은 중복되는 값을 제거한다. 옵션을 주지 않으면 ALL과 같이 동작한다.

```
DECLARE
    TYPE kind_tab IS TABLE OF NUMBER;
    tab1 kind_tab := kind_tab(1,2,3);
    tab2 kind_tab := kind_tab(1,4,5);
    result kind_tab;
BEGIN
    result := tab1 MULTISET UNION ALL tab2;
END;
```

MULTISET UNION 에 의해 tab1과 tab2의 구성요소들이 합쳐져 result 테이블에 저장된다. 옵션은 ALL이므로 중복값을 제거하지 않아, 이 테이블은 1, 2, 3, 1, 4, 5의 여섯 개 구성요소를 갖는다.

```
DECLARE
    TYPE kind_tab IS TABLE OF NUMBER;
    tab1 kind_tab := kind_tab(1,2,3);
    tab2 kind_tab := kind_tab(1,4,5);
    result kind_tab;
BEGIN
    result := tab1 MULTISET UNION tab2;
END;
```

ALL 혹은 DISTINCT 옵션을 사용하지 않은 경우 위 예제의 ALL 옵션과 동일하게 동작한다. result 테이블은 1, 2, 3, 1, 4, 5의 여섯 개 구성요소를 갖는다.

```
DECLARE
    TYPE kind_tab IS TABLE OF NUMBER;
    tab1 kind_tab := kind_tab(1,2,3);
    tab2 kind_tab := kind_tab(1,4,5);
    result kind_tab;
BEGIN
    result := tab1 MULTISET UNION DISTINCT tab2;
END;
```

DISTINCT 옵션을 사용한 경우 MULTISET 연산자의 결과에서 중복된 구성요소는 제거된다. result 테이블은 1, 2, 3, 4, 5의 다섯 개 구성요소를 갖는다.

INTERSECT는 MULTISET 연산자의 두 테이블 사이에 공통된 구성요소를 찾아 준다. 중복값의 제거 여부를 옵션으로 결정할 수 있다. ALL과 DISTINCT의 두 가지 옵션이 있으며, ALL 옵션은 중복되는 값을 제거하지 않는다. 반면 DISTINCT 옵션은 중복되는 값을 제거한다. 옵션을 주지 않으면 ALL과 같이 동작한다.

```
DECLARE
    TYPE kind_tab IS TABLE OF NUMBER;
    tab1 kind_tab := kind_tab(1,1,2,3);
    tab2 kind_tab := kind_tab(1,1,2,4);
    result kind_tab;
BEGIN
    result := tab1 MULTISET INTERSECT ALL tab2;
END;
```

MULTISET INTERSECT에 의해 tab1과 tab2의 구성요소들 중 공통된 구성요소를 찾아 result 테이블에 저장된다. 옵션은 ALL이므로 중복값을 제거하지 않아, 이 테이블은 1, 1, 2의 세 개 구성요소를 갖는다.

```
DECLARE
    TYPE kind_tab IS TABLE OF NUMBER;
    tab1 kind_tab := kind_tab(1,1,2,3);
    tab2 kind_tab := kind_tab(1,1,2,4);
    result kind_tab;
BEGIN
    result := tab1 MULTISET INTERSECT tab2;
END;
```

ALL 혹은 DISTINCT 옵션을 사용하지 않은 경우 위 예제의 ALL 옵션과 동일하게 동작한다. result 테이블은 1, 1, 2의 여섯 개 구성요소를 갖는다.

```
DECLARE
    TYPE kind_tab IS TABLE OF NUMBER;
    tab1 kind_tab := kind_tab(1,1,2,3);
    tab2 kind_tab := kind_tab(1,1,2,4);
    result kind_tab;
BEGIN
    result := tab1 MULTISET INTERSECT DISTINCT tab2;
END;
```

DISTINCT 옵션을 사용한 경우 MULTISET 연산자의 결과에서 중복된 구성요소는 제거된다. result 테이블은 1, 2의 두 개 구성요소를 갖는다.

EXCEPT는 MULTISET 연산자의 앞 테이블에는 있으면서 뒤 테이블에는 없는 구성요소를 찾아 준다. 중복값의 제거 여부를 옵션으로 결정할 수 있다. ALL과 DISTINCT의 두 가지 옵션이 있으며, ALL 옵션은 중복되는 값을 제거하지 않는다. 반면 DISTINCT 옵션은 중복되는 값을 제거한다. 옵션을 주지 않으면 ALL과 같이 동작한다.

```
DECLARE
    TYPE kind_tab IS TABLE OF NUMBER;
    tab1 kind_tab := kind_tab(1,2,3,3);
    tab2 kind_tab := kind_tab(1,2,4,5);
    result kind_tab;
BEGIN
```

```
    result := tab1 MULTISET EXCEPT ALL tab2;
END;
```

MULTISET EXCEPT에 의해 tab1의 구성요소이면서 tab2의 구성요소는 아닌 값들만 result 테이블에 저장된다. 옵션은 ALL이므로 중복값을 제거하지 않아, 이 테이블은 3, 3의 두 개 구성요소를 갖는다.

```
DECLARE
    TYPE kind_tab IS TABLE OF NUMBER;
    tab1 kind_tab := kind_tab(1,2,3,3);
    tab2 kind_tab := kind_tab(1,2,4,5);
    result kind_tab;
BEGIN
    result := tab1 MULTISET EXCEPT tab2;
END;
```

ALL 혹은 DISTINCT 옵션을 사용하지 않은 경우 위 예제의 ALL 옵션과 동일하게 동작한다. result 테이블은 3, 3의 두 개 구성요소를 갖는다.

```
DECLARE
    TYPE kind_tab IS TABLE OF NUMBER;
    tab1 kind_tab := kind_tab(1,2,3,3);
    tab2 kind_tab := kind_tab(1,2,4,5);
    result kind_tab;
BEGIN
    result := tab1 MULTISET EXCEPT DISTINCT tab2;
END;
```

DISTINCT 옵션을 사용한 경우 MULTISET 연산자의 결과에서 중복된 구성요소는 제거된다. result 테이블은 3의 한 개 구성요소를 갖는다.

4.2.2. 인덱스 테이블

인덱스 테이블은 키와 값이 합쳐진 구성요소를 갖는 테이블이다. 각각의 구성요소에 접근할 때에는 숫자 또는 문자열을 이용하여 인덱스로 접근한다. 인덱스 테이블의 구성요소는 PLS_INTEGER, BINARY_INTERGER, VARCHAR2, STRING, LONG 타입이어야 한다.

tbPSM에서 인덱스 테이블을 선언하는 방법은 다음과 같다.

```
TYPE index_table1 OF PLS_INTEGER [NOT NULL] INDEX BY PLS_INTEGER;
TYPE index_table1 OF VARCHAR2(10) INDEX BY STRING(1);
```

인덱스 테이블의 사용

인덱스 테이블은 선언할 때 최대 크기의 제한이 없다. 인덱스 테이블의 크기는 인덱스가 동일하게 설정된 키에 값을 삽입하는 경우 이전 값이 변경되므로 증가하지 않는다. 반면에 인덱스가 다르게 설정된 키에 값을 삽입하는 경우 인덱스 테이블의 크기는 증가하게 된다.

```
DECLARE
  TYPE kind_tab IS TABLE OF NUMBER INDEX BY VARCHAR2(10);
  fruit_table kind_tab;  -- 과일 개수 테이블, 인덱스는 과일명
BEGIN
  fruit_table('apple'):= 10;
  fruit_table('orange'):= 100;
  fruit_table('strawberry'):= 35;
  fruit_table('pineapple'):= 80;
  fruit_table('watermelon'):= 20;
END;
/
```

4.2.3. 배열

배열은 테이블과 달리 선언할 때 길이의 제한이 있다. 따라서 선언할 때 반드시 길이를 지정해야 한다. 그리고 각각의 구성요소는 항상 연속적으로 존재하며, 구성요소에 접근할 때에는 인덱스로 한다.

배열은 모든 구성요소가 서로 동일한 타입이어야 한다. REF CURSOR를 제외한 모든 타입이 될 수 있다. 또한 배열을 구성요소 타입으로 갖는 배열을 선언할 수 있다.

tbPSM에서 배열을 선언하는 방법은 다음과 같다.

```
TYPE name IS VARRAY(limit) OF type [NOT NULL];
```

배열의 초기화

배열은 초기화라는 과정을 거치게 되는데, 이 과정을 거치지 않은 변수는 항상 NULL을 가지며, 구성요소도 존재하지 않게 된다. 만약 초기화되지 않은 배열에 접근할 경우 **COLLECTION_IS_NULL** 예외 상황이 발생한다.

배열을 초기화하는 방법은 다음과 같다.

```
DECLARE
    TYPE class_arr IS VARRAY(10) OF VARCHAR2(10) NOT NULL;
    classes class_arr;
BEGIN
    classes := class_arr('math', 'physics', 'history', 'science');
END;
```

배열의 구성요소

배열은 선언할 때 지정한 길이가 구성요소의 최대 길이가 되므로, 초기화되지 않은 나머지 구성요소는 존재하지 않는 값이 된다. 만약 초기화되지 않은 구성요소에 접근할 경우에는 **BEYOND_SUBSCRIPT** 예외 상황이 발생한다. 만약 배열에서 NOT NULL 제약조건이 없을 경우에는 NULL을 구성요소로 사용할 수 있다.

```
DECLARE
    TYPE class_arr IS VARRAY(5) OF VARCHAR2(10);
    classes class_arr := class_arr('math', NULL, 'science');
BEGIN
    NULL;
END;
```

배열의 구성요소는 하나의 독립적인 변수처럼 사용할 수 있으며, 해당 변수의 값을 추출할 수 있다. 또한 대입도 가능하다.

예를 들면 다음과 같다.

```
DECLARE
    TYPE class_arr IS VARRAY(10) OF VARCHAR2(10);
    classes class_arr := class_arr('math', 'physics', 'art');
BEGIN
    classes(3) := 'history';
    IF classes(1) IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('I hate this class');
    END IF;
    DBMS_OUTPUT.PUT_LINE(classes(3) || ' is my favorite class');
END;
/
history is my favorite class

PSM completed
```

배열의 구성요소를 참조하기 위해서는 다음과 같이 사용해야 한다.

```
varray_name(index)
```

index에는 1 ~ 2^31 - 1 사이의 값만을 사용할 수 있다.

참고

배열은 IS NULL 연산자와 등호 연산자만 사용할 수 있다. 대소 비교와 같은 일반적인 비교 연산은 불가능하다.

4.2.4. 컬렉션 함수와 프러시저

tbPSM은 컬렉션 타입을 쉽고 편하게 사용하기 위해 컬렉션 함수와 프러시저를 제공한다.

구분	이름	설명
컬렉션 함수	EXISTS	컬렉션 함수는 항상 반환값을 갖는다.

구분	이름	설명
	COUNT LIMIT FIRST, LAST PRIOR, NEXT	
프러시저	EXTEND TRIM DELETE	프러시저는 반환값을 갖지 않는다.

컬렉션 함수와 프러시저를 초기화되지 않은 컬렉션 변수에 사용하는 경우 **COLLECTION_IS_NULL** 예외 상황이 발생한다.

EXISTS 함수

EXISTS 함수는 n-번째 구성요소의 존재 여부를 TRUE, FALSE로 반환한다. 이 값은 DELETE 프러시저에 의해 달라질 수 있다.

다음은 EXISTS 함수의 예이다.

```
DECLARE
    TYPE sunday_tab IS TABLE OF PLS_INTEGER;
    my_sunday sunday_tab := sunday_tab(2, 9, 16, 23, 30);
BEGIN
    IF my_sunday.EXISTS(2) THEN
        DBMS_OUTPUT.PUT_LINE('Second sunday is ' || my_sunday(2));
    END IF;
    my_sunday.DELETE(2);
    IF my_sunday.EXISTS(2) = false THEN
        DBMS_OUTPUT.PUT('Second sunday is disappeared....');
        DBMS_OUTPUT.PUT_LINE(' because of work');
    END IF;
    IF my_sunday.EXISTS(10) = false THEN
        DBMS_OUTPUT.PUT_LINE('Only 4 or 5 weeks in a month');
    END IF;
END;
/
Second sunday is 9
Second sunday is disappeared.... because of work
Only 4 or 5 weeks in a month

PSM completed
```

COUNT 함수

COUNT 함수는 컬렉션 변수의 구성요소 개수를 반환한다. 이 값은 DELETE 프러시저에 의해 달라질 수 있다.

다음은 COUNT 함수의 예이다.

```
DECLARE
    TYPE sunday_tab IS TABLE OF PLS_INTEGER;
    my_sunday sunday_tab := sunday_tab(2, 9, 16, 23, 30);
BEGIN
    DBMS_OUTPUT.PUT_LINE(my_sunday.COUNT);
    my_sunday.DELETE(2);
    DBMS_OUTPUT.PUT_LINE(my_sunday.COUNT);
END;
/
5
4

PSM completed
```

LIMIT 함수

LIMIT 함수는 컬렉션 타입에 따라 반환되는 값이 다르다.

컬렉션 타입	설명
테이블, 인덱스 테이블	구성요소의 길이 제한이 없으므로 NULL을 반환한다.
배열	배열을 선언할 때 지정된 값이 반환된다.

다음은 LIMIT 함수의 예이다.

```
DECLARE
    TYPE sunday_tab IS TABLE OF PLS_INTEGER;
    TYPE monday_arr IS VARRAY(7) OF PLS_INTEGER;
    my_sunday sunday_tab := sunday_tab(2, 9, 16, 23, 30);
    my_monday monday_arr := monday_arr(3, 10, 17, 24, 31);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Limit of table = ' || my_sunday.LIMIT);
    DBMS_OUTPUT.PUT_LINE('Limit of varrary = ' || my_monday.LIMIT);
END;
/
Limit of table =
Limit of varrary = 7

PSM completed
```

FIRST, LAST 함수

FIRST 함수는 첫 번째 구성요소의 인덱스를 반환하고, LAST 함수는 마지막 구성요소의 인덱스를 반환한다. 이 값은 DELETE 프러시저에 의해 달라질 수 있다.

FIRST, LAST 함수는 컬렉션 타입에 따라 반환되는 값이 다르다.

컬렉션 타입	설명
인덱스 테이블	– FIRST 함수 : 가장 낮은 키값을 갖는 테이블을 반환한다. – LAST 함수 : 가장 높은 키값을 갖는 테이블을 반환한다.
배열	– FIRST 함수 : 항상 1을 반환한다. – LAST 함수 : 항상 COUNT 함수와 같은 값을 반환한다.

다음은 FIRST, LAST 함수의 예이다.

```
DECLARE
    TYPE sunday_tab IS TABLE OF PLS_INTEGER;
    my_sunday sunday_tab := sunday_tab(2, 9, 16, 23, 30);
BEGIN
    DBMS_OUTPUT.PUT_LINE('First sunday is ' || my_sunday.FIRST());
    DBMS_OUTPUT.PUT_LINE('Last sunday is ' || my_sunday.LAST());
    my_sunday.DELETE(1);
    DBMS_OUTPUT.PUT_LINE('First sunday is ' || my_sunday.FIRST());
END;
/
First sunday is 1
Last sunday is 5
First sunday is 2

PSM completed
```

PRIOR, NEXT 함수

PRIOR 함수는 n-번째 구성요소의 바로 앞에 구성요소 인덱스를 반환하고, NEXT 함수는 n-번째 구성요소의 바로 다음의 구성요소 인덱스를 반환한다. 컬렉션 타입 중 인덱스 테이블은 키값의 순서에 따라 PRIOR, NEXT 함수의 값을 반환한다. 이때 키값의 순서는 기본적으로 키값의 바이너리 값을 비교할 때 정해진다.

n의 값은 EXTEND, TRIM, DELETE 프러시저에 의해 달라질 수 있다. n의 값에 따라 각 함수는 반환하는 값이 다음과 같이 다르다.

함수	n의 값	반환값
PRIOR	n < 1	NULL
	LAST보다 클 경우	항상 LAST 함수의 값을 반환한다.
NEXT	n < 1	1
	LAST보다 클 경우	NULL

다음은 PRIOR, NEXT 함수의 예이다.

```
DECLARE
    TYPE sunday_tab IS TABLE OF PLS_INTEGER;
    my_sunday sunday_tab := sunday_tab(2, 9, 16, 23, 30);
BEGIN
    DBMS_OUTPUT.PUT_LINE('First sunday is ' || my_sunday.PRIOR(2));
    DBMS_OUTPUT.PUT_LINE('Last sunday is ' || my_sunday.NEXT(1));
    my_sunday.DELETE(2);
    DBMS_OUTPUT.PUT_LINE('First sunday is ' || my_sunday.NEXT(1));
END;
/
First sunday is 1
Last sunday is 2
First sunday is 3

PSM completed
```

EXTEND 프러시저

EXTEND 프러시저는 테이블이나 배열의 크기를 늘리기 위해 사용된다.

사용하는 방법은 다음과 같이 세 가지가 있다.

방법	설명
EXTEND	하나의 NULL 구성요소를 추가한다.
EXTEND(n)	n개의 NULL 구성요소를 추가한다.
EXTEND(m, n)	n번째 구성요소를 m개 추가한다.

다음은 EXTENT 프러시저의 예이다.

```
DECLARE
    TYPE sunday_tab IS TABLE OF PLS_INTEGER;
    my_sunday sunday_tab := sunday_tab(2, 9, 16, 23, 30);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Before extend = ' || my_sunday.COUNT);
```

```
    my_sunday.EXTEND(4, 2);
    DBMS_OUTPUT.PUT_LINE('After extend = ' || my_sunday.COUNT);
END;
/
Before extend = 5
After extend = 9

PSM completed
```

TRIM 프러시저

TRIM 프러시저는 배열이나 테이블의 크기를 줄이기 위해 사용된다.

사용하는 방법은 다음과 같이 두 가지가 있다.

방법	설명
TRIM	뒤에서부터 하나의 구성요소를 제거한다.
TRIM(n)	뒤에서부터 n개의 구성요소를 제거한다. 만약 n의 크기가 큰 경우에는 SUBSCRIPT_BEYOND_COUNT 예외 상황이 발생한다.

