Db2 11.1 for Linux, UNIX, and Windows

PL/SQL Support

Db2 11.1 for Linux, UNIX, and Windows

PL/SQL Support

Notice regarding this document

This document in PDF form is provided as a courtesy to customers who have requested documentation in this format. It is provided As-Is without warranty or maintenance commitment.

Contents

Figures

Tables

PL/SQL support

PL/SQL (Procedural Language/Structured Query Language) statements can be compiled and executed using data server interfaces provided by Db2®. This reduces the complexity of enabling existing PL/SQL solutions to work with the Db2 data server.

The data server interfaces you can use include:

- The Db2 command line processor (CLP)
- CLPPlus
- IBM® Data Studio client

Enablement

To enable this capability, set the **DB2_COMPATIBILITY_VECTOR** registry variable to hexadecimal 0x800 (bit position 800), then stop and restart the instance:

```
db2set DB2_COMPATIBILITY_VECTOR=800
db2stop
db2start
```

To activate all compatibility features for Oracle applications, set the **DB2_COMPATIBILITY_VECTOR** registry variable to ORA, then stop and restart the instance:

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
```

PL/SQL features

PL/SQL statements and scripts can be compiled and executed using Db2 database server interfaces.

You can execute the following PL/SQL statements:

- Anonymous blocks; for example, DECLARE...BEGIN...END
- CREATE OR REPLACE FUNCTION statement
- CREATE OR REPLACE PACKAGE statement
- CREATE OR REPLACE PACKAGE BODY statement
- CREATE OR REPLACE PROCEDURE statement
- CREATE OR REPLACE TRIGGER statement
- CREATE OR REPLACE TYPE statement
- DROP PACKAGE statement
- DROP PACKAGE BODY statement

PL/SQL procedures and functions can be invoked from other PL/SQL statements or from Db2 SQL PL statements. You can call a PL/SQL procedure from SQL PL by using the CALL statement.

The following statements and language elements are supported in PL/SQL contexts:

- Type declarations:

  - Associative arrays
  - Record types
  - VARRAY types
- Subtype declarations
- Variable declarations:
  - %ROWTYPE
  - %TYPE
- Basic statements, clauses, and statement attributes:
  - Assignment statement
  - NULL statement
  - RETURNING INTO clause
  - Statement attributes, including SQL%FOUND, SQL%NOTFOUND, and SQL%ROWCOUNT
- Control statements and structures:
  - CASE statements:
    - Simple CASE statement
    - Searched CASE statement
  - Exception handling
  - EXIT statement
  - FOR statement
  - GOTO statement
  - IF statement
  - LOOP statement
  - PIPE ROW statement
  - RETURN statement
  - WHILE statement
- Static cursors:
  - CLOSE statement
  - Cursor FOR loop statement
  - FETCH statement (including FETCH INTO a %ROWTYPE variable)
  - OPEN statement
  - Parameterized cursors
  - Cursor attributes
- REF CURSOR support:
  - Variables and parameters of type REF CURSOR
  - Strong REF CURSORs
  - OPEN FOR statement
  - Returning REF CURSORs to JDBC applications
- Error support:
  - RAISE_APPLICATION_ERROR procedure
  - RAISE statement
  - SQLCODE function
  - SQLERRM function

Creating PL/SQL procedures and functions from a CLP script

You can create PL/SQL procedures and functions from a Db2 command line processor (CLP) script.

Procedure

1. Formulate PL/SQL procedure or function definitions within a CLP script file. Terminate each statement with a new line and a forward slash character (/). Other statement termination characters are also supported.
2. Save the file. In this example, the file name is `script.db2`.
3. Execute the script from the CLP. If a forward slash character or a semicolon was used to terminate statements, issue the following command:

   ```
   db2 -td/ -vf script.db2
   ```

   If another statement termination character (for example, the @ character) was used in the script file, you must specify that character in the command string. For example:

   ```
   db2 -td@ -vf script.db2
   ```

Results

The CLP script should execute successfully if there are no syntax errors.

Example

The following example of a CLP script creates a PL/SQL function and procedure, and then calls the PL/SQL procedure.