다음은 TRIM 프러시저의 예이다.

```
DECLARE
    TYPE sunday_tab IS TABLE OF PLS_INTEGER;
    my_sunday sunday_tab := sunday_tab(2, 9, 16, 23, 30);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Before trim = ' || my_sunday.COUNT);
    my_sunday.TRIM(2);
    DBMS_OUTPUT.PUT_LINE('After trim = ' || my_sunday.COUNT);
END;
/
Before trim = 5
After trim = 3

PSM completed
```

DELETE 프러시저

DELETE 프러시저는 특정 인덱스의 구성요소를 제거하거나 전체 구성요소를 제거하기 위해 사용된다.

사용하는 방법은 다음과 같이 세 가지가 있다.

방법	설명
DELETE	전체 구성요소를 제거한다.
DELETE(n)	– 테이블 : 테이블에서 n번째 구성요소를 제거한다. – 인덱스 테이블 : 키값이 n인 값을 제거한다.
DELETE(m, n)	테이블 또는 인덱스 테이블에서 m부터 n까지의 구성요소를 제거한다. m이 n보다 크거나 m과 n 중 하나라도 NULL이면 어떠한 값도 제거되지 않는다.

다음은 DELETE 프러시저의 예이다.

```
DECLARE
    TYPE sunday_tab IS TABLE OF PLS_INTEGER;
    my_sunday sunday_tab := sunday_tab(2, 9, 16, 23, 30);
BEGIN
    my_sunday.delete(2, 4);
    IF my_sunday.EXISTS(2) THEN
        DBMS_OUTPUT.PUT_LINE('Second sunday is ' || my_sunday(2));
    END IF;
END;
/

PSM completed
```

4.2.5. 예외 상황

컬렉션 타입과 관련된 예외 상황을 정리하면 다음과 같다.

예외 상황	발생 조건
COLLECTION_IS_NULL	초기화되지 않은 컬렉션 변수에 접근하는 경우
NO_DATA_FOUND	DELETE 프러시저에 의해 제거된 구성요소에 접근하는 경우 또는 인덱스 테이블에 존재하지 않는 값에 접근하는 경우
SUBSCRIPT_BEYOND_COUNT	인덱스가 구성요소의 개수를 초과한 값으로 주어진 경우
SUBSCRIPT_OUSIDE_LIMIT	인덱스가 허용 범위를 벗어난 경우
VALUE_ERROR	인덱스가 NULL이거나 숫자의 형태로 변환이 안 되는 경우

4.3. 레코드

레코드는 관련 있는 구성요소의 집합인데, 일반적인 프로그래밍 언어의 구조체와 동일하다. 레코드에 소속된 모든 구성요소의 타입이 서로 같을 필요가 없다. 또한 필드가 다시 레코드를 포함할 수도 있다.

레코드는 주로 데이터베이스 테이블의 로우 전체를 저장하기 위해 사용된다. “2.2.5. 기타 타입”에서 설명한 내용 중에서 **%ROWTYPE**은 레코드의 대표적인 예이다.

레코드를 선언하는 방법은 다음과 같다.

```
TYPE rec_name IS
    RECORD (field1 field1_type[, field2 field2_type...]);
```

레코드는 일반적으로 다음과 같은 특성이 있다.

- 레코드는 컬렉션 타입과 달리 초기화 과정이 필요 없다.
- 선언과 동시에 메모리가 할당되어 각각의 필드에 직접 접근할 수 있다.
- 레코드 변수는 소속된 필드의 타입이 모두 같을 경우에만 대입할 수 있다.
- 레코드 내부에 포함된 각 필드는 일반적인 변수와 동일하게 취급된다.
- 레코드는 NULL의 허용 여부뿐만 아니라 등호, 부등호 등의 비교가 불가능하다.

  만약 이러한 비교가 필요하다면, 두 개의 레코드를 파라미터로 전달 받아 필드를 직접 비교하는 함수를 작성해야 한다.

다음은 레코드 선언의 예이다.

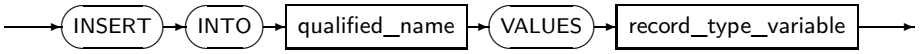
```
DECLARE
    music_info musics%ROWTYPE;
BEGIN
    SELECT * INTO music_info FROM musics WHERE kind = 'POP';
    DBMS_OUTPUT.PUT_LINE(music_info.title);
END;
```

4.4. 레코드 타입 변수의 DML

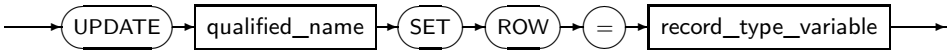
레코드는 테이블의 로우와 유사한 형태를 갖는다. 이러한 특징으로 PSM에서는 DML 문에서 특별한 문법을 사용할수 있는데, 켐코드 타입의 변수를 DML에 사용하여 ROW를 삽입 또는 변경 가능하다.

- 문법

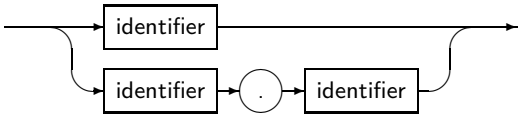
insert_record



update_setrow



record_type_variable



- 예제

```
create table student (id varchar(16), name varchar(16), grade varchar(2));

declare
    new_student student%rowtype;
    modify_student student%rowtype;
begin
    /* 1학년 학생 입학 */
    new_student.id      := '00000001';
    new_student.name    := 'studentname1';
    new_student.grade   := '1';
    insert into student values new_student;
    commit;

    /* 학생 진급 */
    modify_student.id      := '00000001';
    modify_student.name    := 'studentname1';
    modify_student.grade   := '2';
    update student set row = modify_student where id = modify_student.id;
    commit;
end;
/

select * from student;

ID               NAME             GRADE
---------------- ---------------- -----
00000001         studentname1     2

1 row selected.
```

제5장 서브 프로그램

본 장에서는 서브 프로그램을 작성하는 방법을 설명한다.

5.1. 개요

서브 프로그램(subprogram)은 다른 tbPSM 프로그램 내에서 호출할 수 있는 프로그램 블록이다. 서브 프로그램을 호출할 때에는 해당 서브 프로그램 내에서 사용할 파라미터를 함께 전달해야 한다.

서브 프로그램의 실행이 모두 끝나면 해당 서브 프로그램을 호출한 뒷부분부터 프로그램의 실행은 계속된다. 서브 프로그램은 항상 이름을 가져야 하며, 서브 프로그램 내에서 또 다른 서브 프로그램을 호출할 수 있다.

서브 프로그램을 이용함으로써 얻을 수 있는 장점은 다음과 같다.

- 모듈성(modularity)

  기능별로 각각의 모듈로 분리하여 프로그램을 작성할 수 있다. 이러한 모듈은 복잡한 모듈이 될 수도 있고 작은 모듈일 수도 있다.

- 재사용성(reusability), 관리의 편의성(maintainability)

  서브 프로그램은 정해진 특정 기능을 수행하므로, 동일한 기능을 필요로 하는 여러 프로그램에서 해당 서브 프로그램을 공통적으로 이용할 수 있다. 재사용성은 관리의 편의성(maintainability)을 의미하기도 한다.

- 추상화(abstraction)

  기능이 같다면 서브 프로그램 내부를 변경하더라도 프로그램 전체에 영향을 주지 않는다. 즉, 인터페이스에 변화가 없으며 문제가 발생되지 않는다.

5.2. 서브 프로그램의 구분

서브 프로그램은 결과 값의 반환 여부에 따라 프러시저(procedure)와 함수(function)로 구분된다.

5.2.1. 프러시저

프러시저는 함수와는 다르게 반환값이 없다.

프러시저를 선언하는 방법은 다음과 같다.

```
[CREATE [OR REPLACE]] PROCEDURE 프러시저_이름 [(파라미터[, 파라미터])]
    [AUTHID {DEFINER | CURRENT_USER}] {AS | IS}
[PRAGMA AUTONOMOUS_TRANSACTION;]
[선언부]
BEGIN
[실행부]
[예외 처리부]
END;
```

프러시저는 한 개 이상의 파라미터를 가질 수 있으며, 두 개 이상의 파라미터를 갖는 경우 각 파라미터는 콤마(,)로 분리하여 사용한다.

프러시저_이름과 동일한 이름을 갖는 프러시저가 이미 존재하는 경우에도 **OR REPLACE** 문이 삽입된 경우 해당 프러시저를 생성할 수 있다. 또한 동일한 이름을 갖는 프러시저가 이미 존재하는 경우 프러시저의 내용은 새롭게 작성된 프러시저의 내용으로 갱신된다.

다음은 새로운 직원 정보를 EMP 테이블에 삽입하는 프러시저이다.

```
CREATE OR REPLACE PROCEDURE NEW_EMP (ename VARCHAR2, deptno NUMBER) ... ① ...
IS
    salary NUMBER;
    deptno_not_found EXCEPTION;
BEGIN
    IF deptno = 5 THEN
        salary := 30000;
    ELSIF deptno = 6 THEN
        salary := 35000;
    ELSE
        RAISE deptno_not_found;                                    ... ② ...
    END IF;

    INSERT INTO EMP (ENAME, SALARY, DEPTNO)                        ... ③ ...
        VALUES (ename, salary, deptno);
EXCEPTION
WHEN deptno_not_found THEN
    -- Report Unknown Deptno
    ROLLBACK;                                                      ... ④ ...
END NEW_EMP;
```

① 두 개의 입력 파라미터를 받는다.

② 입력된 부서 번호에 따라 봉급(salary)을 결정하고, 만약 미리 정해진 부서 번호가 아니면 예외 상황을 발생시킨다. 사용자가 정의한 예외 상황을 발생시키는 것은 RAISE 문을 이용한다. 이때 사용자가 정의한 예외 상황은 선언부에 미리 정의되어 있어야 한다.

③ 봉급이 결정되면 EMP 테이블에 새로운 로우를 삽입한다.

④ 프러시저를 종료한다. 프러시저 또는 함수의 맨 마지막에 COMMIT 또는 ROLLBACK 문을 포함시킨다. 단, 하나의 트랜잭션에서 여러 프러시저나 함수를 호출하는 경우에는 모든 프러시저와 함수를 호출한 후에 커밋 또는 롤백을 수행해야 한다.

5.2.2. 함수

함수는 프러시저와는 다르게 반환값이 있다. 따라서 **RETURN 문**이 반드시 삽입되어야 한다.

함수를 선언하는 방법은 다음과 같다.

```
[CREATE [OR REPLACE]] FUNCTION 함수_이름 [(파라미터[, 파라미터])]
RETURN 반환_타입 [AUTHID {DEFINER | CURRENT_USER}]
[DETERMINISTIC]
[PARALLEL_ENABLE]
[RESULT_CACHE [RELIES_ON (데이터소스이름[,데이터소스이름])]]
[PIPELINED]
{AS | IS}
    [PRAGMA AUTONOMOUS_TRANSACTION;]
    [선언부]
BEGIN
    [실행부]
    RETURN 반환값;
    [예외 처리부]
END;
```

RETURN 문 다음에는 함수의 선언부에서 해당 함수가 반환할 값의 데이터 타입을 미리 지정해야 한다.

만약 예외 처리부가 있는 경우에는 예외 처리부가 시작하기 직전에 반환할 값을 입력하고, 예외 처리부가 없는 경우에는 실행부의 마지막에 반환할 값을 입력한다.

다음은 함수의 예이다. salary 값을 반환하는 NEW_EMP 함수이다.

```
CREATE OR REPLACE FUNCTION NEW_EMP (ename VARCHAR, deptno INT)
RETURN NUMBER IS     ... ① ...
    ...
BEGIN
    ...
    RETURN salary;   ... ② ...
EXCEPTION
    ...
END NEW_EMP;
```

① 반환값의 데이터 타입을 선언한다.

② RETURN 문장을 포함한다.

5.3. 서브 프로그램의 파라미터

5.3.1. 파라미터

실제 파라미터의 값은 프러시저가 호출될 때 프러시저에 전달되며, 프러시저 내에서 형식 파라미터를 사용하여 참조된다. 이때 값뿐만 아니라 변수에 대한 제한도 함께 전달된다.

프러시저의 내부에 선언된 변수는 다른 프러시저의 인수로 전달된다. 이를 **실제 파라미터**라고 한다. 또한 프러시저의 인수 부분에 선언된 파라미터는 **형식 파라미터**라고 한다.

- **실제 파라미터**

  실제 파라미터는 호출될 때 프러시저에 전달되는 값을 포함하고, 반환할 때 프러시저로부터 결과를 받는다. 실제 파라미터의 값은 프러시저에서 사용될 값이다.

- **형식 파라미터**

  형식 파라미터는 실제 파라미터의 값에 대한 위치 지정자이다. 프러시저가 호출될 때 형식 파라미터는 실제 파라미터의 값을 할당 받고, 프러시저 내에서 형식 파라미터에 의해 참조된다. 프러시저가 호출될 때 실제 파라미터는 형식 파라미터의 값을 할당 받는다. 이 할당은 필요하다면 타입 변환을 실행한다.

프러시저를 선언할 때는 형식 파라미터의 데이터 타입에 대한 제한이 없다. 데이터 타입에 대한 제한은 실제 프로그램 내에서 선언한 프러시저를 사용할 때 명시한다.

5.3.2. 파라미터 모드

형식 파라미터는 다음과 같이 4개의 모드를 제공한다. 만약 모드를 지정하지 않으면 디폴트는 IN 모드로 선언된다.

다음은 각 모드에 대한 설명이다.

모드	설명
IN 모드	실제 파라미터의 값은 프러시저가 호출될 때 프러시저에 전달한다. 프러시저 내에서 형식 파라미터는 읽기 전용으로 간주된다. 즉, 값을 변경할 수 없다는 뜻이다. 프러시저가 끝나고 제어가 호출 환경에 반환될 때 실제 파라미터는 변경되지 않는다.
OUT 모드	프러시저가 호출될 때 실제 파라미터가 가지는 값은 무시된다. 프러시저 내에서 형식 파라미터는 쓰기 전용으로 간주된다. 즉, 할당만 될 수 있고 읽을 수는 없다. 프러시저가 끝나고 제어가 호출 환경에 반환될 때 형식 파라미터의 내용은 실제 파라미터에 전달된다.

모드	설명
INOUT 모드	IN 모드와 OUT 모드의 조합이다. 실제 파라미터의 값은 프러시저가 호출될 때 프러시저에 전달한다. 프러시저 내에서 형식 파라미터는 읽거나 쓸 수 있다. 프러시저가 끝나고 제어가 호출 환경에 반환될 때 형식 파라미터의 내용은 실제 파라미터에 할당된다.
NOCOPY 모드	실제 파라미터가 형식 파라미터에 전달될 때 복사 여부를 결정한다. NOCOPY 모드가 선언된 경우에는 복사하지 않고, 단순히 참조(reference)만 하므로 서브 프로그램의 호출 속도를 높일 수 있다. 단, OUT 모드 또는 IN OUT 모드에만 사용할 수 있다.

5.4. 중복 선언

tbPSM에서는 서브 프로그램의 이름을 중복 선언(Overloading)하는 것을 허용한다. 단, 다음과 같은 몇 가지 제약조건이 있다.

- 중복 선언은 패키지 내의 서브 프로그램이나 선언부에 정의된 서브 프로그램 사이에서만 가능하다.
- 파라미터의 이름만 다른 경우에는 중복 선언을 할 수 없다.
- 파라미터의 모드만 다른 경우에는 중복 선언을 할 수 없다.
- 파라미터의 서브타입만 다른 경우에는 중복 선언을 할 수 없다.
- 함수에서 반환값의 타입만 다른 경우에는 중복 선언을 할 수 없다.

다음은 중복 선언의 예이다.

```
DECLARE
    PROCEDURE compute_mass(age NUMBER, name VARCHAR2) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Type of age is NUMBER');
    END;
    PROCEDURE compute_mass(age DATE, name VARCHAR2) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Type of age is DATE');
    END;
BEGIN
    compute_mass(24, 'Tibero');                 -- 첫 번째 프러시저 호출
    compute_mass(date '2010/10/24', 'Tibero');  -- 두 번째 프러시저 호출
END;
/
Type of age is NUMBER
Type of age is DATE
```

```
PSM completed
SQL>
```

5.5. 정의자 권한과 호출자 권한

tbPSM에서는 다음과 같이 두 개의 권한을 정의하고 있다.

- **정의자 권한**(Definer's Rights)

  서브 프로그램이 처음 정의될 때 해당 스키마(schema)가 가진 권한으로 항상 수행되며, 그 스키마가 소유한 객체만 볼 수 있다. 따라서 항상 같은 스키마에 같은 객체를 보게 된다.

- **호출자 권한**(Invoker's Rights)

  서브 프로그램을 수행하는 주최자(또는 호출자)의 스키마가 가진 권한으로 수행되며, 호출자가 소유한 스키마의 객체만 볼 수 있다.

  호출자 권한으로 정의한 경우 다음과 같은 장점이 있다.

  – 프로그램의 유연성

    소스 코드를 재사용할 수 있어 하나의 프로그램으로 여러 사용자에게 프로그래밍의 유연성을 제공할 수 있다.

  – 중요한 데이터 보호

    중요한 정보에 접근하는 프로그램은 정의자 권한으로 생성하고, 다시 이 프로그램을 호출자 권한으로 생성하면 다수의 사용자로부터 중요한 데이터를 보호할 수 있다.

정의자 권한으로 선언할 때에는 AUTHID 절에 **DEFINER**를 정의하고, 호출자 권한으로 선언할 때에는 **CURRENT_USER**로 정의한다.

예를 들면 다음과 같이 선언한다.

```
CREATE PROCEDURE count_total_days_per_year(year NUMBER)
AUTHID {CURRENT_USER | DEFINER} IS
   ....
BEGIN
   ....
END;
```

5.6. 캐싱 가능 함수

다음 조건에서 캐싱 가능 함수를 정의하고 있다.

- 함수내에서 패키지 변수를 사용하지 않을 때
- 함수내에서 시퀀스를 사용하지 않을 때

- 함수내의 모든 SQL이 캐싱 가능할 때
  - SQL이 캐싱 가능할 조건 : INSERT, UPDATE, DELETE, MERGE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIMESTAMP, SYSTIMESTAMP, USERENV, SYS_CONTEXT, SYS_CONTEXT, SYS_GUID, CURRENT_USER, ROWNUM, USER, UID, SYSDATE 등이 들어가지 않을 때

캐싱 가능 함수는 SQL에서 실행시 결과값을 캐싱하여 실행한다.

함수 선언 시 DETERMINISTIC 절을 명시하면 강제로 캐싱 가능 함수로 정의할 수 있다.

예를 들면 다음과 같이 선언한다.

```
CREATE OR REPLACE FUNCTION NEW_EMP (ename VARCHAR, deptno INT)
RETURN NUMBER
[DETERMINISTIC] IS
   ....
BEGIN
   ....
END;
```

5.7. RESULT CACHE 함수

RESULT_CACHE 절을 명시하면 함수 실행시 RESULT CACHE 기능을 사용한다.

RESULT CACHE 기능은 SGA(Shared Global Area) 메모리에 결과를 캐싱할수 있는 기능이다.

RELIES_ON 절은 함수 결과가 의존하는 데이터 소스들을 지정한다.

RELIES_ON 절은 더이상 사용되지 않는 기능이다. 결과 캐시 기능이 실행되는 동안 쿼리되는 모든 데이터 소스들을 감지하므로, 적는 것이 무의미하다.

함수 선언 시 RESULT_CACHE 절을 명시하면 RESULT CACHE 함수로 정의할 수 있다.

예를 들면 다음과 같이 선언한다.

```
CREATE OR REPLACE FUNCTION NEW_EMP (ename VARCHAR, deptno INT)
RETURN NUMBER
[RESULT_CACHE [RELIES_ON (데이터소스이름[,데이터소스이름])]] IS
   ....
BEGIN
   ....
END;
```

제6장 패키지

본 장에서는 tbPSM의 변수나 타입, 서브 프로그램 등을 그룹화하여 모아 놓은 객체인 패키지를 기술한다.

6.1. 패키지 구조

패키지(package)는 개념적으로 관련 있는 tbPSM의 변수나 타입, 서브 프로그램을 그룹화하여 모아 놓은 객체이다.

패키지는 보통 선언부와 구현부로 구성된다.

- 선언부

  사용자에게 보이는 인터페이스로 공개(public)적인 성격을 띤다 .

  패키지의 선언부는 다음과 같은 형식을 갖는다.

```
CREATE [OR REPLACE] PACKAGE 패키지_이름
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
    [변수, 타입 선언...]        -- 공개
    [커서 선언...]
    [함수 선언...]
    [프러시저 선언...]
END;
```

옵션	설명
AUTHID	패키지 수준으로 패키지 된 모듈이 수행하는 권한(privileges)을 지정한다.

  패키지 모듈에서는 독립형 모듈과 달리 AUTHID 절을 사용하여 각 단위를 개별적으로 다른 권한을 지정할 수 없다.

- 구현부

  선언부에서 선언한 내용을 실제로 구현하는 부분이다. 사용자에게는 구현된 내용이 보이지 않으며, 비공개적(private)인 성격을 띤다 .

  패키지 구현부는 다음과 같은 형식을 갖는다.

```
CREATE [OR REPLACE] PACKAGE BODY 패키지_이름
[AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
    [변수, 타입 선언...]        -- 비공개
    [커서 구현...]
    [함수 구현...]
```

```
    [프러시저 구현...]
[BEGIN
    초기화]
END;
```

패키지는 선언부에서 선언된 커서나 서브 프로그램 등에 대해 구현부에서 실제로 실행될 내용을 구현해야 한다. 그렇지 않으면 예외 상황이 발생한다.

패키지 구현부에서는 변수나, 타입, 커서, 예외 상황, 서브 프로그램 등을 선언하고 구현할 수 있는데, 선언부에 존재하지 않는 변수나 서브 프로그램 등은 사용자에게 공개되지 않기 때문에 사용할 수 없다.

6.2. 패키지 초기화

패키지 구현부에는 BEGIN ... END 절을 사용할 수 있다. BEGIN ... END 절 사이에는 선언부에서 명시한 변수를 초기화할 수 있고, 패키지가 처음 사용되는 시점에 단 한번 호출된다.

다음은 패키지 초기화의 예제이다.

- anniversary_manager 패키지의 선언부

```
CREATE PACKAGE anniversary_manager IS
    month_counter PLS_INTEGER;
    PROCEDURE compute_elapsed_days(start_day DATE);
END;
```

- anniversary_manager 패키지의 구현부

```
CREATE PACKAGE BODY anniversary_manager IS
    PROCEDURE compute_elapsed_days(start_day DATE) IS
    BEGIN
        month_counter := months_between(sysdate(), start_day);
        DBMS_OUTPUT.PUT_LINE(month_counter * 30 || ' days...');
    END;
BEGIN
    month_counter := 0;
    DBMS_OUTPUT.PUT_LINE('package is initialized');
END;
```

다음은 anniversary_manager 패키지를 수행한 결과이다.

```
SQL> BEGIN
       anniversary_manager.compute_elapsed_days('1979/10/27');  ... 첫 번째 호출 ...
       DBMS_OUTPUT.PUT_LINE('============ ');
       anniversary_manager.compute_elapsed_days('1998/03/02');  ... 두 번째 호출 ...
     END;
/
```

```
package is initialized
9780 days...
============
3180 days...

PSM completed
SQL>
```

위 결과에서 보듯이 첫 번째로 compute_elapsed_days 프러시저가 호출된 경우에는 패키지 구현부의 BEGIN...END 절이 수행되는 반면, 두 번째로 호출된 경우에는 수행되지 않는 것을 알 수 있다.

6.3. 패키지 객체

패키지 객체(Instance)란 패키지에 선언한(서브 프로그램을 제외한) 변수, 타입, 예외 상황, 커서 등을 말한다.

6.3.1. 패키지 객체 연속성

최초로 패키지가 참조되는 순간에 패키지 객체의 전부가 메모리에 로딩되고, 한번 로딩된 패키지 객체는 세션이 닫히기 전까지는 계속 존재하게 된다. 또한 한 명의 사용자가 서로 다른 프로그램을 호출하는 동안에도(직접 해당 변수를 수정하지 않는 조건이라면) 패키지 객체는 동일한 값을 갖는다.

다음은 패키지 객체 연속성 예제이다.

- test_instance 패키지의 선언부

```
CREATE PACKAGE test_instance IS
    total_cnt NUMBER;
    FUNCTION get_total_cnt RETURN NUMBER;
END;
```

- test_instance 패키지의 구현부

```
CREATE PACKAGE BODY test_instance IS
    FUNCTION get_total_cnt RETURN NUMBER IS
    BEGIN
        RETURN total_cnt;
    END;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Package is initialized');
    total_cnt := 0;
END;
```

다음은 test_instance 패키지를 수행한 결과이다.

```
SQL> BEGIN
       DBMS_OUTPUT.PUT_LINE('초기 값 = ' || test_instance.get_total_cnt);
       -- 패키지 객체인 total_cnt의 값을 바꾼다.
       test_instance.total_cnt := 3;    ... ⓐ ...
     END;
/
Package is initialized
초기 값 = 0

PSM completed
SQL>
```

ⓐ에서 total_cnt에 대입한 값이 계속 유지되고 있다. 이때 SQL 문장을 실행하면 다음과 같이 변경된 값의 결과가 출력된다.

```
SQL> BEGIN
       DBMS_OUTPUT.PUT_LINE(test_instance.get_total_cnt);
     END;
/
3

PSM completed
SQL>
```

6.3.2. 패키지 객체 참조범위

한 패키지 객체는 한 명의 사용자에게만 한정된다. 따라서 서로 다른 사용자는 각각 다른 객체를 참조하게 된다. test_instance 패키지의 예를 기준으로 다음과 같이 SQL 문장을 실행한다.

```
SQL> BEGIN
    DBMS_OUTPUT.PUT_LINE('초기 값 = ' || test_instance.get_total_cnt);
    -- 패키지 객체인 total_cnt의 값을 바꾼다.
    test_instance.total_cnt := 3;
    DBMS_OUTPUT.PUT_LINE('바꾼 값 = ' || test_instance.get_total_cnt);
END;
/
Package is initialized
초기 값 = 0
바꾼 값 = 3

PSM completed
SQL>
```

이러한 과정을 수행하고 나서 다른 콘솔 창을 실행하여 다시 같은 사용자로 접속한다.

```
SQL> BEGIN
       DBMS_OUTPUT.PUT_LINE(test_instance.get_total_cnt);
     END;
/
Package is initialized
0

PSM completed
SQL>
```

이전에 total_cnt의 값을 0에서 3으로 바꾼 것과는 상관없이 초기값 0이 그대로 유지되고 있는 것을 확인할 수 있다.

6.4. 패키지 서브 프로그램 중복 선언

패키지의 서브 프로그램은 중복 선언을 할 수 있다. 다음은 패키지 서브 프로그램의 중복 선언에 대한 예이다.

- calculator 패키지의 선언부

```
CREATE PACKAGE calculator IS
    ...
    FUNCTION add (x NUMBER, y NUMBER) RETURN NUMBER;
    FUNCTION add (x NUMBER, y VARCHAR2) RETURN NUMBER;
    ...
END;
```

- calculator 패키지의 구현부

```
CREATE PACKAGE BODY calculator IS
    ...
    FUNCTION add (x NUMBER, y NUMBER) RETURN NUMBER IS
    BEGIN
        RETURN x + y;
    END;
    FUNCTION add (x NUMBER, y VARCHAR2) RETURN NUMBER IS
    BEGIN
        RETURN x + to_number(y);
    END;
END;
```

6.5. SERIALLY RESUABLE 패키지

SERIALLY RESUABLE 패키지는 패키지 객체(Instance)의 상태가 서버 호출 동안 유지된다. 사용할 때마다 패키지 객체가 초기화되며 이전 서버 호출에서 변경한 패키지의 상태는 유지되지 않는다.

다음은 SERIALLY RESUABLE 패키지 예제이다.

- sr_pkg 패키지의 선언부

```
CREATE PACKAGE sr_pkg IS
    PRAGMA SERIALLY_REUSABLE;
    n NUMBER := 1;
    FUNCTION f1 RETURN NUMBER;
END;
```

- sr_pkg 패키지의 구현부

```
CREATE PACKAGE BODY sr_pkg IS
    PRAGMA SERIALLY_REUSABLE;
    FUNCTION f1 RETURN NUMBER IS
    BEGIN
        RETURN sr_pkg.n;
    END;
END;
```

다음은 sr_pkg 패키지를 수행한 결과이다.

```
SQL> BEGIN
       sr_pkg.n := 10;
       DBMS_OUTPUT.PUT_LINE(sr_pkg.n);
     END;
/
10

PSM completed
SQL>

BEGIN
   DBMS_OUTPUT.PUT_LINE(sr_pkg.n);
 END;
/
1

PSM completed
SQL>
```

위 결과에서 보듯이 SERIALLY_REUSABLE 패키지의 경우 패키지 객체(Instance)의 상태가 서버 호출 동안만 유지된다.

다음은 SQL 문을 통해 sr_pkg 패키지에 접근하는 경우 예제이다.

```
SQL> SELECT sr_pkg.f1() FROM DUAL;
/
```

```
TBR-15240: Unable to access the SERIALLY REUSABLE 'SR_PKG' package.
SQL>
```

SERIALLY_REUSABLE 패키지를 SQL 문을 통해서 접근하려고 하면 오류가 발생한다.

6.6. 시스템 패키지

tbPSM에서는 사용자의 편의를 위해 STANDARD,DBMS_LOB, DBMS_OUTPUT, UTL_RAW 패키지 등의 **시스템 패키지**를 제공한다.

STANDARD 패키지는 tbPSM에서 사용하는 모든 타입과 예외 상황, 시스템 서브 프로그램을 정의한다. 만약 사용자의 실수로 이 패키지가 삭제되거나 내용이 변경된다면 tbPSM 프로그램의 수행 자체가 실패하거나 잘못된 결과를 출력할 수 있으므로 주의해야 한다.

STANDARD 패키지를 직접 수정하지 않고, 사용자의 편의에 따라 같은 이름의 서브 프로그램이나 패키지를 생성할 수 있는데, 이러한 경우 우선순위에 따라 항상 사용자 패키지나 서브 프로그램이 우선권을 가지게 된다. 따라서 STANDARD 패키지를 다시 사용하려면 STANDARD.XXX 형식으로 생성하면 된다.

참고

DBMS_LOB, DBMS_OUTPUT, UTL_RAW 패키지에 대한 자세한 내용은 "Tibero tbPSM 참조 안내서"를 참고한다.

제7장 SQL 문장의 실행

본 장에서는 tbPSM 프로그램에서 SQL 문장을 실행하는 방법을 설명한다.

tbPSM은 SQL 문장의 범주 중에서 데이터 정의어(Data Definition Language, 이하 DDL)와 데이터 조작어(Data Manipulation Language, 이하 DML), 데이터 제어어(Data Control Language, 이하 DCL)에 속한 트랜잭션 제어의 일부를 프로그램 내부에서 실행할 수 있도록 지원한다.

7.1. DCL

트랜잭션은 데이터베이스를 조회하거나 갱신 또는 조작 등을 하는 처리의 기본 단위이다. 즉, SQL 문장의 처리가 성공하거나 실패하는 일련의 과정을 말한다. 트랜잭션은 RDBMS의 표준으로써, 일관되지 않은 데이터가 생성되는 것을 방지한다.

트랜잭션은 원자성(Atomic), 일관성(Consistent), 고립성(Isolated), 영속성(Durable) 등의 특성에 의해 데이터베이스 실행이 보장된다.

트랜잭션은 다음과 같은 특징이 있다.

- 트랜잭션은 COMMIT이나 ROLLBACK 문으로 끝난다.

  tbPSM 프로그램은 사용자의 필요에 따라 실행 코드부에는 COMMIT 문을 포함시키고, 에러 처리부에는 ROLLBACK 문을 포함시킨다. 특히 에러 처리부에서는 에러 처리뿐만 아니라 현재 트랜잭션이 갱신한 내용도 취소할 수 있다.

  다음은 COMMIT, ROLLBACK 문의 예이다.

```
COMMIT;

ROLLBACK WORK RELEASE;
```

- ROLLBACK 문을 이용하여 현재 트랜잭션이 갱신한 내용의 일부만을 취소할 수 있다. 이를 위해 저장점(savepoint)을 미리 설정해야 한다.

  다음은 SAVEPOINT 문의 예이다. SAVEPOINT 문은 ROLLBACK 문에 반드시 포함시켜야 한다.

```
SAVEPOINT sp1;
...
ROLLBACK TO SAVEPOINT sp1;
```

- 트랜잭션에 고립성의 수준을 설정할 수 있다. 고립성은 트랜잭션의 특성 중 하나로 임의의 트랜잭션이 동시에 실행되는 다른 트랜잭션에 영향을 주어서는 안 된다는 것이다.

  다음은 현재 트랜잭션에 고립성의 수준을 설정하고 읽기 작업만을 수행하도록 하는 예이다.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
... 고립성의 수준(직렬화)을 설정한다. ....

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
... 읽기 작업만을 수행한다. ...
```

- 트랜잭션에 잠금을 설정할 수 있다. 갱신 또는 삭제를 하기 전에 질의 결과의 컬럼에 잠금을 설정하여, 다른 트랜잭션이 액세스하지 못하도록 막으려면 SELECT FOR UPDATE 문을 사용해야 한다.

  다음은 테이블 EMP의 DEPNO 컬럼에 잠금을 설정하는 예이다.

  ```
  SELECT * FROM EMP WHERE DEPTNO = deptno FOR UPDATE;
  ```

  만약 질의가 두 개의 테이블에 조인을 수행하는 데 이중 하나의 테이블에만 잠금을 설정하고 싶다면 FOR UPDATE OF 절을 사용해야 한다. 사용하는 방법은 FOR UPDATE OF 절에 잠금 설정을 원하는 테이블의 컬럼 이름을 포함하면 된다.

  다음은 EMP, DEPT 테이블 중에서 EMP 테이블의 SALARY 컬럼에만 잠금을 설정하는 예이다.

  ```
  SELECT ENAME, SALARY, DNAME FROM EMP E, DEPT D
  WHERE E.DEPTNO = D.DEPTNO AND E.DEPTNO = deptno FOR UPDATE OF SALARY;
  ```

7.1.1. COMMIT

COMMIT 문을 사용하면 트랜잭션이 끝남과 동시에 해당 트랜잭션에서 수행된 모든 작업이 데이터베이스에 영구히 반영된다. 또한 다른 세션이 해당 트랜잭션에 의해 수행된 내용을 볼 수 있으며, 트랜잭션에 설정된 잠금이 해제된다.

COMMIT 문을 선언하는 방법은 다음과 같다.

```
COMMIT [task];
```

옵션	설명
task	SQL 문장의 가독성을 높이기 위해 추가하는 옵션이다.

7.1.2. ROLLBACK

ROLLBACK 문을 사용하면 트랜잭션이 끝남과 동시에 해당 트랜잭션이 수행한 모든 작업은 취소되며, 트랜잭션이 가지고 있는 잠금을 해제할 수 있다. 세션이 COMMIT 문이나 ROLLBACK 문을 사용하여 현재 트랜잭션을 끝내지 않고 데이터베이스 연결을 끊는다면, 트랜잭션은 시스템에 의해 자동으로 롤백된다.

ROLLBACK 문을 선언하는 방법은 다음과 같다.

```
ROLLBACK [task];
```

옵션	설명
task	SQL 문장의 가독성을 높이기 위해 추가하는 옵션이다.

7.1.3. SAVEPOINT

일반적으로 ROLLBACK 문은 트랜잭션의 전체를 취소한다. 그러나 SAVEPOINT 문을 사용하면 트랜잭션의 일부만을 취소할 수 있다.

SAVEPOINT 문을 선언하는 방법은 다음과 같다.

```
SAVEPOINT savepoint_name;
```

항목	설명
savepoint_name	저장점의 이름으로 tbPSM의 식별자이다. – 트랜잭션 전체에서 사용되므로 선언부에 선언하지 않는다. – ROLLBACK 문에서 사용되며 설정된 저장점 이후의 내용을 모두 취소할 수 있다.

저장점이 정의되면, 다음과 같은 문법을 통해 정의된 저장점까지 롤백할 수 있다.