```
CONNECT TO mydb
/

CREATE TABLE emp (
     name            VARCHAR2(10),
     salary          NUMBER,
     comm            NUMBER,
     tot_comp        NUMBER
)
/

INSERT INTO emp VALUES ('Larry', 1000, 50, 0)
/
INSERT INTO emp VALUES ('Curly', 200, 5, 0)
/
INSERT INTO emp VALUES ('Moe', 10000, 1000, 0)
/

CREATE OR REPLACE FUNCTION emp_comp (
     p_sal           NUMBER,
     p_comm          NUMBER )
RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp
/

CREATE OR REPLACE PROCEDURE update_comp(p_name IN VARCHAR) AS
BEGIN
    UPDATE emp SET tot_comp = emp_comp(salary, comm)
      WHERE name = p_name;
END update_comp
```

```
/

CALL update_comp('Curly')
/

SELECT * FROM emp
/

CONNECT RESET
/
```

This script produces the following sample output:

```
CALL update_comp('Curly')

  Return Status = 0

SELECT * FROM emp

NAME       SALARY             COMM               TOT_COMP
---------- ------...--------- ----...---------- --------...-------
Larry                   1000                 50                  0
Curly                    200                  5               4920
Moe                    10000               1000                  0

  3 record(s) selected.
```

What to do next

Test your new procedures or functions by invoking them. For procedures, use the CALL statement. For functions, execute queries or other SQL statements that contain references to those functions.

Restrictions on PL/SQL support

It is important to note the restrictions on PL/SQL compilation support before performing PL/SQL compilation, or when troubleshooting PL/SQL compilation or runtime problems.

In this version:

- PL/SQL procedures, functions, triggers, and packages can only be created from the catalog partition in a partitioned database environment.
- The NCLOB data type is not supported for use in PL/SQL statements or in PL/SQL contexts when the database is not defined as a Unicode database. In Unicode databases, the NCLOB data type is mapped to a Db2 DBCLOB data type.
- The XMLTYPE data type is not supported.
- In a partitioned database environment, you cannot access cursor variables from remote nodes. You can only access cursor variables from the coordinator node.
- The use of nested type data types with PL/SQL package variables is not supported in autonomous routines.

PL/SQL sample schema

Most of the PL/SQL examples are based on a PL/SQL sample schema that represents employees in an organization.

The following script (`plsql_sample.sql`) defines that PL/SQL sample schema.

```
--
--  Script that creates the 'sample' tables, views, procedures,
--  functions, triggers, and so on.
--
--  Create and populate tables used in the documentation examples.
--
--  Create the 'dept' table
--
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14) NOT NULL CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR2(13)
);
--
--  Create the 'emp' table
--
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
--
--  Create the 'jobhist' table
--
CREATE TABLE jobhist (
    empno           NUMBER(4) NOT NULL,
    startdate       DATE NOT NULL,
    enddate         DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2),
    chgdesc         VARCHAR2(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
    CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
        REFERENCES emp(empno) ON DELETE CASCADE,
    CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
        REFERENCES dept (deptno) ON DELETE SET NULL,
    CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
);
--
--  Create the 'salesemp' view
--
CREATE OR REPLACE VIEW salesemp AS
    SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
--
--  Sequence to generate values for function 'new_empno'
--
CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
--
--  Issue PUBLIC grants
--
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
--
--  Load the 'dept' table
--
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
```

```
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');
--
--  Load the 'emp' table
--
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-81',2975,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-81',1250,1400,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-81',2850,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-81',2450,NULL,10);
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-87',3000,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'17-NOV-81',5000,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3000,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);
--
--  Load the 'jobhist' table
--
INSERT INTO jobhist VALUES (7369,'17-DEC-80',NULL,'CLERK',800,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,
  'New Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,
  'New Hire');
INSERT INTO jobhist VALUES (7566,'02-APR-81',NULL,'MANAGER',2975,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7654,'28-SEP-81',NULL,'SALESMAN',1250,1400,30,
  'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,
  'New Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-88','CLERK',1000,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-89','CLERK',1040,NULL,20,
  'Raise');
INSERT INTO jobhist VALUES (7788,'05-MAY-90',NULL,'ANALYST',3000,NULL,20,
  'Promoted to Analyst');
INSERT INTO jobhist VALUES (7839,'17-NOV-81',NULL,'PRESIDENT',5000,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7844,'08-SEP-81',NULL,'SALESMAN',1500,0,30,
  'New Hire');
INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-83','CLERK',950,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7900,'15-JAN-83',NULL,'CLERK',950,NULL,30,
  'Changed to Dept 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,
  'New Hire');