```
ROLLBACK [task] TO SAVEPOINT savepoint_name;
```

ROLLBACK TO SAVEPOINT 문을 사용하면 저장점 이후로 수행된 모든 작업을 롤백할 수 있다. ROLLBACK TO SAVEPOINT 문이 실행된 이후에도 저장점은 유지되므로 동일한 저장점으로 다시 롤백할 수 있다. 이때 저장점 이후의 SQL 문장이 가지고 있던 잠금과 자원은 모두 반환된다. 그러나 ROLLBACK TO SAVEPOINT 문을 사용해도 트랜잭션은 끝나지 않는다.

7.1.4. 자율 트랜잭션

자율 트랜잭션(Autonomous Transaction)이란 기존에 이미 존재하는 트랜잭션과는 완전히 다른 독립적인 트랜잭션이다. 자율 트랜잭션에 의해 만들어진 내용은 기존의 트랜잭션에 전혀 영향을 주지도 받지도 않기 때문에 자율 트랜잭션 내에서 롤백을 수행하더라도 기존 트랜잭션과는 관계없다.

이러한 자율 트랜잭션은 현재 트랜잭션이 수행되는 중간에 독립적으로 작업하고자 할 때 유용하며 주로 서브 프로그램으로 만들어 놓고, tbPSM 프로그램을 수행하는 중간에 그 서브 프로그램을 호출함으로써 독립적인 트랜잭션 작업을 수행하게 된다.

자율 트랜잭션을 선언할 수 있는 부분은 한정적이다. 따라서 다음과 같은 위치에서만 선언할 수 있다.

- 이름 없는 블록(Anonymous block)의 중첩되지 않은 최외곽 블록
- 독립적인 서브 프로그램과 로컬 서브 프로그램
- 패키지 서브 프로그램
- 트리거

자율 트랜잭션을 선언하는 방법은 다음과 같다.

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

자율 트랜잭션은 아래와 같이 선언부에 선언된다.

```
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO emp VALUES (1, 'Peter');
    ...
END;
```

자율 트랜잭션도 일반적인 트랜잭션과 마찬가지로 DCL을 사용하여 제어할 수 있다. 예를 들면 다음과 같다.

```
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO emp VALUES (1, 'Peter');
    update_dept(1, 'marketing');
    COMMIT;
EXCEPTION
WHEN NO_DATA_FOUND THEN
    ROLLBACK;
END;
```

자율 트랜잭션을 사용할 때 다음과 같은 경우에 주의해야 한다.

- 기존 트랜잭션이 사용하고 있는 자원(예: LOCK 등)에 접근하려고 할 때 교착 상태(deadlock)가 발생한다.
- 자율 트랜잭션 내부에서 발생한 예외 상황이 처리되지 않았을 때 그 동안 수행된 내용이 롤백된다.
- 커밋이나 롤백을 하지 않은 채로 자율 트랜잭션을 종료할 때 예외 상황이 발생한다.

다음은 트리거에서 자율 트랜잭션을 사용한 예이다. 일반적인 트리거와는 달리 자율 트랜잭션을 사용하면 **EXECUTE IMMEDIATE 절**을 사용할 수 있다.

```
CREATE TRIGGER multiply_trg
AFTER INSERT ON calc
FOR EACH ROW
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO result VALUES (:new.c1 * :new.c2);
    EXECUTE IMMEDIATE 'DROP TABLE tmp_result';
END;
```

7.2. Dynamic SQL

일반적으로 tbPSM에서는 DDL이나 GRANT와 같은 DCL, ALTER SESSION과 같은 세션 관리 제어 문장을 사용할 수 없다. 하지만 이를 가능하게 해 주는 것이 **Dynamic SQL**이다.

Dynamic SQL은 **EXECUTE IMMEDIATE 절**에서 사용할 수 있다.

```
EXECUTE IMMEDIATE sql_stmt
    [INTO id_list USING id_list RETURNING INTO id_list];
```

다음은 Dynamic SQL를 사용한 예이다.

```
DECLARE
    sql_stmt  VARCHAR2(2000);
    emp_id    NUMBER := 1;
    emp_name  VARCHAR2(10) := 'John';
BEGIN
    sql_stmt := 'INSERT INTO emp VALUES (:1, :2)';
    EXECUTE IMMEDIATE sql_stmt USING emp_id, emp_name;
    sql_stmt := 'SELECT name FROM emp WHERE id = :id';
    EXECUTE IMMEDIATE sql_stmt INTO emp_name USING emp_id;
    EXECUTE IMMEDIATE 'CREATE TABLE dept (id NUMBER)';
END;
```

주의

일반적으로 SQL 문장을 작성할 때에는 세미콜론(;)이 필요 없지만, tbPSM에서 Dynamic SQL를 사용할 때에는 반드시 세미콜론이 있어야 한다.

7.3. 커서

커서(Cursor)란 tbPSM에서 SQL을 수행하기 위해 하나의 문장마다 사용하는 내부 구조를 말한다.

일반적으로 커서에는 묵시적 커서와 명시적 커서, 커서 변수가 있다. 또한 %ISOPEN, %FOUND, %NOT FOUND, %ROWCOUNT의 특별한 속성을 제공한다.

7.3.1. 묵시적 커서

묵시적 커서(Implicit cursor)란 INSERT, UPDATE, DELETE 문장과 SELECT INTO를 비롯한 SQL 문장을 실행할 때 tbPSM에서 내부적으로 사용하는 커서를 말한다. 하나의 SQL 문장을 수행할 때마다 이 커서가 열리고 닫힌다.

DML

tbPSM에서는 제약 없이 DML을 사용할 수 있다.

```
BEGIN
    INSERT INTO emp VALUES (1, 'Susan');
    UPDATE emp SET name = 'Peter' WHERE id = 1;
    DELETE FROM emp WHERE id = 1;
END;
```

QUERY

tbPSM에서는 SELECT한 SQL 문장의 결과를 변수로 저장할 수 있다. 그러나 일반적인 SELECT INTO를 사용하는 경우 아래 예와 같이 한 번에 한 개의 컬럼 밖에 얻어오지 못한다.

```
DECLARE
    name VARCHAR2(20);
BEGIN
    SELECT name INTO name FROM emp WHERE id = 2;
END;
```

결과 컬럼이 두 개 이상일 경우에는 TOO_MANY_ROWS 예외 상황이 발생하고, 이와는 반대로 결과 컬럼이 없을 경우에는 NO_DATA_FOUND 예외 상황이 발생한다.

tbPSM에서는 이러한 문제를 해결하기 위해 결과 컬럼이 여러 개인 경우에도 SQL 문장이 실행될 수 있는 기능을 제공한다. 이 기능을 **커서 반복문**(CURSOR_FOR_LOOP)이라고 한다.

다음은 커서 반복문의 예이다.

```
BEGIN
    FOR result IN (SELECT * FROM emp) LOOP
```

```
        DBMS_OUTPUT.PUT_LINE(result.id || '번 직원=' || result.name);
    END LOOP;
END;
```

7.3.2. 명시적 커서

명시적 커서(Explicit cursor)란 사용자가 tbPSM 프로그램 내에서 직접 선언한 커서를 말한다. 이 커서는 전적으로 사용자에 의해 OPEN, FETCH, CLOSE가 이루어지며, tbPSM은 관여하지 않는다.

명시적 커서는 일반적인 tbPSM의 변수와 달리 대입할 수 없고, 선택적으로 파라미터를 가질 수 있다. 이 파라미터는 이후에 실행되는 SQL에서 사용된다.

명시적 커서를 선언하는 방법은 다음과 같다.

```
CURSOR cursor_name IS SELECT ...;
CURSOR cursor_name (param1 TYPE [DEFAULT VALUE], ...) IS SELECT ...;
```

OPEN

명시적 커서를 열기 위해서는 **OPEN 문**을 사용해야 한다.

사용 방법은 다음과 같다.

```
DECLARE
    CURSOR c1 IS SELECT * FROM emp;
BEGIN
    OPEN c1;
END;
```

FETCH

명시적 커서를 Fetch하는 방법은 **FETCH INTO 문**을 사용하며 한 번에 하나의 로우만 받아올 수 있다.

사용 방법은 다음과 같다.

```
DECLARE
    emp_rec emp%ROWTYPE;
    CURSOR c1 IS SELECT * FROM emp;
BEGIN
    OPEN c1;
    FETCH c1 INTO emp_rec;
END;
```

CLOSE

명시적 커서를 닫기 위해서는 **CLOSE 문**을 사용한다. 만약 이미 닫혔거나 열리지 않은 커서에 대해 CLOSE를 수행한다면, **INVALID_CURSOR** 예외 상황이 발생한다.

```
DECLARE
    emp_rec emp%ROWTYPE;
    CURSOR c1 IS SELECT * FROM emp;
BEGIN
    OPEN c1;
    FETCH c1 INTO emp_rec;
    CLOSE c1;
END;
```

커서 반복문의 사용

명시적 커서도 커서 반복문을 사용할 수 있다.

사용 방법은 다음과 같다.

```
DECLARE
    CURSOR c1 IS SELECT * FROM emp;
BEGIN
    FOR result IN c1 LOOP
        DBMS_OUTPUT.PUT_LINE(result.id || '번 직원=' || result.name);
    END LOOP;
END;
```

패키지에서의 정의

패키지에서는 선언부와 구현부로 나누어 커서를 정의한다.

사용 방법은 다음과 같다.

```
CREATE OR REPLACE PACKAGE pkg
IS
    TYPE rec IS record (a number, b varchar2(100), c date);
    CURSOR c1 RETURN rec;
END;
/
CREATE OR REPLACE PACKAGE BODY pkg
IS
    CURSOR c1 RETURN rec IS select 1, 'tibero', sysdate from dual;
END;
/
DECLARE
```

```
    a NUMBER;
    b VARCHAR2(100);
    c DATE;
BEGIN
    OPEN pkg.c1;
    FETCH pkg.c1 INTO a, b, c;
    DBMS_OUTPUT.PUT_LINE(a||','||b||','||c);
    CLOSE pkg.c1;
END;
/
```

다음은 명시적 커서의 상태에 따른 커서 속성 값이다.

상태	%ISOPEN	%FOUND	%NOTFOUND	%ROWCOUNT
Before OPEN	false	INVALID_CURSOR	INVALID_CURSOR	INVALID_CURSOR
After OPEN	true	NULL	NULL	0
After first FETCH	true	true	false	1
After next FETCH	true	true	false	임의의 값
After last FETCH	true	false	true	임의의 값
After CLOSE	false	INVALID_CURSOR	INVALID_CURSOR	INVALID_CURSOR

7.3.3. 커서 변수

커서 변수(Cursor variable)란 특정 SQL 문장에만 해당되지 않고, 커서를 열 때 주어진 SQL 문장에 따라 자유롭게 변경하여 사용할 수 있으며, 일반적인 변수처럼 서브 프로그램의 파라미터의 반환값으로 사용되는 커서를 말한다. 커서 변수를 사용하면 많은 양의 데이터를 주고 받는 대신 포인터만 전달하므로 프로그램의 성능을 개선할 수 있다.

커서 변수를 선언하는 방법은 다음과 같다.

```
TYPE csr_type_name IS REF CURSOR [RETURN type_name];

csr_var_name csr_type_name;
```

항목	설명
csr_type_name	REF CURSOR 타입의 이름이다.
[RETURN type_name]	선택적으로 사용된다. – RETURN 절이 있는 경우에는 **타입의 안정성을 갖는 커서**가 생성된다.

항목	설명
	– type_name 즉 질의의 반환 타입은 RETURN 절에서 정한 타입과 호환되어야 한다.
csr_var_name	커서 변수의 이름이다. 커서 변수를 선언할 때는 먼저 REF CURSOR 타입을 선언한 후에 이 타입으로 변수를 선언한다.

다음은 커서 변수를 선언한 예이다.

```
DECLARE
    TYPE ref_csr IS REF CURSOR;
    c1 ref_csr;
```

선언된 커서 변수를 열기 위해서는 다음과 같이 사용해야 한다.

```
OPEN csr_var_name FOR sql_query;
```

다음은 선언한 커서 변수를 여는 예이다.

```
DECLARE
    TYPE ref_csr IS REF CURSOR;
    c1 ref_csr;
BEGIN
    OPEN c1 FOR SELECT * FROM emp;
    OPEN c1 FOR SELECT part FROM dept;
```

커서 변수를 사용할 때 커서의 FETCH와 CLOSE의 사용 방법은 “7.3.2. 명시적 커서”와 같다.

커서 변수는 다음과 같은 제약조건이 있다.

- 커서 변수는 NULL 허용 여부 검사나 등호 또는 부등호 비교를 할 수 없다.
- 커서 변수에는 NULL을 대입할 수 없다.
- 커서 변수는 테이블(nested table)이나 배열(varray)의 요소가 될 수 없다.
- 커서 변수는 커서 반복문에서 사용될 수 없다.

7.3.4. 커서 속성

tbPSM은 하나의 SQL 문장을 수행하고 그 결과를 속성으로 저장하여 사용자에게 확인할 수 있도록 4가지의 특별한 커서 속성(Cursor attribute)을 제공한다.

커서 속성은 묵시적 커서, 명시적 커서에 따라 그 사용 방법이 다르다.

- 묵시적 커서

```
SQL%{ISOPEN | FOUND | NOTFOUND | ROWCOUNT}
```

- 명시적 커서

```
cursor_name%{ISOPEN | FOUND | NOTFOUND | ROWCOUNT}
```

%ISOPEN

%ISOPEN은 커서가 OPEN되었는지 여부를 반환하는 커서 속성이다. 묵시적 커서의 경우 SQL 문장을 수행하고 나서 항상 닫히므로 무조건 FALSE이다. 반면에 명시적 커서의 경우 사용자가 CLOSE를 했는지의 여부에 따라 반환값이 달라진다.

```
BEGIN
    INSERT INTO emp VALUES (1, 'Susan');
    IF SQL%ISOPEN = FALSE THEN
        DBMS_OUTPUT.PUT_LINE('묵시적 커서는 항상 자동으로 닫힌다.');
    END IF;
END;
```

%FOUND, %NOTFOUND

%FOUND와 %NOTFOUND는 INSERT, UPDATE, DELETE에 의해 영향을 받은 로우가 있는지 없는지, SELECT INTO에 의한 결과 로우가 있는지 없는지를 반환하는 커서 속성이다.

```
BEGIN
    UPDATE emp SET name = 'John' WHERE id = 1;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('1번 직원은 존재하지 않는다.');
    END IF;
END;
```

%ROWCOUNT

%ROWCOUNT는 INSERT, UPDATE, DELETE에 의해 영향을 받은 로우의 개수나 SELECT INTO의 결과 로우의 개수를 반환하는 커서 속성이다.

```
BEGIN
    DELETE FROM emp WHERE id = 1;
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || '로우가 삭제되었으므로 존재하지 않는다.');
```

```
    END IF;
END;
```

7.3.5. SYS_REFCURSOR

SYS_REFCURSOR는 커서 변수를 사용할 때 사용자의 편의를 위해 표준 패키지에서 SYS_REFCURSOR라는 타입을 제공하고 있다.

SYS_REFCURSOR는 전역 타입이므로, 사용자가 별도로 약한 타입의 참조 커서 타입을 선언하지 않아도 바로 사용할 수 있다.

```
DECLARE
  cx SYS_REFCURSOR;
BEGIN
  ...
END;
/
```

제8장 에러 처리

본 장에서는 tbPSM 프로그램에서 발생하는 에러를 처리하는 방법을 설명한다.

8.1. 개요

에러는 tbPSM 프로그램에서 다음과 같은 경우에 발생할 수 있다.

- **컴파일 중일 때**

  컴파일 중에 발생하는 에러는 tbPSM이 발견하여 사용자에게 보고한다. 이 경우 프로그램은 아직 실행되지 않은 상태이며, 사용자에 의해 수정되어야 한다.

- **프로그램이 실행 중일 때**

  에러를 처리하기 위해서는 **예외 상황**과 **예외 처리 루틴**을 사용한다.

  에러가 발생하면 프로그램은 비정상적으로 종료되며, 사용자는 원하는 결과를 얻을 수 없다. 따라서 에러 발생이 예상되는 부분을 예외 상황으로 정의하고 예외 처리 루틴을 만든다. 그러면 프로그램을 작성하거나 디버깅할 때 편리하게 작업할 수 있다.

  다음은 프로그램이 실행 중일 때 발생한 에러를 처리하는 예이다.

```
DECLARE
    a NUMBER := 5;
    b NUMBER := 0;
BEGIN
    a := a / b;
END;
```

위의 프로그램을 실행하면 다음과 같은 에러가 발생하고 프로그램은 비정상적으로 종료된다.

```
TBR-5070: Divide by zero error.
TBR-15163: Unhandled exception at line 5.
```

에러가 발생하면 다음과 같이 예외 처리 루틴을 만든다.

```
DECLARE
    a NUMBER := 5;
    b NUMBER := 0;
BEGIN
    a := a / b;
EXCEPTION
WHEN OTHERS THEN
```

```
    handle_numeric_error(b);
END;
```

위 예에서 보듯이 에러가 발생하면 예외 처리 루틴으로 제어가 이동되고 handle_numeric_error 함수가 호출된다. 이 프로그램은 예외 처리 루틴에 의해 자동으로 에러가 처리되며, 정상적으로 종료된다.

8.2. 예외 상황 선언

발생된 에러를 예외 상황으로 정의하기 위해서는 우선 먼저 예외 상황을 선언해야 한다. 선언하는 방법은 tbPSM의 변수의 선언과 같다.

8.2.1. 시스템 정의 예외

tbPSM 프로그램에서 발생하기 쉬운 에러를 시스템에 미리 정의하는 것을 **시스템 정의 예외**라고 한다.

시스템 정의 예외는 사용자가 프로그램의 선언부에 예외 상황을 명시적으로 선언하지 않아도 사용할 수 있다. 또한 에러와 예외 상황이 자동으로 연결되어 있어 사용자가 명시적으로 연결하지 않아도 에러가 발생하면 자동적으로 예외 상황이 된다.

시스템 정의 예외를 선언하는 방법은 다음과 같다.

```
DECLARE
    exception_identifier EXCEPTION;
```

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.
COLLECTION_IS_NULL	초기화되지 않은 컬렉션 변수의 요소에 값을 대입하려는 경우이다.

예외 상황	설명
	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.

다음은 시스템 정의 예외의 예이다.