SET SQLCOMPAT PLSQL;
--
--  Procedure that lists all employees' numbers and names
--  from the 'emp' table using a cursor
--
CREATE OR REPLACE PROCEDURE list_emp
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur IS
```

```
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;
/
--
--  Procedure that selects an employee row given the employee
--  number and displays certain columns
--
CREATE OR REPLACE PROCEDURE select_emp (
    p_empno         IN  NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_dname         dept.dname%TYPE;
    v_disp_date     VARCHAR2(10);
BEGIN
    SELECT ename, hiredate, sal, NVL(comm, 0), dname
        INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
        FROM emp e, dept d
        WHERE empno = p_empno
          AND e.deptno = d.deptno;
    v_disp_date := TO_CHAR(v_hiredate, 'YYYY/MM/DD');
    DBMS_OUTPUT.PUT_LINE('Number    : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name      : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
    DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
    DBMS_OUTPUT.PUT_LINE('Department: ' || v_dname);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
/
--
--  Procedure that queries the 'emp' table based on
--  department number and employee number or name. Returns
--  employee number and name as IN OUT parameters and job,
--  hire date, and salary as OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query (
    p_deptno        IN     NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
    p_job           OUT    VARCHAR2,
    p_hiredate      OUT    DATE,
    p_sal           OUT    NUMBER
)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
```

```
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
          AND (empno = p_empno
           OR  ename = UPPER(p_ename));
END;
/
--
--  Procedure to call 'emp_query_caller' with IN and IN OUT
--  parameters. Displays the results received from IN OUT and
--  OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query_caller
IS
    v_deptno        NUMBER(2);
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 0;
    v_ename  := 'Martin';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee was selected');
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No employees were selected');
END;
/
--
--  Function to compute yearly compensation based on semimonthly
--  salary
--
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal           NUMBER,
    p_comm          NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
/
--
--  After statement-level triggers that display a message after
--  an insert, update, or deletion to the 'emp' table. One message
--  per SQL command is displayed.
--
CREATE OR REPLACE TRIGGER user_ins_audit_trig
    AFTER INSERT ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' added employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
```

```
CREATE OR REPLACE TRIGGER user_upd_audit_trig
    AFTER UPDATE ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' updated employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
CREATE OR REPLACE TRIGGER user_del_audit_trig
    AFTER DELETE ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' deleted employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
--
--  Before row-level triggers that display employee number and
--  salary of an employee that is about to be added, updated,
--  or deleted in the 'emp' table
--
CREATE OR REPLACE TRIGGER emp_ins_sal_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
    DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
END;
/
CREATE OR REPLACE TRIGGER emp_upd_sal_trig
    BEFORE UPDATE ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    sal_diff := :NEW.sal - :OLD.sal;
    DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
    DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
    DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
    DBMS_OUTPUT.PUT_LINE('..Raise     : ' || sal_diff);
END;
/
CREATE OR REPLACE TRIGGER emp_del_sal_trig
    BEFORE DELETE ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
    DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
END;
/
--
--  Package specification for the 'emp_admin' package
--
CREATE OR REPLACE PACKAGE emp_admin
IS
    FUNCTION get_dept_name (
        p_deptno        NUMBER
```

```
    ) RETURN VARCHAR2;
    FUNCTION update_emp_sal (
        p_empno         NUMBER,
        p_raise         NUMBER
    ) RETURN NUMBER;
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    );
    PROCEDURE fire_emp (
        p_empno         NUMBER
    );
END emp_admin;
/
--
--  Package body for the 'emp_admin' package
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
    --
    --  Function that queries the 'dept' table based on the department
    --  number and returns the corresponding department name
    --
    FUNCTION get_dept_name (
        p_deptno        IN NUMBER
    ) RETURN VARCHAR2
    IS
        v_dname         VARCHAR2(14);
    BEGIN
        SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
        RETURN v_dname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
            RETURN '';
    END;
    --
    --  Function that updates an employee's salary based on the
    --  employee number and salary increment/decrement passed
    --  as IN parameters. Upon successful completion the function
    --  returns the new updated salary.
    --
    FUNCTION update_emp_sal (
        p_empno         IN NUMBER,
        p_raise         IN NUMBER
    ) RETURN NUMBER
    IS
        v_sal           NUMBER := 0;
    BEGIN
        SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
        v_sal := v_sal + p_raise;
        UPDATE emp SET sal = v_sal WHERE empno = p_empno;
        RETURN v_sal;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
            RETURN -1;
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
            DBMS_OUTPUT.PUT_LINE(SQLERRM);
            DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
```

```
            DBMS_OUTPUT.PUT_LINE(SQLCODE);
            RETURN -1;
    END;
    --
    --  Procedure that inserts a new employee record into the 'emp' table
    --
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    )
    AS
    BEGIN
        INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
            VALUES(p_empno, p_ename, p_job, p_sal,
                   p_hiredate, p_comm, p_mgr, p_deptno);
    END;
    --
    --  Procedure that deletes an employee record from the 'emp' table based
    --  on the employee number
    --
    PROCEDURE fire_emp (
        p_empno         NUMBER
    )
    AS
    BEGIN
        DELETE FROM emp WHERE empno = p_empno;
    END;
END;
/

SET SQLCOMPAT DB2;
```

Obfuscation

Obfuscation encodes the body of the DDL statements for database objects such as routines, triggers, views, and PL/SQL packages. Obfuscating your code helps protect your intellectual property because users cannot read the code, but the Db2 data server can still understand it.

The DBMS_DDL module provides two routines for obfuscating your routines, triggers, views, or your PL/SQL packages:

WRAP function

Takes a routine, trigger, PL/SQL package, or PL/SQL package body definition as an argument and produces a string containing the initial header followed by an obfuscated version of the rest of the statement. For example, input like:

```
CREATE PROCEDURE P(a INT)
BEGIN
  INSERT INTO T1 VALUES (a);
END
```

using the DBMS_DDL.WRAP function might result in:

```
CREATE PROCEDURE P(a INT) WRAPPED SQL09072
aBcDefg12AbcasHGJG6JKHhgkjFGHHkkkljljk878979HJHui99
```

The obfuscated portion of the DDL statement contains codepage invariant characters, ensuring that it is valid for any codepage.

CREATE_WRAPPED procedure
Takes the same input as the WRAP function described previously, but instead of returning the obfuscated text, an object is created in the database. Internally the object is not obfuscated so that it can be processed by the compiler, but in catalog views like SYSCAT.ROUTINES or SYSCAT.TRIGGERS the content of the TEXT column is obfuscated.

An obfuscated statement can be used in CLP scripts and can be submitted as dynamic SQL using other client interfaces.

Obfuscation is available for the following statements:

- **db2look** by using the **-wrap** option
- CREATE FUNCTION
- CREATE PACKAGE
- CREATE PACKAGE BODY
- CREATE PROCEDURE
- CREATE TRIGGER
- CREATE VIEW
- ALTER MODULE

The **db2look** tool obfuscates all the preceding statements when the **-wrap** option is used.

Blocks (PL/SQL)

PL/SQL block structures can be included within PL/SQL procedure, function, or trigger definitions or executed independently as an anonymous block statement.

PL/SQL block structures and the anonymous block statement contain one or more of the following sections:

- An optional declaration section
- A mandatory executable section
- An optional exception section

These sections can include SQL statements, PL/SQL statements, data type and variable declarations, or other PL/SQL language elements.

Anonymous block statement (PL/SQL)

The PL/SQL anonymous block statement is an executable statement that can contain PL/SQL control statements and SQL statements. It can be used to implement procedural logic in a scripting language. In PL/SQL contexts, this statement can be compiled and executed by the Db2 data server.

The anonymous block statement, which does not persist in the database, can consist of up to three sections: an optional declaration section, a mandatory executable section, and an optional exception section.

The optional declaration section, which can contain the declaration of variables, cursors, and types that are to be used by statements within the executable and exception sections, is inserted before the executable BEGIN-END block.

The optional exception section can be inserted near the end of the BEGIN-END block. The exception section must begin with the keyword EXCEPTION, and continues until the end of the block in which it appears.

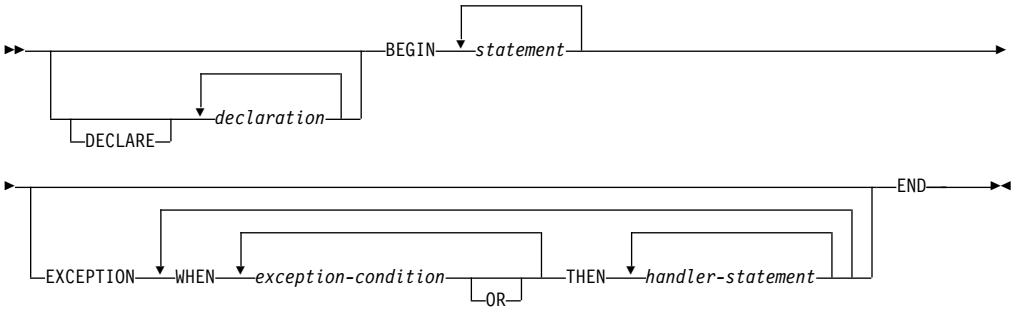
Invocation

This statement can be executed from an interactive tool or command line interface such as the CLP. This statement can also be embedded within a PL/SQL procedure definition, function definition, or trigger definition. Within these contexts, the statement is called a block structure instead of an anonymous block statement.

Authorization

No privileges are required to invoke an anonymous block. However, the privileges held by the authorization ID of the statement must include all necessary privileges to invoke the SQL statements that are embedded within the anonymous block.

Syntax



Description

DECLARE
An optional keyword that starts the DECLARE statement, which can be used to declare data types, variables, or cursors. The use of this keyword depends upon the context in which the block appears.

declaration
Specifies a data type, variable, cursor, exception, or procedure declaration whose scope is local to the block. Each declaration must be terminated by a semicolon.

BEGIN
A mandatory keyword that introduces the executable section, which can include one or more SQL or PL/SQL statements. A BEGIN-END block can contain nested BEGIN-END blocks.

statement
Specifies a PL/SQL or SQL statement. Each statement must be terminated by a semicolon.

EXCEPTION
An optional keyword that introduces the exception section.

WHEN *exception-condition*
Specifies a conditional expression that tests for one or more types of exceptions.

THEN *handler-statement*
Specifies a PL/SQL or SQL statement that is executed if a thrown exception matches an exception in *exception-condition*. Each statement must be terminated by a semicolon.

END
A mandatory keyword that ends the block.

Examples

The following example shows the simplest possible anonymous block statement that the Db2 data server can compile:

```
BEGIN
    NULL;
END;
```

The following example shows an anonymous block that you can enter interactively through theDb2 CLP:

```
SET SERVEROUTPUT ON;

BEGIN
  dbms_output.put_line( 'Hello' );
END;
```

The following example shows an anonymous block with a declaration section that you can enter interactively through the Db2 CLP:

```
SET SERVEROUTPUT ON;