```
DECLARE
    employee_num NUMBER := 980180;
    employee_grade VARCHAR2(10) := 'S';
BEGIN
    CASE employee_grade
    WHEN 'A' THEN pay_bonus_a(employee_num);
    WHEN 'B' THEN pay_bonus_b(employee_num);
    WHEN 'C' THEN pay_bonus_c(employee_num);
    WHEN 'D' THEN pay_bonus_d(employee_num);
    END CASE;
EXCEPTION
WHEN case_not_found THEN
    pay_bonus_special(employee_num);
END;
```

본 예에서는 employee_grade 변수에 **'S'**라는 값을 할당하였으며, CASE 문에서 'S'를 찾지 못하면 이 프로그램은 에러를 발생하게 된다. tbPSM은 이러한 에러를 처리하기 위해 **case_not_found**라는 시스템 정의 예외를 정의하였으며, 사용자는 예외 상황을 선언하는 과정 없이 바로 사용할 수 있다. 또한 에러 처리 루틴도 정의할 수 있다. 만약 이 예에서 에러가 발생한다면 사용자가 정의한 예외 처리 루틴으로 이동하게 되고, 프로그램은 정상적으로 종료된다.

8.2.2. 사용자 정의 예외

시스템 정의 예외 이외에 프로그램을 실행하는 중에 발생할 가능성이 있는 예외 상황을 **사용자 정의 예외**로 선언할 수 있다. 이러한 예외 상황은 일반적인 변수와 같이 프로그램의 선언부에 선언할 수 있다.

사용자 정의 예외를 선언하는 방법은 다음과 같다.

```
DECLARE
    exception name EXCEPTION;
```

exception name은 사용자가 원하는 이름으로 선언할 수 있다. 선언된 예외 상황은 일반적인 변수처럼 영역을 갖게 된다. 예외 상황의 영역은 선언된 블록의 끝까지이다.

사용자 정의 예외를 사용하는 방법에는 다음과 같이 두 가지가 있다.

- **RAISE 문 사용**

  시스템 정의 예외와는 다르게 사용자 정의 예외는 에러와 예외 상황을 연결하기 위해서는 RAISE 문을 사용해야 한다. RAISE 문에서 사용자 정의 예외를 사용하는 문법은 다음과 같다.

```
RAISE exception name;
```

  사용자는 원하는 위치에 RAISE 문을 사용하여 예외 상황을 발생시킬 수 있다. 만약 예외 상황이 발생되면 제어는 예외 처리 루틴으로 이동하게 된다. 다음은 RAISE 문의 예이다.

```
DECLARE
    too_many_guest EXCEPTION;
    cur_guest BINARY_INTEGER;
    max_guest BINARY_INTEGER;

BEGIN
    /* guest_info 테이블에서 local이 New York인 현재 고객 수와 최대 고객 수를 찾는다. */
    SELECT current_guest, max_guest INTO cur_guest, max_guest
        FROM guest_info WHERE local = 'New York';

    IF cur_guest > max_guest THEN
        /* 만약 현재 고객 수가 최대 고객 수를 초과하면 예외 상황을 발생시킨다. */
        RAISE too_many_guest;
    END IF;
END;
```

- **RAISE_APPLICATION_ERROR 프러시저 사용**

  RAISE_APPLICATION_ERROR 프러시저에서 사용자 정의 예외를 사용하는 문법은 다음과 같다.

```
RAISE_APPLICATION_ERROR
(
  error_number, error_message[, {true|false}]
);
```

항목	설명
error_number	-20000 ~ -20999 사이의 값만 사용할 수 있다.
error_message	에러 메시지의 길이는 712bytes까지 가능하다. 712bytes를 초과하는 메시지는 712bytes까지만 사용된다.
{true|false}	– true: 기존에 존재하는 에러에 추가한다. – false: 기존에 존재하는 에러를 모두 삭제한다.

다음은 RAISE_APPLICATION_ERROR 프러시저의 예이다.

```
DECLARE
    man_cnt   NUMBER;
    woman_cnt NUMBER;
BEGIN
    SELECT count(*) INTO man_cnt   FROM class WHERE gender = 'M';
    SELECT count(*) INTO woman_cnt FROM class WHERE gender = 'F';
    IF man_cnt != woman_cnt THEN
        RAISE_APPLICATION_ERROR(-20101, '남자와 여자의 비율이 맞지 않는다.');
    END IF;
END;
```

8.3. 예외 처리 루틴

예외 처리 루틴은 다음과 같은 절차로 실행된다.

1. 예외 상황이 발생하면 예외 처리 루틴으로 이동한다.
2. 예외 처리 루틴이 실행된다.
3. 실행이 끝나면, 외부 블록으로 이동한다.
4. 외부 블록을 실행한다. 만약 외부 블록이 없으면 해당 프로그램은 종료된다.
5. 프로그램을 종료한다.

8.3.1. 예외 처리 루틴 형식

시스템 정의 예외나 사용자 정의 예외와 관계없이 예외 처리 루틴은 동일한 형식으로 사용된다.

예외 처리 루틴은 선언부, 실행부와 같이 tbPSM 프로그램 내부의 블록의 한 종류로 표현된다.

```
BEGIN
    ...
EXCEPTION
WHEN A THEN
    실행문
WHEN B THEN
    실행문
WHEN C THEN
    실행문
...
END;
```

예외 처리 루틴의 형식은 다음과 같은 특징을 가진다.

- EXCEPTION으로 시작하고 해당 블록이 끝나는 END 사이에 정의 된다.
- EXCEPTION의 바로 앞에서 프로그램의 실행부는 끝난다.
- 예외 처리 루틴 내부에 WHEN 문을 사용한다.

  해당 블록에서 발생할 가능성이 있는 예외 상황을 구분하여 처리할 각 실행문(또는 실행문의 집합)을 설정한다. 또한 WHEN 문에서 서로 다른 예외 상황들은 **OR**로 묶어서 처리할 수 있다.
- WHEN 문에 정의된 실행문은 앞에서부터 순차적으로 실행된다.

  만약 발생된 예외 상황에 실행문이 예외 처리 루틴 안에 정의되어 있지 않으면 새로운 예외 상황이 발생하게 된다.
- 정의되지 않은 예외 상황을 처리하려면 OTHERS 문을 사용한다.

  OTHERS 문은 이전에 명시된 예외 상황을 제외한 모든 예외 상황을 처리하기 때문에 예외 처리 루틴의 맨 마지막에 사용한다. OTHERS 문 이후에 또 다른 예외 처리 핸들러가 오는 경우 새로운 예외 상황이 발생한다.

  OTHERS 문을 사용하는 문법은 다음과 같다.

```
BEGIN
    ...
EXCEPTION
WHEN A THEN
    실행문;
WHEN B THEN
    실행문;
...
WHEN OTHERS THEN
    실행문;
END;
```

8.3.2. 예외 상황 전파

예외 상황은 tbPSM 프로그램의 모든 부분 즉 tbPSM 블록의 선언부, 실행부, 예외 처리 루틴에서 발생할 수 있다. 대체로 tbPSM의 블록의 선언부나 예외 처리 루틴에서 예외 상황이 발생하는 경우 또는 발생한 예외 상황에 대해 예외 처리 루틴이 없는 경우 등이 빈번하게 발생한다. 이와 같은 경우를 통제하기 위한 과정을 예외 상황의 전파(Exception Propagation)라 한다.

본 절에서는 이러한 예외 상황의 전파에 대해 케이스 별로 예를 들어 다음과 같이 설명한다.

선언부에서 예외 상황이 발생한 경우

다음은 선언부에서 예외 상황이 발생한 예이다.

```
DECLARE
    tmp_number NUMBER(10) := 'STRING';
BEGIN
    ...
EXCEPTION
    /* VALUE_ERROR나 OTHERS를 포함하고 있으나 에러가 선언부에서 발생했으므로
       예외 처리 루틴은 실행되지 않는다. */
WHEN VALUE_ERROR THEN
    ...
WHEN OTHERS THEN
    ...
    /* 외부 블록이 없으므로 블록은 예외 상황을 처리하지 못한다.
       프로그램이 비정상적으로 종료된다. */
END;
```

선언부에서 예외 상황이 발생한 경우 발생 즉시 외부 블록으로 전파된다. 예외 상황이 발생한 블록에 예외 처리 루틴이 있어도 이 처리 루틴은 무시된다.

OTHERS 문을 이용한 예외 상황 처리

다음은 외부 블록의 예외 처리 루틴 중에서 OTHERS 문에 의해 내부 블록의 VALUE_ERROR가 처리되는 경우의 예이다. 이때 프로그램은 정상적으로 종료된다.

```
BEGIN
    DECLARE
        tmp_number VARCHAR2(10) := 'STRING';
    BEGIN
        ...
    EXCEPTION
    WHEN VALUE_ERROR THEN
        ...
    WHEN OTHERS THEN
        ...
    END;

EXCEPTION
/* 내부 블록에서 전파된 예외 상황이 OTHERS 문에 의해 처리된다. */
WHEN OTHERS THEN
    ...
    /* 프로그램은 정상적으로 종료된다. */
END;
```

RAISE 문을 이용한 예외 상황 처리

예외 처리 루틴 내에서도 사용자의 필요에 의해 RAISE 문을 사용하여 명시적으로 예외를 발생시키거나, 사용자가 의도하지 않은 런타임 에러가 발생할 수 있다. 이렇게 발생된 예외 상황은 선언부에서와 같이 외부 블록으로 바로 전파된다.

다음은 RAISE 문을 이용하여 예외 상황을 처리하는 예이다.

```
DECLARE
    a EXCEPTION;
    b EXCEPTION;
BEGIN
    RAISE a;
EXCEPTION
WHEN a THEN
    /* 예외 상황 A를 처리하는 중에 예외 상황 B가 발생되었으므로
       예외 상황 B가 다시 외부 블록으로 전파된다. */
    RAISE b;
WHEN b THEN
    ...
/* 외부 블록이 없으므로, 예외 상황을 처리하지 못한 채로
   프로그램이 비정상적으로 종료된다. */
END;
```

예외 상황은 위 예에서처럼 한 번에 하나씩 발생된다. 물론 이렇게 발생된 하나의 예외 상황이 처리되자마자 또 다른 예외 상황이 발생될 수 있지만, 동시에 두 개 이상의 예외 상황이 발생될 수는 없다.

다음은 RAISE 문에 예외 상황을 명시하지 않아, 예외 상황을 처리할 때 불필요한 실행이 발생하지 않도록 작성한 프로그램의 예이다.

```
DECLARE
    a EXCEPTION;
BEGIN
    RAISE a;
EXCEPTION
WHEN a THEN
    send_log('exception error');
    RAISE;
    /* RAISE 문에 의해 예외 상황이 발생되었으므로 send_log_date 함수는
       실행되지 않고 예외 처리 루틴이 강제로 종료된다.
       이때 a가 그대로 외부 블록으로 전파된다. */
    send_log_date('2009-04-30');
WHEN OTHERS THEN
    ...
END;
```

예외 처리 루틴의 존재 여부

실행부에서 예외가 발생하였으나 예외 처리 루틴이 없는 경우 또는 예외 처리 루틴은 있지만 발생된 예외 상황을 처리할 수 없는 경우에 예외 상황은 외부 블록으로 전파된다.

```
DECLARE
    a EXCEPTION;
BEGIN
    BEGIN
        RAISE a;    /* 예외 상황 a 발생 */
    EXCEPTION       /* 발생한 예외 상황 a에 대한 예외 처리 루틴이 있다. */
    WHEN a THEN
        ...
                    /* 예외 상황 a를 처리한 후 외부 블록으로 제어가 이동된다. */
    WHEN OTHERS THEN
        ...
    END;
                    /* 내부 블록이 정상적으로 종료되고 나서
                       외부 블록의 나머지 부분을 계속 수행한다 */
END;
```

위의 예에서는 내부 블록에서 발생한 예외 상황 a를 처리할 수 있는 예외 처리 루틴이 존재한다. 따라서 정상적으로 내부 블록을 실행하고 제어를 다시 외부 블록으로 넘김으로써 나머지 부분을 계속 수행하게 된다.

반면에 다음의 예는 다르다.

```
DECLARE
    a EXCEPTION;
    b EXCEPTION;
BEGIN
    BEGIN
        RAISE b;    /* 예외 상황 b 발생 */
    EXCEPTION       /* 발생한 예외 상황 b에 대한 예외 처리 루틴이 없다. */
    WHEN a THEN
        ...
    WHEN OTHERS THEN
        ...
    END;
                    /* 외부 블록으로 예외 상황이 전파된다. */
EXCEPTION
WHEN b THEN         /* 예외 상황 b를 처리한 후 외부 블록은 정상적으로 종료된다. */
    ...
END;
```

이 예에서는 내부 블록에 예외 상황 b를 처리하기 위한 예외 처리 루틴이 없으므로 예외 상황은 외부 블록으로 전파된다. 이렇게 전파된 예외 상황은 외부 블록에서 정상적으로 종료된다.

추가로 외부 블록에서도 발생한 예외 상황을 처리하기 위한 에러 처리 루틴이 없을 수 있다. 이러한 경우 예외 상황은 다시 상위 블록으로 계속 전파된다. 이때 상위 블록의 작업이 끝나면 예외 상황은 처리되지 못하고 프로그램은 비정상적으로 종료된다.

8.3.3. 에러 정보

에러에 대한 정보를 얻기 위해서는 Tibero가 제공하는 함수 중에 SQLCODE와 SQLERRM를 사용해야 한다. SQLCODE 함수는 에러 코드를 반환하고, SQLERRM 함수는 에러 메시지를 반환한다. 이 두 함수는 SQL 문장 내에서 직접 사용할 수 없으며, 변수에 할당하여 사용한다.

다음은 SQLCODE와 SQLERRM 함수를 통해 얻을 수 있는 결과이다.

예외 상황	**SQLCODE**	**SQLERRM(SQLCODE)**
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

SQLERRM을 parameter없이 사용하면 에러 스택의 탑에 있는 에러의 메시지를 반환하고, SQLERRM(SQLCODE)의 경우 에러 스택과는 관계없이 SQL 코드에 해당하는 에러의 메시지를 반환한다.

다음은 SQLCODE와 SQLERRM 함수의 예이다.

```
BEGIN
    copy_dept_info(20090430, 'Tibero');
EXCEPTION
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLCODE);
    /* 에러 코드를 반환한다. */

    DBMS_OUTPUT.PUT_LINE(SQLERRM);
    /* 에러 스택의 탑에 있는 에러의 메시지를 반환한다. */

    DBMS_OUTPUT.PUT_LINE(SQLERRM(SQLCODE));
    /* 에러 코드에 대한 메시지를 반환한다. */
END;
```

제9장 파이프라인드 테이블 함수

본 장에서는 파이프라인드 방식으로 데이터를 처리할 수 있는 파이프라인드 테이블 함수를 설명한다.

9.1. 개요

파이프라인드 테이블 함수(Pipelined Table Functions)는 tbPSM의 서브 프로그램을 관계형 테이블과 같이 로우의 형태로 결과를 반환하는 함수이다. 이 함수는 SQL 문장의 FROM 절에 사용할 수 있으며, IN 모드로 파라미터를 지정하고 결과 집합으로 반환하도록 구성할 수 있다.

파이프라인드 테이블 함수를 사용하면 다음과 같은 장점이 있다.

- 응답 시간(Response Time)

  결과 집합이 모두 생성될 때까지 기다리지 않고 완료된 부분 결과 집합을 순차적으로 처리할 수 있으므로, 연산의 응답 시간이 빠르다.

- 스트리밍(Streaming)

  하나의 연산을 종료하지 않고 일련의 연산을 순차적으로 반복하여 처리할 수 있다.

- 파이프라이닝(Pipelining)

  여러 가지 연산을 연속해서 수행할 수 있다.

- 유연성(Flexibility)

  SQL 문장으로 표현하기 어려운 과정을 필터를 적용한 것처럼 유연하게 처리할 수 있다.

9.2. 함수

파이프라인드 테이블로 만들 수 있는 것은 단독 함수(Standalone Function)뿐이다. 따라서 로컬 함수(Local Function), 패키지의 멤버 함수 등은 파이프라인드 테이블 함수로 만들 수 없다.

파이프라인드 테이블 함수의 세부 내용은 다음과 같다.

- 프로토타입

```
CREATE TABLE tblname(m1 NUMBER, m2 NUMBER)
/

CREATE OR REPLACE PACKAGE pkg IS
TYPE tbl_rec IS TABLE OF tblname%ROWTYPE;
END;
```

```
/

CREATE OR REPLACE FUNCTION PTF(limit IN NUMBER)
RETURN pkg.tbl_rec PIPELINED
AS
  x tblname%ROWTYPE;
BEGIN
  FOR i IN 1 .. limit LOOP
  x.m1 := i;
  x.m2 := i+1;
  PIPE ROW(X);
  END LOOP;
END;
/
select * from TABLE(ptf(1000)) where m1 = 500
/
```

- 파라미터

파라미터	설명
IN	IN 모드만 사용할 수 있다. 따라서 OUT, INOUT 모드는 사용할 수 없다.

- 반환값

반환값	설명
배열 또는 테이블	배열이나 테이블로 반환해야 하며, 인덱스 테이블로는 반환할 수 없다. – 컬렉션의 원소는 데이터베이스 타입이거나, 데이터베이스 타입으로 구성된 레코드 타입이어야 한다. – 일반 함수와는 다르게 RETURN 문을 사용할 수 없다. 하지만 tbPSM 프로그램의 흐름을 위해 값이 없는 RETURN 문은 사용할 수 있다.

- 예제

  파이프라인드 테이블 함수는 **PIPELINED**라는 키워드를 이용하여 지정할 수 있다. 예를 들면 다음과 같다.

```
CREATE PACKAGE PKG IS
  TYPE rec IS RECORD(m1 NUMBER, m2 VARCHAR2(100));
  TYPE tbl_rec IS TABLE OF rec;
END;
/
CREATE FUNCTION pipelined_table_func
RETURN PKG.TBL_REC PIPELINED
AS
x PKG.REC;
```

```
BEGIN
  FOR i IN 1 .. 1000 LOOP
    x.m1 := i;
    x.m2 := 'abc';

    PIPE ROW(x);
  END LOOP;
END;
/

select * from TABLE(pipelined_table_func()) where m1 = 500
/
```

제10장 오브젝트 타입

본 장에서는 추상 데이터 타입인 오브젝트 타입에 대해서 설명한다.

10.1. 개요

오브젝트 타입(Object Type)은 데이터 타입의 데이터 구성 요소와 서브 프로그램을 캡슐화(Encapsulation)한 추상 데이터 타입(Abstract Data Type)이다.

데이터 구성요소는 Attribute라고 부르며, 서브 프로그램은 메소드(Method)라고 부른다. 사용자는 추상 데이터 타입을 이용하여 애플리케이션에 관련된 데이터와 서브 프로그램을 한 곳으로 모을 수 있어 애플리케이션 작성이 쉬워지고 가독성이 높아진다.

tbPSM의 오브젝트 타입이 클래스나 패키지와의 다른점은 다음과 같다.

- 클래스와 다른 점은 상속성과 다형성이 제외된 순수한 추상 데이터 타입이라는 것이다. (상속성과 다형성은 추후 추가 예정)
- 패키지와 다른 점은 타입 실체화이다.
  - 오브젝트 타입의 변수는 컬렉션 타입 변수처럼 각각 다른 메모리 공간을 사용한다.
  - 패키지 자체가 세션마다 유일한 데이터 셋를 가지는 반면 오브젝트 타입 자체로는 데이터 셋를 가지지 않으며 오브젝트 타입 실체화를 통해 각각의 인스턴스마다 데이터 셋를 가진다.
  - 오브젝트 타입의 실체화 방법은 컬렉션 타입과 동일하게 생성자를 호출하는 것이며, 생성자가 삭제되는 것은 변수의 생명 주기와 같다.
  - 변수의 Block Scope가 끝났을 경우 오브젝트 인스턴스에 대한 메모리가 해제된다. 단, 패키지 멤버 변수인 경우에는 패키지 인스턴스가 무효화 되는 경우에 삭제된다.

참고

tbPSM의 오브젝트 타입은 tbPSM 내에서 사용되는 기능만 구현되었으므로 SQL이나 테이블을 생성하는 곳에서는 사용이 불가능하다. (기능 추가 예정)

다음은 오브젝트 타입을 이용한 Number Stack의 구현 예로 Stack의 데이터 배열과 Top 포인터, 그리고 푸쉬(Push), 팝(Pop), 탑(Top) 동작을 한 곳으로 모은 것을 확인할 수 있다.

```
DROP TYPE number_array FORCE;

-- Stack 내부에서 사용하는 데이터를 위한 최대 크기의 한계가 없는 배열 데이터 타입
```

```
CREATE OR REPLACE TYPE number_array AS TABLE OF NUMBER;
/
show err

DROP TYPE type_nstack force;

-- number stack의 추상 데이터 타입
CREATE OR REPLACE TYPE type_nstack AS OBJECT(
    maxlen pls_integer,  -- Stack의 최대 크기
    topptr pls_integer,  -- Stack의 TOP
    array  number_array, -- Stack 내부 데이터

    MEMBER PROCEDURE pop (self in out nocopy type_nstack),
    MEMBER FUNCTION top (self in out nocopy type_nstack) return NUMBER,
    MEMBER PROCEDURE push (self in out nocopy type_nstack, input NUMBER),
    CONSTRUCTOR FUNCTION type_nstack(maxlen pls_integer) RETURN SELF AS RESULT
);
/
show err

CREATE OR REPLACE TYPE BODY type_nstack
AS
    MEMBER PROCEDURE pop (self in out nocopy type_nstack)
    AS
    BEGIN
        IF topptr <= 0 THEN
            raise_application_error(-20000, 'stack is empty', true);
        ELSE
            array(topptr) := NULL;
            topptr := topptr - 1;
        END IF;
    END;

    MEMBER FUNCTION top (self in out nocopy type_nstack) return NUMBER
    AS
    BEGIN
        IF topptr <= 0 THEN
            return NULL;
        ELSE
            return array(topptr);
        END IF;
        return NULL;
    END;

    MEMBER PROCEDURE push (self in out nocopy type_nstack, input NUMBER)
    AS
    BEGIN
```

```
        IF topptr = maxlen THEN
            raise_application_error(-20001, 'stack is full', true);
        ELSE
            array(self.topptr + 1) := input;
            topptr := topptr + 1;
        END IF;
    END;

    CONSTRUCTOR FUNCTION type_nstack(maxlen pls_integer) RETURN SELF AS RESULT
    AS
    BEGIN
        array := number_array();
        array.extend(maxlen);
        topptr := 0;
        RETURN;
    END;
END;
/
show err

set serveroutput on
-- 간단한, Stack Push/Pop 동작 테스트 애플리케이션
declare
    s1 type_nstack;
begin

    s1 := type_nstack(4); -- 최대 크기 4짜리 Stack의 생성
    s1.push(10);    -- 10을 Push
    s1.push(20);    -- 20을 Push
    s1.push(30);    -- 30을 Push
    s1.push(40);    -- 40을 Push

    dbms_output.put_line (s1.top);     -- Stack의 Top을 조회
    s1.pop;    -- stack pop

    dbms_output.put_line (s1.top);    -- Stack의 Top을 조회
    dbms_output.put_line (s1.top);    -- Stack의 Top을 조회

    s1.pop;    -- stack pop
    dbms_output.put_line (s1.top);    -- Stack의 Top을 조회

    s1.pop;    -- stack pop
    s1.pop;    -- stack pop
    s1.pop;    -- stack pop : Stack이 비어 있으므로 에러
end;
/
```

10.2. 생성자

생성자는 디폴트 생성자와 사용자 정의 생성자가 있다.

10.2.1. 디폴트 생성자

사용자 정의 생성자가 없는 경우 디폴트 생성자가 호출되며, 디폴트 생성자의 함수 시그네쳐는 Attribute의 순서와 동일하다. 호출자가 argument로 다른 타입을 가지고 생성자를 호출하는 경우 tbPSM implicit type conversion rule을 따른다. 만약 오브젝트 변수가 NULL일 경우 생성자 함수 호출 중에 Exception이 발생한다(오브젝트 인스턴스가 NULL일 경우 Attribute는 항상 NULL이다).

다음은 디폴트 생성자를 호출하는 예이다.

```
create or replace type default_constructor as object (
   a1 number,
   a2 varchar2(1024)
);
/

declare
    var default_constructor;
begin
    var := default_constructor(10, 'default');

    dbms_output.put_line (var.a1);
    dbms_output.put_line (var.a2);
end;
/
```

10.2.2. 사용자 정의 생성자

사용자 정의 생성자가 디폴트 생성자와 시그네쳐가 다르면 함수 오버로딩 규칙에 따라 타입 변환이 가장 적고 정보 손실이 가장 적은 쪽을 우선적으로 선택한다. 만약 선택하지 못하는 경우 컴파일 에러가 발생한다. 사용자 정의 생성자를 생성할 때 오브젝트 변수는 기본으로 NULL 값을 가진다. 사용자 정의 생성자에 대한 자세한 설명은 “10.4. 메소드”의 “10.4.1. 생성자 메소드”를 참고한다.

다음은 사용자 정의 생성자를 호출하는 예이다.

```
create or replace type constructor_body_test as object (
    a1 number,
    constructor function constructor_body_test (param1 number)
    return self as result,
    constructor function constructor_body_test (param1 number, param2 number)
```

```
    return self as result
);
/

create or replace type body  constructor_body_test
as
    constructor function constructor_body_test (param1 number)
    return self as result
    as
    begin
        a1 := param1 + 1000;
        return;
    end;

    constructor function constructor_body_test (param1 number, param2 number)
    return self as result
    as
    begin
        a1 := param1 + param2 + 1000;
        return;
    end;
end;
/

declare
    var2 constructor_body_test ;
begin
    var2 := constructor_body_test (5, 60);
    dbms_output.put_line( var2.a1 );
end;
/
```

10.3. Attribute

Attribute는 오브젝트 타입의 구성요소인 데이터들이다.

- 원시 데이터 타입(Primitive Data Type)
- 오브젝트 타입(Object Type)
- 컬렉션 타입(Collection Type) 중 인덱스 바이 테이블 타입(Index by table Type)을 제외한 배열 타입(Varray Type)
- 테이블 타입(Table Type)

레코드 타입(Record Type)은 Attribute 타입으로 사용될 수 없다. 레코드 타입과 인덱스 바이 테이블 타입은 tbPSM 로컬 타입(tbPSM BLOCK 안에서 선언 가능)이기 때문이다. 즉, 원시 데이터 타입과 create type

DDL 문으로 선언하는 타입은 Attribute 타입으로 사용 가능하다. Attribute의 이름은 중복을 허용하지 않으며, 메소드 이름과 중복되면 안 된다.

다음은 컬렉션 타입과 오브젝트 타입을 혼합 사용한 예이다.

```
-- 국가 타입
create type o_nation  as object (
    name       varchar2(16),
    continent  varchar2(16),
    climate    varchar2(16)
);
/
-- 주소 타입
create type o_address as object (
    nation  o_nation,
    city    varchar2(16),
    street  varchar2(16),
    zipcode varchar2(16)
);
/

-- 사람 타입
create type o_person as object (
    name                varchar2(16),
    height              number,
    weight              number,
    living_place        o_address,
    working_place       o_address,
    gender              char(1),
    ex_military_service char(1)
);
/

-- 이직 프러시저
create procedure  change_working_place (p in out o_person, a in o_address)
as
begin
    p.working_place := a;
end;
/
-- 직장을 출력하는 프러시저
create procedure  print_working_place (p in o_person)
as
begin
   dbms_output.put_line('>' || p.working_place.nation.name);
   dbms_output.put_line('>' || p.working_place.nation.continent);
   dbms_output.put_line('>' || p.working_place.nation.climate);
```

```
   dbms_output.put_line('>' || p.working_place.city);
   dbms_output.put_line('>' || p.working_place.street);
   dbms_output.put_line('>' || p.working_place.zipcode);
end;
/

-- 그 나라의 수도를 기본 주소로 주소를 생성하는 함수
create function default_address (nation varchar2) return o_address
as
begin
    if nation = 'Korea' then
      return o_address(o_nation(nation, null, null), 'Seoul', null, null);
    elsif nation = 'USA' then
      return o_address(o_nation(nation, null, null), 'Washington', null, null);
    elsif nation = 'France' then
      return o_address(o_nation(nation, null, null), 'Paris', null, null);
    else
      return o_address(o_nation(nation, null, null), null, null, null);
    end if;
end;
/
-- 미국 피닉스에서 태어난 "인피니데이터"의 직장위치를
-- 크로아티아의 수도에서 한국의 수도로 변경하는 애플리케이션
set serveroutput on

declare
    my_person o_person;
begin
    my_person := o_person('Infinidata', '180', '72',
     o_address(o_nation('USA', null, null), 'Pheonix', null, null),
     o_address(o_nation('Croatia', null, null), 'Zagreb', null, null),'M', 'Y');
    print_working_place (my_person);
    change_working_place (my_person, default_address('Korea'));
    print_working_place (my_person);
end;
/
```

10.4. 메소드

메소드는 오브젝트 타입에 필요한 서브 프로그램이다.

- 생성자 메소드
- 정적 메소드
- ORDER 메소드

- MAP 메소드
- 멤버 메소드

정적 메소드, 멤버 메소드는 함수나 프러시저가 가능하고, ORDER 메소드와 MAP 메소드는 함수만 가능하다. 오프젝트 타입은 ORDER 메소드나 MAP 메소드 둘 중에 하나만 가질 수 있다.

10.4.1. 생성자 메소드

사용자 정의 생성자를 생성할 때 리턴 문에 리턴 값을 써주지 않더라도 암묵적으로 새 인스턴스를 반환한다. 사용자 정의 생성자의 첫 번째 파라미터로 SELF라는 이름의 자기 자신의 타입을 가지는 파라미터를 명시 가능하지만 지정하지 않는 것과 동일하다. 첫 번째 파라미터를 SELF로 지정하지 않더라도 암묵적으로 SELF가 있는 것과 같기 때문이다(암묵적인 SELF는 IN/OUT NOCOPY 파라미터이다).

10.4.2. 정적 메소드

정적 메소드는 오브젝트 INSTANTIATION을 하지 않고도 호출이 가능한 메소드로 다음과 같은 형태로 호출한다.

```
"오브젝트 타입 이름"."정적 메소드 이름" (아규먼트들)
```

정적 메소드는 타입 이름을 가지고 호출하기 때문에 생성자의 호출과 상관없이 불리는 경우 인스턴스에 대한 접근이 불가능하다. 즉, 정적 메소드는 인스턴스 데이터 접근과 전혀 상관없는 경우에만 사용되어야 한다.

다음은 정적 메소드의 사용 예이다.

```
-- 스머프 타입
drop type smuff force;

create type smuff as object(
    name   varchar(1024),   -- 스머프 이름
    height number,          -- 스머프 키
    static function eat(blueberry in number, sizeofmeal in number)
    return number
);
/

create type body smuff
as
    -- 스머프가 밥을 먹는다고 키가 크지 않는다.
    -- 이름이 비뀌지도 않는다.
    static function eat(blueberry in number, sizeofmeal in number)
    return number
    as
```

```
    begin
        return blueberry - sizeofmeal;
    end;
end;
/
show err

set serveroutput on
declare
    fafa_smuff smuff;
    blueberry_amount number := 10000;
begin
     -- 8cm 키의 파파 스머프 생성
    fafa_smuff := smuff('Fafa', 8);

    -- 키와 상관없이 모두 하루에 100개의 블루베리를 먹는다.
    -- 스머프 식사와 스머프 데이터와 관련이 없고,
    -- 블루베리의 양만 줄어들고 있으므로,
    -- static method 사용이 적절하다.
    blueberry_amount := smuff.eat(blueberry_amount, 100);

    dbms_output.put_line ('# of blueberry:' || blueberry_amount);
end;
/
```

10.4.3. ORDER 메소드

ORDER 메소드는 두 개의 오브젝트 인스턴스를 비교하기 위해 사용되는 메소드이다. 매개변수로 SELF를 제외하고 유일한 자기 자신의 타입을 가지는 파라미터 하나가 있어야 한다. 또한 리턴 값은 숫자 타입(Number, Bin_Integer, Binary_double, Binary_float와 그에 따른 Sub Type)만을 가져야 한다.

만약 이항 비교 연산자가 표현식에 존재하는 경우 **a "비교연산자" b**는 다음과 같이 변환되어 오브젝트 인스턴스의 비교가 가능하도록 한다. 이러한 기능은 tbPSM 표현식에서만 가능하고 SQL에서는 사용 불가능하다.

```
a.order_method(b) "비교연산자" 0
```

MAP 메소드와 ORDER 메소드의 파라미터와 반환값 비교는 다음과 같다.

메소드	파라미터	반환값
MAP	SELF를 제외하고 파라미터 사용이 불가능하다.	Primitive Data Type
ORDER	SELF를 제외하고 자기 자신 타입을 가지는 1개의 IN 파라미터 사용이 가능하다.	숫자 타입 – Number – Bin_Integer

메소드	파라미터	반환값
		– Binary_double – Binary_float – 그에 따른 Sub Type

사용자는 두 개의 파라미터를 비교하여 다음과 같이 메소드를 작성해야 한다.

- SELF가 더 큰 경우: 음수를 리턴한다.
- 두개가 같은 경우: 0을 리턴한다.
- SELF가 작은 경우: 양수를 리턴한다.

다음은 Stack 타입의 ORDER 메소드 작성을 이용하여 두 개의 Stack을 비교하는 예이다(Stack의 element 개수가 많은것이 더 값이 크다고 가정).

```
DROP TYPE number_array force;

CREATE OR REPLACE TYPE number_array AS TABLE OF NUMBER;
/
show err

DROP TYPE type_nstack force;

CREATE OR REPLACE TYPE type_nstack AS OBJECT(
    maxlen pls_integer,  -- Stack의 최대 크기
    topptr pls_integer,  -- Stack의 TOP
    array  number_array, -- Stack 내부 데이터

    MEMBER PROCEDURE pop (self in out nocopy type_nstack),
    MEMBER FUNCTION top (self in out nocopy type_nstack) return NUMBER,
    MEMBER PROCEDURE push (self in out nocopy type_nstack, input NUMBER),

    CONSTRUCTOR FUNCTION type_nstack(maxlen pls_integer) RETURN SELF AS RESULT,
    ORDER MEMBER FUNCTION compare(v in type_nstack) RETURN NUMBER
);
/
show err

CREATE OR REPLACE TYPE BODY type_nstack
AS
    MEMBER PROCEDURE pop (self in out nocopy type_nstack)
    AS
    BEGIN
```

```
    IF topptr <= 0 THEN
        raise_application_error(-20000, 'stack is empty', true);
    ELSE
        array(topptr) := NULL;
        topptr := topptr - 1;
    END IF;
END;

MEMBER FUNCTION top (self in out nocopy type_nstack) return NUMBER
AS
BEGIN
    IF topptr <= 0 THEN
        return NULL;
    ELSE
        return array(topptr);
    END IF;
    return NULL;
END;

MEMBER PROCEDURE push (self in out nocopy type_nstack, input NUMBER)
AS
BEGIN
    IF topptr = maxlen THEN
        raise_application_error(-20001, 'stack is full', true);
    ELSE
        array(self.topptr + 1) := input;
        topptr := topptr + 1;
    END IF;
END;

CONSTRUCTOR FUNCTION type_nstack(maxlen pls_integer) RETURN SELF AS RESULT
AS
BEGIN
    array := number_array();
    array.extend(maxlen);
    topptr := 0;

    RETURN;
END;

ORDER MEMBER FUNCTION compare(v in type_nstack) RETURN NUMBER
AS
BEGIN
    IF      self.array.count = v.array.count THEN
        RETURN 0;
    ELSIF   self.array.count < v.array.count THEN
        RETURN -1;
```

```
        ELSE
            RETURN 1;
        END IF;
    END;
END;
/

set serveroutput on

// 두 개의 Stack을 비교하는 애플리케이션
declare
    s1 type_nstack;
    s2 type_nstack;
begin
    s1 := type_nstack(8);
    s2 := type_nstack(4);
    if s2 < s1 then
        dbms_output.put_line('s2 is smaller then s1.');
    end if;

    s1 := type_nstack(4);
    s2 := type_nstack(8);
    if s2 > s1 then
        dbms_output.put_line('s2 is bigger then s1.');
    end if;

    s1 := type_nstack(4);
    s2 := type_nstack(4);
    if s2 = s1 then
        dbms_output.put_line('s2 is equal to s1.');
    end if;

end;
/
```

10.4.4. MAP 메소드

MAP 메소드는 인스턴스에서 primitive 타입의 scalar 값을 리턴하는 메소드이다. 함수는 SELF 외에 아무런 파라미터가 있어서는 안되며 primitive 타입을 리턴해야 한다. 만약 이항 연산자가 있는 경우 **a < b**는 다음과 같이 변환되어 오브젝트 인스턴스의 비교가 가능하도록 한다.

```
a.map_mehod() < b.map_method()
```

다음은 MAP 메소드의 사용 예이다.

```
CREATE OR REPLACE TYPE some_type AS OBJECT (
    key_value   number,
    divider     number,

    MAP MEMBER FUNCTION get_hashval RETURN NUMBER
    );
/

CREATE OR REPLACE TYPE BODY some_type
AS
    MAP MEMBER FUNCTION get_hashval RETURN NUMBER
    AS
    BEGIN
        return mod (key_value, divider);
    END;
END;
/



DECLARE
    var1 some_type;
    var2 some_type;
BEGIN
    var1 := some_type (15, 10);
    var2 := some_type (16, 10);
    dbms_output.put_line ( var1.get_hashval );
    dbms_output.put_line ( var2.get_hashval );

    IF var1 < var2 THEN
        -- var1 < var2에서 암묵적으로 MAP 메소드가 호출된다.
        -- 즉, var1.get_hashval() < var2.get_hashval()로 변환된다.
        dbms_output.put_line ('var2 is bigger then var1.');
    ELSE
        dbms_output.put_line ('var1 is bigger then var2.');
    END IF;
END;
/
```

10.4.5. 멤버 메소드

멤버 메소드는 다음과 같은 형태로 호출한다.

```
"인스턴스 이름"."메소드 이름"
```

멤버 메소드는 첫 번째 파라미터가 암묵적인 SELF 파라미터로 자기 자신의 타입을 가지는 파라미터이다. 기본 속성은 IN 파라미터이며 사용자가 IN/OUT 형태로 변경이 가능하다. IN으로 사용하면 Attribute 값을

변경할 수 없고 리턴을 통해 변경이 가능하므로 원소적인 동작을 취할수 있다. SELF 파라미터는 인스턴스 데이터에 접근할 수 있도록 해주며 "SELF"."Attribute 이름"으로 사용하거나 "Attribute 이름"으로 직접 사용이 가능하다.

멤버 메소드는 OVERLOADING이 가능하지만 MAP, ORDER 메소드는 OVERLOADING이 불가능하다. 멤버 메소드는 LOCAL METHOD(BODY에서만 사용 가능한 메소드)를 정의할 수 없다. 항상 TYPE OBJECT에 기술된 메소드가 정확히 BODY에도 존재해야 하며, BODY에 존재하는 메소드는 반드시 TYPE OBJECT에 기술되야 한다.

다음은 Number Stack의 사용 예이다.

```
declare
  s1 type_nstack;
begin
  s1 := type_nstack (16);

  s1.push (1);                      -- 멤버 메소드 호출(procedure)
  dbms_output.put_line (s1.top())  -- 멤버 베소드 호출(function)
  s1.pop;                           -- 멤버 메소드 호출(procedure)
end;
/
```

10.5. 오브젝트 관련 정적 뷰

다음은 **오브젝트 관련 정적 뷰**이다.

- ALL_TYPES, DBA_TYPES, USER_TYPES
- ALL_COLL_TYPES, DBA_COLL_TYPES, USER_COLL_TYPES
- ALL_TYPE_ATTRS, DBA_TYPE_ATTRS, USER_TYPE_ATTRS
- ALL_TYPE_METHODS, DBA_TYPE_METHODS, USER_TYPE_METHODS
- ALL_METHOD_PARAMS, DBA_METHOD_PARAMS, USER_METHOD_PARAMS
- ALL_METHOD_RESULTS, DBA_METHOD_RESULTS, USER_METHOD_RESULTS

참고

해당 뷰에 대한 자세한 내용은 "Tibero 참조 안내서"의 "제3장 Static View"를 참고한다.

제11장 사용자 정의 AGGREGATION 함수

본 장에서는 사용자 정의 AGGREGATION 함수에 대하여 대해서 설명한다.

11.1. 개요

AGGREGATION 함수는 미리 정해진 AGGREGATION 함수와 사용자 정의 AGGREGATION 함수로 나뉘어진다. 미리 정해진 AGGREGATION 함수는 사전에 정의하여 제공되는 함수로 자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

사용자 정의 AGGREGATION 함수는 미리 정해진 AGGREGATION 동작 외에 사용자가 정의한 임의의 동작의 AGGREGATION을 수행한다. 임의의 동작을 수행하기 위해 쿼리 수행기는 AGGREGATION의 여러 단계를 정의하고, 각 단계에서 AGGREGATION 동작을 사용자가 정의한 메소드로 콜백(CALLBACK)함으로써 AGGREGATION 동작을 수행할수 있는 확장성을 제공한다.

11.2. 함수 DDL

사용자 정의 AGGREGATION 함수는 바디가 없는 함수로 선언된다.

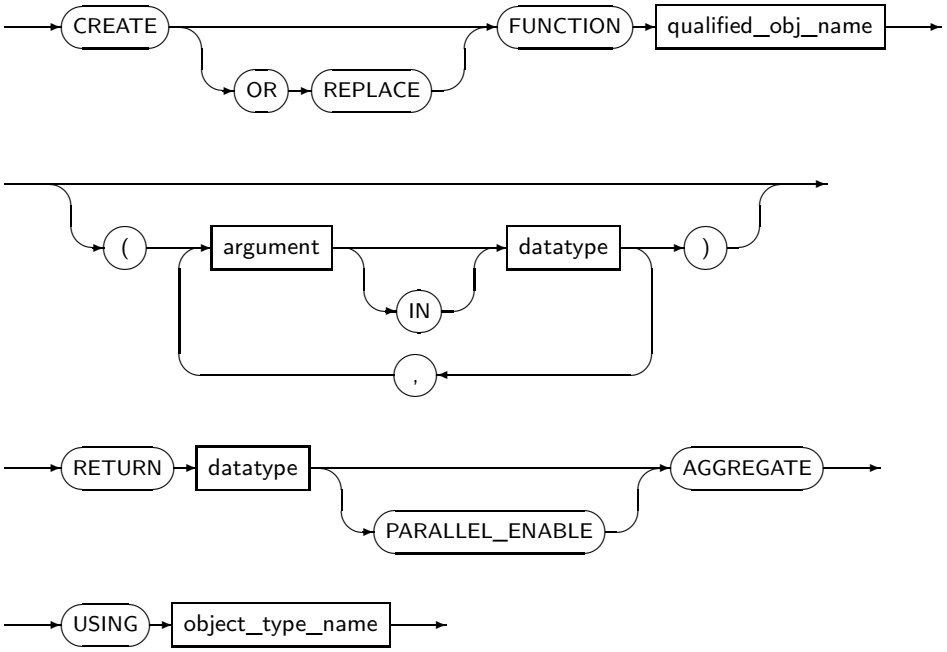
바디가 없는 이유는 쿼리 수행기의 동작이 다양한 단계를 가지기 때문에 하나의 함수로 정의할 수 없다. 대신 쿼리 수행기의 여러 단계에서 사용 될 콜백 메소드들을 포함하는 오브젝트 타입으로 정의한다.

사용자 정의 AGGREGATION 동작이 병렬로 수행 가능한지 여부 또한 사용자가 지정 가능하다. 사용자가 병렬 수행 가능 여부를 지정한 경우 병렬 수행 가능하며 그에 대한 동작이나 알고리즘은 병렬 수행에 맞게 올바로 기술되어야 한다.

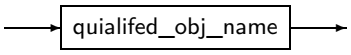
기술하지 않는 경우 병렬로 수행될 수 없으므로, 옵티마이져의 실행 계획 생성 과정에서 효율적인 실행 계획 생성에 제약을 줄 수 있다.

- 문법

user_defined_aggregation_function



object_type_name



- 구성요소

구성요소	설명
qualified_obj_name	"스키마명.함수이름"으로 쓸 수 있으며, 스키마명이 생략 가능하다. 스키마명이 기입되면 함수가 속해있는 스키마를 명시한다. 생략하면 현재 사용자의 스키마로 인식된다.
argument	함수의 파라미터이다. 1개 이상의 파리미터가 주어져야 한다.
IN	함수의 파라미터의 전달 방향에 따른 구분이다. IN 파라미터는 외부로부터 값을 입력 받는다. 기본값은 IN 파라미터이다. AGGREGATION 함수에서는 IN 파라미터만을 지원한다. IN OUT, OUT, NOCOPY 옵션은 허용하지 않는다.
datatype	함수의 파라미터의 데이터 타입이다. 파리미터 타입은 스칼라 타입만 지원한다.
PARALLEL_ENABLE	AGGREGATION 동작이 병렬로 동작 가능한지를 나타낸다. 기술하지 않는 경우 병렬로 수행이 불가능하다.
AGGREGATE	사용자 지정 AGGREGATION 함수 여부를 나타낸다.

구성요소	설명
object_type_name	객체 타입의 이름이다. 객체 타입은 쿼리 수행기의 AGGREGATION 동작을 포함하는 TUDIAGGREGATEINITIALIZE, TUDIAGGREGATEITERATE, TUDIAGGREGATETERMINATE, TUDIAGGREGATEMERGE(선택가능)이 포함되어야 한다. "객체 타입의 이름" 또는 "스키마명.객체 타입"의 이름으로 쓸 수 있다.

11.3. 객체 타입

사용자가 정의한 AGGREGATION을 표현하기 위해서 하나의 함수로 표현할수 있으면 좋겠지만, 쿼리 수행기의 동작은 이와 같지 않다.

쿼리 수행기는 AGGREGATION 동작 중 다양한 단계를 가지며, 각각의 동작은 다른 연산이 요구된다. 또한 사용자 정의 AGGREGATION은 중간 연산 도중 사용자가 정의한 타입의 데이터를 유지할 수 있어야 한다. 이러한 것을 표현하기에 적합한 것은, 오브젝트 타입이다.

오브젝트 타입은 임의의 타입을 애트리뷰트로 가지며, 임의의 동작들을 메소드로 정의하여 하나로 묶을 수 (Encpsulation) 있기 때문이다. 이에 따라, 사용자 정의 AGGREGATION 동작은 그에 필요한 임의의 애트리뷰트, 단계별 AGGREGATION 동작은 메소드로 정의한다.

애트리뷰트는 일반 오브젝트 타입의 제약에 따른다. 그렇기 때문에 사용자가 객체 타입의 사용자 정의 타입의 표현 안에서 어떠한 데이터도 AGGREGATION 동작에 포함시킬 수 있다. 메소드는 PSM의 언어의 제약 범위 안에서 자유롭게 표현 가능하다. 단, 해당 오브젝트의 생성 및 메소드 호출은 사용자 의해 직접 컨트롤되지 않는다.

쿼리 수행기는 오브젝트 인스턴스를 생성하고 해당 인스턴스에 값을 누적시킴으로 인해 AGGREGATION 동작을 수행하며 각 메소드는 쿼리 수행도중 필요한 단계에 맞게 자동으로 콜백(CALLBACK)되는 형태로 사용된다. 값을 생성하는 AGGREGATION 함수 동작을 수행한다.

11.3.1. AGGREGATEINITIALIZE 메소드

AGGREGATION 동작할 때 필요한 객체 인스턴스를 생성하는 함수이다. 최초에 호출되므로 객체 인스턴스가 없게 된다. 이에 따라 반드시 STATIC 함수로 정의해야 한다.

- 프로토타입

```
STATIC FUNCTION TUDIAGGREGATEINITIALIZE
(
  SCTX IN OUT object_type_name
)
RETURN NUMBER;
```

- 파라미터

파라미터	설명
SCTX	AGGREGATION 동작에 필요한 객체를 instanciation한다.

- 반환값

  해당 메소드 동작 후 결과를 반환한다. 결과 값은 TUDICONST에 의미가 정의 되어 있으며, 해당 반환에 따라 쿼리 수행기의 동작이 달라진다.

11.3.2. AGGREGATEITERATE 메소드

단일 컬럼의 값을 입력받아, 객체 인스턴스 애트리뷰트에 중간 결과를 누적한다. 해당 동작은 AGGREGATION 동작이 진행될 컬럼에 대하여 반복 수행된다.

- 프로토타입

```
MEMBER FUNCTION TUDIAGGREGATEITERATE
(
  SELF IN OUT MYSUM_OBJ,
  VALUE IN primitive_type
)
RETURN NUMBER;
```

- 파라미터

파라미터	설명
SELF	객체 인스턴스 자기 자신을 입력으로 받고, 연산 결과를 누적하여 자신의 인스턴스를 업데이트한다.
VALUE	누적할 컬럼값이 해당 입력으로 전달된다. 커서 타입과 복합 타입(오브젝트, 콜렉션, 레코드)은 지원되지 않는다.

- 반환값

  해당 메소드 동작 후 결과를 반환한다. 결과 값은 TUDICONST 패키지에 의미가 정의 되어 있으며, 해당 반환에 따라 쿼리 수행기의 동작이 달라진다.

11.3.3. AGGREGATETERMINATE 메소드

객체 인스턴스에 누적되어 있는 중간 결과로부터 최종의 단일 AGGREGATION 결과 값을 생성한다.

- 프로토타입

```
MEMBER FUNCTION TUDIAGGREGATETERMINATE
(
  SELF IN object_type_name,
  RETURNVALUE OUT NUMBER,
  FLAGS IN NUMBER )
RETURN NUMBER;
```

- 파라미터

파라미터	설명
SELF	연산이 누적된 객체 인스턴스를 참조하기 위함이며, 객체 인스턴스 자기 자신을 입력으로 받는다.
RETURNVALUE	AGGREGATION 결과값이다. 커서 타입과 복합타입(오브젝트, 콜렉션, 레코드)은 지원되지 않는다.
FLAGS	사용되지 않는다.

- 반환값

  해당 메소드 동작 후 결과를 반환한다. 결과 값은 TUDICONST에 의미가 정의 되어 있으며, 해당 반환에 따라 쿼리 수행기의 동작이 달라진다.

11.3.4. AGGREGATEMERGE 메소드

객체 인스턴스 중간 결과 둘을 합하여 하나의 객체 인스턴스로 중간 결과를 생성한다. 해당 동작은 선택적(Optional)으로 정의할수 있다. 해당 동작을 정의하지 않으면 쿼리 수행 엔진에서의 동작이 정의되지 않아, 병렬 수행, 2PASS 알고리즘 등과 같은 동작 제약이 생긴다.

- 프로토타입

```
MEMBER FUNCTION TUDIAGGREGATEMERGE
(
    SELF IN OUT MYSUM_OBJ,
    CTX2 IN MYSUM_OBJ
)
RETURN NUMBER;
```

- 파라미터

파라미터	설명
SELF	객체 인스턴스 자기 자신을 입력으로 받는다. 다른 중간 결과를 병합한 결과이다.
CTX2	병합 대상의 객체 인스턴스이다.

- 반환값

해당 메소드 동작 후 결과를 반환한다. 결과 값은 TUDICONST에 의미가 정의 되어 있으며, 해당 반환에 따라 쿼리 수행기의 동작이 달라진다.

11.4. TUDICONST 패키지

사용자 정의 AGGREGATION 함수의 반환값의 종류를 나타낸다. 반환값에 따라, 쿼리 수행기가 동작을 달리한다.

- 상수

상수명	설명
SUCCESS	콜백 수행 결과가 정상 상태임을 쿼리 수행기에 전달할 때 사용한다.
ERROR	콜백 수행 결과가 비정상 상태임을 쿼리 수행기에 전달할 때 사용한다. 쿼리 수행 중 예외 발생한다.

11.5. 예외

다음은 사용자 정의 AGGREGATION 수행 중 발생할수 있는 예외이다.

- ERROR_DML_UDA_INVALID_ARG_CNT

  AGGREGATION 함수의 파라미터 개수가 맞지 않는 경우 발생하는 에러이다.

- ERROR_DML_UDA_NOT_FOUND_INTERFACE

  쿼리수행기에서 사용자가 정의한 콜백(CALLBACK) 메소드를 찾지 못한 경우 발생하는 에러이다.

- ERROR_DML_UDA_INVALID_ITERATE_ARG

  쿼리 수행기에서 사용자가 정의한 콜백(CALLBACK) 메소드로 파라미터를 전달하지 못했을 때 발생하는 에러이다.

- ERROR_DML_UDA_INVALID_RETURN_TYPE

  사용자가 정의한 콜백(CALLBACK)메소드에서 RETURNVALUE 의 값이 사용자 정의 AGGREGATION 함수의 리턴값으로 변환을 하지 못할때 발생하는 에러이다.

- ERROR_EXEC_UDA_ERROR

  사용자가 정의한 콜백(CALLBACK) 메소드에서 TUDICONST.ERROR를 리턴한 경우에 발생하는 에러이다.

11.6. 간단한 예제

다음은 AGGREGATION에 필요한 객체 타입을 정의하는 예이다.

```
CREATE OR REPLACE TYPE AGGR_STRING_OBJTYPE AS OBJECT (
   data VARCHAR2(32767),
   STATIC FUNCTION TUDIAGGREGATEINITIALIZE( SCTX IN OUT AGGR_STRING_OBJTYPE )
   RETURN NUMBER,
   MEMBER FUNCTION TUDIAGGREGATEITERATE( SELF IN OUT AGGR_STRING_OBJTYPE,
                                         VALUE IN VARCHAR2)
   RETURN NUMBER,
   MEMBER FUNCTION TUDIAGGREGATETERMINATE( SELF IN AGGR_STRING_OBJTYPE,
                                           RETURNVALUE OUT VARCHAR2,
   FLAGS IN NUMBER )
   RETURN NUMBER,
   MEMBER FUNCTION TUDIAGGREGATEMERGE( SELF IN OUT AGGR_STRING_OBJTYPE,
                                      CTX2 IN AGGR_STRING_OBJTYPE)
   RETURN NUMBER
);
/

CREATE OR REPLACE TYPE BODY AGGR_STRING_OBJTYPE
IS
   STATIC FUNCTION TUDIAGGREGATEINITIALIZE( SCTX IN OUT AGGR_STRING_OBJTYPE )
    RETURN NUMBER
    IS
    BEGIN
      SCTX := AGGR_STRING_OBJTYPE(NULL);
      RETURN TUDICONST.SUCCESS;
   END;
   MEMBER FUNCTION TUDIAGGREGATEITERATE( SELF IN OUT AGGR_STRING_OBJTYPE,
                                         VALUE IN VARCHAR2)
    RETURN NUMBER
    IS
    BEGIN

      IF VALUE IS NOT NULL THEN
        IF SELF.data IS NULL THEN
          SELF.data := VALUE;
        ELSE
          SELF.data := SELF.data || ',' || VALUE;
        END IF;
      END IF;

      RETURN TUDICONST.SUCCESS;

   EXCEPTION WHEN OTHERS THEN
```

```
      RETURN TUDICONST.ERROR;
   END;
   MEMBER FUNCTION TUDIAGGREGATETERMINATE( SELF IN AGGR_STRING_OBJTYPE,
                                           RETURNVALUE OUT VARCHAR2,
                                           FLAGS IN NUMBER )
    RETURN NUMBER
    IS
    BEGIN
      RETURNVALUE := SELF.data;
      RETURN TUDICONST.SUCCESS;
   END;
   MEMBER FUNCTION TUDIAGGREGATEMERGE( SELF IN OUT AGGR_STRING_OBJTYPE,
                                       CTX2 IN AGGR_STRING_OBJTYPE)
    RETURN NUMBER
    IS
    BEGIN
      SELF.data := SELF.data || ',' || CTX2.data;
      RETURN TUDICONST.SUCCESS;

   EXCEPTION WHEN OTHERS THEN
      RETURN TUDICONST.ERROR;
   END;
END;
/
```

정의한 객체 타입으로 AGGREGATION 함수를 정의한다.

```
CREATE OR REPLACE FUNCTION AGGR_STRING(input VARCHAR) RETURN VARCHAR
PARALLEL_ENABLE AGGREGATE USING AGGR_STRING_OBJTYPE;
/
```

다음은 AGGREGATION 함수의 실제 사용의 예이다. 미리 정의된 AGGREGATION이 아닌 사용자가 정의한 AGGREGATION 동작이 SQL을 통해 수행된다.

```
CREATE TABLE BEST_COMPANY (
  group_name VARCHAR(32),
  affiliate VARCHAR(32));

Table 'BEST_COMPANY' created.

INSERT INTO BEST_COMPANY VALUES('티맥스', '티맥스소프트');

1 row inserted.

INSERT INTO BEST_COMPANY VALUES('티맥스', '티맥스데이터');

1 row inserted.
```

```
INSERT INTO BEST_COMPANY VALUES('티맥스', '티맥스글로벌');

1 row inserted.

COMMIT;

Commit completed.


-- BEST 회사 테이블에서 티맥스의 계열사의 이름을 다 붙여서
-- 하나의 값으로 보고 싶다.
SELECT  aggr_string(affiliate)
FROM    best_company
WHERE   group_name = '티맥스';


AGGR_STRING(AFFILIATE)
--------------------------------------------------------------------------------
티맥스소프트,티맥스데이터,티맥스글로벌

1 row selected.

/
```

제12장 BULK SQL

본 장에서는 BULK 로 수행하는 SQL 에 대해 설명한다.

12.1. 개요

BULK SQL 은 연속된 여러 번의 SQL 을 BATCH 단위로 한 번에 처리하는 것을 말한다. PSM 에서 SQL 처리 엔진으로 메시지를 BATCH 단위로 처리하기 때문에 일반적으로 연속된 SQL 처리보다 훨씬 빠른 성능상의 이점이 있다.

12.2. FORALL 문

FORALL 문은 FOR-LOOP 문과 비슷하나 하나의 DML 만을 사용할 수 있으며 batch 단위 로 처리하여 실행이 빠른 반복문이다. 단, index 를 DML 에 직접적으로 명시할 수는 없으며 반드시 다른 collection 의 index 로 사용해야 한다. Index 로 사용하므로 FOR-LOOP 의 REVERSE 는 사용할 수 없고, collection 자체의 순서를 바꿔야 한다.

기본적인 사용법은 다음과 같다.

```
FORALL index IN low_bound..high_bound
    DML 실행문;
```

항목	설명
index	일반적으로 변수를 사용한다. 시스템이 내부적으로 선언하기 때문에 명시적으로 선언할 필요가 없다. 다른 collection 의 참조 index 로 사용해야 한다. index를 볼 수 있는 범위는 FORALL 문의 내부이다.
low_bound, high_bound	LOOP 문의 범위를 지정한다. 주로 숫자 상수를 사용한다. 그러나 반드시 숫자 상수일 필요는 없으며, 숫자 상수로 변환될 수 있는 임의의 식을 사용할 수 있다.

다음은 FORALL 문의 예이다.

```
DROP TABLE T1;
CREATE TABLE T1 (C1 NUMBER, C2 VARCHAR2(20));
DECLARE
```

```
  TYPE NumTab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  TYPE NameTab IS TABLE OF VARCHAR2(20) INDEX BY PLS_INTEGER;
  nums  NumTab;
  names NameTab;
BEGIN

FOR i IN 1..10 LOOP
  nums(i) := i;
  names(i) := 'Name' || TO_CHAR(i);
END LOOP;

FORALL i IN 1..10
  INSERT INTO T1 VALUES(nums(i), names(i));
END;
```

위 예에서 보듯이 index는 i 라는 변수로 정의하고, low_bound와 high_bound는 1부터 시작하여 10에 도달할 때까지 값을 1씩 증가시키며 LOOP 문을 반복하며 미리 만들어 놓은 nums 와 names 의 index 로 i 를 사용한다. FOR-LOOP 문과 마찬가지로 low_bound와 high_bound는 반드시 숫자 상수일 필요는 없다. 예를 들어 다음과 같이 쓸 수 있다.

```
DECLARE
  TYPE NumArr IS VARRAY(20) OF NUMBER;
  nums NumArr := NumArr(2, 5, 7);
BEGIN
  FORALL i IN nums.FIRST..nums.LAST
    DELETE FROM T1 WHERE C1 = nums(i);
END;
```

SAVE EXCEPTIONS 예약어의 사용

FORALL 문 안에서 SAVE EXCEPTIONS 예약어를 사용하게 되면 해당 DML 이 실패해도 계속 진행하게 된다. 이 때 해당 EXCEPTION 을 저장해놓고 나머지 DML 들을 실행한 뒤 FORALL 문이 종료되고 EXCEPTION 이 처리된다. SAVE EXCEPTIONS 를 명시하지 않으면 DML 이 실패하는 즉시 EXCEPTION 이 발생하고, FORALL 문이 종료한다. 예시는 다음과 같다.

```
DROP TABLE T1;
CREATE TABLE T1 (C1 NUMBER(1), C2 VARCHAR2(20));
DECLARE
  TYPE NumTab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  TYPE NameTab IS TABLE OF VARCHAR2(20) INDEX BY PLS_INTEGER;
  nums  NumTab;
  names NameTab;
BEGIN
```

```
FOR i IN 1..10 LOOP
  nums(i) := 11-i;
  names(i) := 'Name' || TO_CHAR(11-i);
END LOOP;

FORALL i IN 1..10 SAVE EXCEPTIONS
  INSERT INTO T1 VALUES(nums(i), names(i));
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('DML Error Occurred.');
END;
```

위 예시에서는 precesion 이 1 로 설정된 column 에 10 부터 1 까지를 INSERT 하려 하고 있다. SAVE EXCEPTIONS 를 명시하지 않으면 INSERT 가 실패하는 즉시 FORALL 문이 종료되지만 SAVE EXCEPTIONS 를 명시하였으므로 10 에서 에러를 발생다음 9 부터 1 까지는 실행 된다.

SAVE EXCEPTIONS 를 명시하고 FORALL 문이 끝나면 -15231 ERROR_PSM_PATCH_DML_FAILED 단 하나의 EXCEPTION 이 발생하지만 묵시적 커서 속성 SQL%BULK_EXCEPTIONS 를 통해 FORALL 문 수행 도중 발생한 여러 EXCEPTION 들에 대한 정보를 받아올 수 있다. SQL%BULK_EXCEPTIONS 는 BULK SQL 수행시 발생하는 여러 EXCEPTION 에 대한 ASSOCIATIVE ARRAY 로, SQL%BULK_EXCEPTIONS.COUNT 가 EXCEPTION 개수를 나타낸다. 각 에러 i번째에 대하여 SQL%BULK_EXCEPTIONS(i).ERROR_CODE 는 해당 SQL 의 TIBERO 에러코드를, SQL%BULK_EXCEPTIONS(i).ERROR_INDEX 는 FORALL 문 내에서 몇 번째 실행의 에러인지 나타낸다. 다음은 그 예시이다.

```
DROP TABLE T1;
CREATE TABLE T1 (C1 NUMBER(1), C2 VARCHAR2(20));
DECLARE
  TYPE NumTab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  TYPE NameTab IS TABLE OF VARCHAR2(20) INDEX BY PLS_INTEGER;
  nums  NumTab;
  names NameTab;

  error_message VARCHAR2(100);
  bad_stmt_no   PLS_INTEGER;
  dml_errors EXCEPTION;
  PRAGMA EXCEPTION_INIT(dml_errors, -15231);

BEGIN

FOR i IN 1..10 LOOP
  nums(i) := 12-i;
  names(i) := 'Name' || TO_CHAR(12-i);
END LOOP;

FORALL i IN 1..11 SAVE EXCEPTIONS
```

```
  INSERT INTO T1 VALUES(nums(i), names(i));
EXCEPTION
  WHEN dml_errors THEN
    FOR i IN 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
      error_message := SQLERRM(-(SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
      DBMS_OUTPUT.PUT_LINE (error_message);
      bad_stmt_no := SQL%BULK_EXCEPTIONS(i).ERROR_INDEX;
    END LOOP;
END;
```

INDICES OF 예약어의 사용

IN lowbound..highbound 대신 INDICES OF collection 을 사용하여 해당 collection 의 인덱스를 사용할 수 있다. 이 경우, 뒤에 BETWEEN lower_bound AND upper_bound 를 명시하여 사용할 인덱스를 제한할 수 있다. 예시는 다음과 같다.

```
DROP TABLE T1;
CREATE TABLE T1 (C1 NUMBER, C2 VARCHAR2(20));
DECLARE
  TYPE NumArr IS VARRAY(20) OF NUMBER;
  nums NumArr := NumArr(2, 5, 7, 9);
BEGIN
  FORALL i IN INDICES OF nums BETWEEN 2 and 3
    INSERT INTO T1 VALUES (nums(i), TO_CHAR(nums(i)));
END;
```

주의할 점은 위 예시의 VARRAY 내에서 2 와 3 사이의 값을 사용하는게 아니고 두 번째, 세 번째 값 까지 사용한다는 점이다. COLLECTION 내의 인덱스는 1 부터 시작한다.

VALUES OF 예약어의 사용

IN lowbound..highbound 대신 VALUES OF collection 을 사용하여 해당 collection 의 값을 사용할 수 있다. 단, collection 의 원소들은 PLS_INTEGER 이거나 BINARY_INTEGER 여야 한다. 예시는 다음과 같다.

```
DROP TABLE T1;
CREATE TABLE T1 (C1 NUMBER, C2 VARCHAR2(20));
DECLARE
  TYPE NumTab IS TABLE OF PLS_INTEGER;
  nums NumTab := NumTab(1, 2, 3);
BEGIN
```

```
FORALL i IN VALUES OF nums
  INSERT INTO T1 VALUES(nums(i), TO_CHAR(nums(i)));
END;
```

Appendix A. 예약어

본 장에서는 Tibero에서 사용하는 tbPSM의 예약어를 알파벳 순으로 기술한다.

참고

예약어는 상수, 변수명, 커서명 등에 사용할 수 없다.

A.1. B

```
BEGIN
BODY
BOTH
```

A.2. C

```
CASE
COMMIT
CONSTANT
CONSTRUCTOR
CURSOR
```

A.3. D

```
DELETE
DISTINCT
```

A.4. E

```
ELSE
ELSIF
END
EXCEPTION
EXIT
```

A.5. F

```
FETCH
FINAL
FROM
FUNCTION
```

A.6. G

```
GOTO
```

A.7. I

```
IF
IN
INDICATOR
INSERT
INSTANTIABLE
```

A.8. L

```
LANGUAGE
LEADING
LOOP
```

A.9. M

```
MAP
MEMBER
MERGE
```

A.10. N

```
NOCOPY
NOT
NULL
```

A.11. O

```
ORDER
OTHERS
OUT
OVERRIDING
```

A.12. R

```
RAISE
REF
RETURN
REVERSE
ROLLBACK
```

A.13. S

```
SAVEPOINT
SELECT
SELF
SHORT
SQL
STATIC
SUBTYPE
```

A.14. T

```
TRAILING
TYPE
```

A.15. U

```
UPDATE
```

A.16. W

```
WHEN
WHILE
```

Appendix B. tbPSM 소스코드 암호화

본 장에서는 tbPSM 언어로 작성된 소스코드를 암호화하는 방법을 기술한다.

B.1. 개요

Tibero는 tbPSM 언어 개발자가 작성한 소스코드를 일반 사용자 혹은 경쟁자들에게 노출되지 않도록 암호화하는 기능을 제공한다.

사용자는 tbwrap 프로그램을 이용하여 소스 코드를 암호화할 수 있다. tbwrap 프로그램은 tbPSM 언어로 작성된 소스 파일을 암호화하여 파일로 저장하고, 암호화된 DDL문으로 프러시저, 패키지, 함수 등을 데이터베이스에 생성할 수 있다.

B.2. tbwrap 프로그램으로 tbPSM 코드 암호화하기

tbwrap 프로그램은 SQL 파일을 입력으로 받아서 프러시저, 함수, 패키지 선언부, 패키지 구현부와 같은 tbwrap 객체만 암호화한다. 익명 블록이나 트리거, tbwrap가 아닌 객체들은 암호화하지 않는다.

운영체제 프롬프트 창에서 아래와 같이 입력하여 프로그램을 실행한다.

```
tbwrap iname=input_file [ oname=output_file ]
```

예제는 아래와 같다.

```
-- 현재 경로에 확장자만 달라진 test.tbw라는 이름의 파일로 출력된다.
tbwrap iname=test.sql

-- 현재 경로에 result.tbw라는 이름의 파일로 출력된다.
tbwrap iname=test.sql oname=result.tbw

-- 디렉터리도 지정할 수 있음
tbwrap iname=/far/test.sql oname=/boo/boo2/test.tbw
```

암호화된 파일을 열어보면 아래와 같이 작성되어 있다.

```
create or replace procedure p wrapped
{wrapped_text}
/
```

위와 같이 작성된 DDL로 tbPSM 객체를 컴파일하여 생성할 수 있다. 생성 후 *_SOURCE 계열 뷰로 조회하면 암호화된 채로 저장되었음을 확인할 수 있다.

B.3. 예제

프러시저를 암호화하고 데이터베이스에 생성하는 과정을 예제를 통해 설명한다.

cre.sql 파일에는 아래와 같은 DDL문이 있다.

```
create or replace procedure p (a number)
is
    b number := 12;
begin
    for i in 1..100 loop
        dbms_output.put_line(a + b - i);
    end loop;
end;
/
```

개발자는 위 파일을 아래와 같이 암호화화여 생성된 cre.tbw 파일을 사용자에게 배포한다.

```
tbwrap iname=cre.sql oname=cre.tbw

tbWrap 5 Trunk

Copyright (c) 2008, 2009, 2011, 2012 TmaxData Corporation. All rights reserved.

Processing cre.sql to cre.tbw.............................Done
```

cre.tbw을 열어보면 암호화되어 있음을 확인할 수 있다.

```
create or replace procedure p wrapped
eHH4Ck8sjWn5HYfQr/vlvToOrCyf+W0VGwH/kQ/pFtaCNMMz0s6KVwxkIoYVLQhAIhsO9DTZttreui2mgun
vxR4AuG5sovFku/ep67aWrOabjKBeyu+Ni5PSWVuBZh+W+bBr38w4XRF9Ank8IHkJo4pMwkAshBgyJLiVrA
by3EQZiWKvMhBNI87Ac2hdLM9z+Bck1QhMOULlf2aItYZ3cg==
/
```

사용자는 파일을 받아서 tbsql 등의 프로그램으로 실행하여 프러시저를 생성한다.

```
$ tbsql tibero/tmax

tbSQL 5 Trunk

Copyright (c) 2008, 2009, 2011, 2012 TmaxData Corporation. All rights reserved.

Connected to Tibero.

SQL> @cre.tbw

Procedure 'P' created.

File finished.
```

이제 사용자는 생성된 프러시저를 사용할 수 있다.

```
SQL> call p(1);

PSM called.
```

B.4. 주의사항

다음은 tbPSM 소스코드 암호화의 주의사항에 대한 설명이다.

- 패키지를 배포할 경우 사용자에게 사용 방법은 노출시키되 구현 내용은 노출되지 말아야 한다. 따라서 선언부는 암호화하지 말고, 구현부만 암호화하여 배포하도록 한다.
- 암호화된 구문은 복호화가 불가능하므로 원본 소스 코드를 분실하지 않도록 한다.
- 트리거는 암호화되지 않으므로 내부 구문에서 암호화된 서브 프로그램을 호출하도록 한다.
- tbwrap 프로그램은 문법 체크는 하지 않으므로 암호화된 파일로 컴파일해야 문법 오류를 확인할 수 있다.

색인

J

L

M

N

O

P

Q

R

S

T

V

W

X

ㄱ

ㄷ

ㄹ

ㅁ

ㅂ

ㅅ

ㅇ

ㅈ

ㅊ

ㅋ

ㅌ

ㅍ

ㅎ