DECLARE
   current_date DATE := SYSDATE;
BEGIN
     dbms_output.put_line( current_date );
END;
```

Procedures (PL/SQL)

The Db2 data server supports the compilation and execution of PL/SQL procedures.

PL/SQL procedures are database objects that contain PL/SQL procedural logic and SQL statements that can be invoked in contexts where the CALL statement or procedure references are valid.

PL/SQL procedures are created by executing the PL/SQL CREATE PROCEDURE statement. Such procedures can be dropped from the database by using the Db2 SQL DROP statement. If you want to replace the implementation for a procedure, you do not need to drop it. You can use the CREATE PROCEDURE statement and specify the OR REPLACE option to replace the procedure implementation.

CREATE PROCEDURE statement (PL/SQL)

The CREATE PROCEDURE statement defines a procedure that is stored in the database.

Invocation

This statement can be executed from the Db2 command line processor (CLP), any supported interactive SQL interface, an application, or a routine.

Authorization

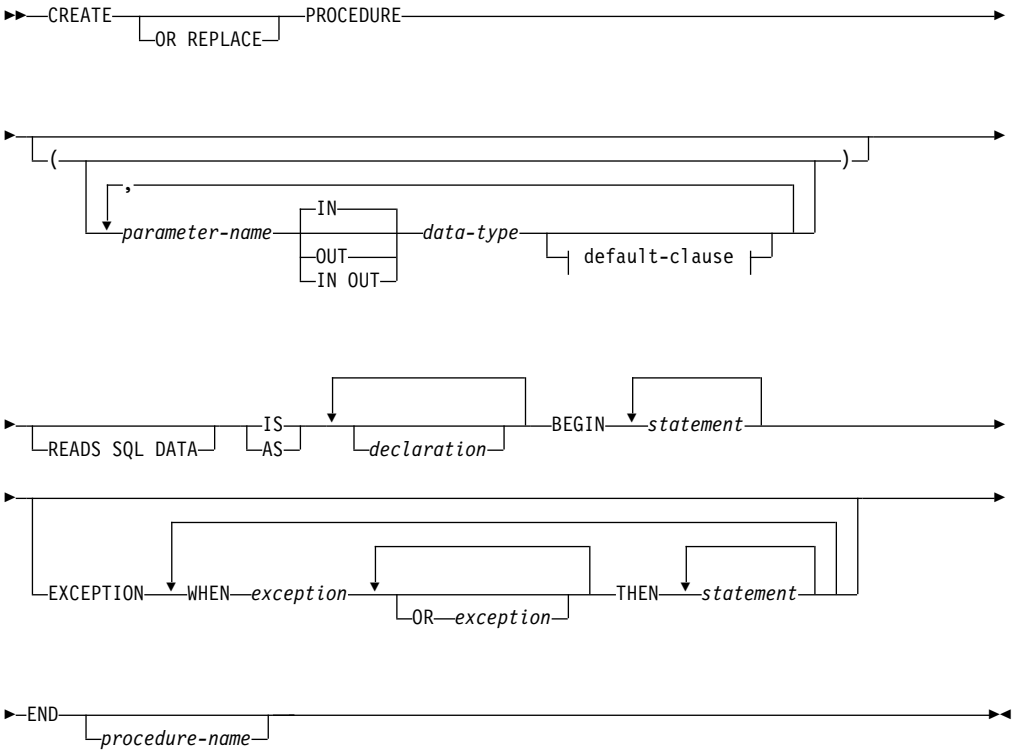
The privileges held by the authorization ID of the statement must include at least one of the following:

- If the schema name of the procedure does not exist, IMPLICIT_SCHEMA authority on the database
- If the schema name of the procedure refers to an existing schema, CREATEIN privilege on the schema
- DBADM authority

The privileges held by the authorization ID of the statement must also include all of the privileges necessary to invoke the SQL statements that are specified in the procedure body.

The authorization ID of the statement must be the owner of the matched procedure if OR REPLACE is specified (SQLSTATE 42501).

Syntax



Description

PROCEDURE *procedure-name*
Specifies an identifier for the procedure. The unqualified form of *procedure-name* is an SQL identifier with a maximum length of 128. In dynamic

SQL statements, the value of the CURRENT SCHEMA special register is used to qualify an unqualified object name. In static SQL statements, the QUALIFIER precompile or bind option implicitly specifies the qualifier for unqualified object names. The qualified form of *procedure-name* is a schema name followed by a period character and an SQL identifier. If a two-part name is specified, the schema name cannot begin with 'SYS'; otherwise, an error is returned (SQLSTATE 42939).

The name (including an implicit or explicit qualifier), together with the number of parameters, must not identify a procedure that is described in the catalog (SQLSTATE 42723). The unqualified name, together with the number of parameters, is unique within its schema, but does not need to be unique across schemas.

parameter-name
Specifies the name of a parameter. The parameter name must be unique for this procedure (SQLSTATE 42734).

data-type
Specifies one of the supported PL/SQL data types.

READS SQL DATA
Indicates that SQL statements that do not modify SQL data can be included in the procedure. This clause is a Db2 data server extension.

IS or AS
Introduces the procedure body definition.

declaration
Specifies one or more variable, cursor, or REF CURSOR type declarations.

BEGIN
Introduces the executable block. The BEGIN-END block can contain an EXCEPTION section.

statement
Specifies a PL/SQL or SQL statement. The statement must be terminated by a semicolon.

EXCEPTION
An optional keyword that introduces the exception section.

WHEN *exception-condition*
Specifies a conditional expression that tests for one or more types of exceptions.

statement
Specifies a PL/SQL or SQL statement. The statement must be terminated by a semicolon.

END
A mandatory keyword that ends the block. You can optionally specify the name of the procedure.

Notes

The CREATE PROCEDURE statement can be submitted in obfuscated form. In an obfuscated statement, only the procedure name is readable. The rest of the statement is encoded in such a way that it is not readable, but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

Examples

The following example shows a simple procedure that takes no parameters:

```
CREATE OR REPLACE PROCEDURE simple_procedure
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('That''s all folks!');
END simple_procedure;
```

The following example shows a procedure that takes an IN and an OUT parameter, and that has GOTO statements whose labels are of the standard PL/SQL form (**<<label>>**):

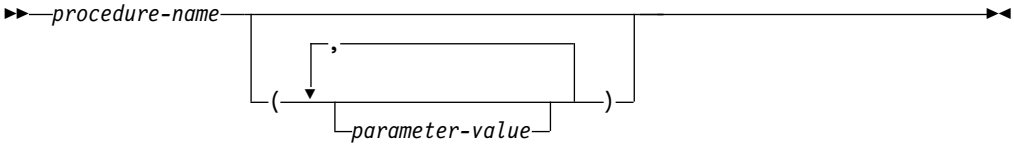
```
CREATE OR REPLACE PROCEDURE test_goto
( p1 IN INTEGER, out1 OUT VARCHAR2(30) )
IS
BEGIN
 <<LABEL2ABOVE>>
 IF p1 = 1 THEN
  out1 := out1 || 'one';
  GOTO LABEL1BELOW;
 END IF;
 if out1 IS NULL THEN
  out1 := out1 || 'two';
  GOTO LABEL2ABOVE;
 END IF;

 out1 := out1 || 'three';

 <<LABEL1BELOW>>
 out1 := out1 || 'four';

END test_goto;
```

Procedure references (PL/SQL)

Invocation references to PL/SQL procedures within PL/SQL contexts can be compiled by the Db2 data server.

A valid PL/SQL procedure reference consists of the procedure name followed by its parameters, if any.

Syntax



Description

procedure-name
Specifies an identifier for the procedure.

parameter-value
Specifies a parameter value. If no parameters are to be passed, the procedure can be called either with or without parentheses.

Example

The following example shows how to call a PL/SQL procedure within a PL/SQL context:

```
BEGIN
    simple_procedure;
END;
```

After a PL/SQL procedure has been created in a Db2 database, it can also be called using the CALL statement, which is supported inDb2 SQL contexts and applications using supported Db2 application programming interfaces.

Function invocation syntax support (PL/SQL)

A number of procedures support function invocation syntax in a PL/SQL assignment statement.

These procedures include:

- DBMS_SQL.EXECUTE
- DBMS_SQL.EXECUTE_AND_FETCH
- DBMS_SQL.FETCH_ROWS
- DBMS_SQL.IS_OPEN
- DBMS_SQL.LAST_ROW_COUNT
- DBMS_SQL.OPEN_CURSOR
- UTL_SMTP.CLOSE_DATA
- UTL_SMTP.COMMAND
- UTL_SMTP.COMMAND_REPLIES
- UTL_SMTP.DATA
- UTL_SMTP.EHLO
- UTL_SMTP.HELO
- UTL_SMTP.HELP
- UTL_SMTP.MAIL
- UTL_SMTP.NOOP
- UTL_SMTP.OPEN_DATA
- UTL_SMTP.QUIT
- UTL_SMTP.RCPT
- UTL_SMTP.RSET
- UTL_SMTP.VRFY

Examples

```
DECLARE
  cursor1 NUMBER;
  rowsProcessed NUMBER;
BEGIN
  cursor1 := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(cursor1, 'INSERT INTO T1 VALUES (10)', DBMS_SQL.NATIVE);
  rowsProcessed := DBMS_SQL.EXECUTE(cursor1);
  DBMS_SQL.CLOSE_CURSOR(cursor1);
END;
/
```

```
DECLARE
  v_connection UTL_SMTP.CONNECTION;
  v_reply UTL_SMTP.REPLY;
```

```
BEGIN
  UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 25, v_connection, 10, v_reply);
  UTL_SMTP.HELO(v_connection,'127.0.0.1');
  UTL_SMTP.MAIL(v_connection, 'sender1@ca.ibm.com');
  UTL_SMTP.RCPT(v_connection, 'receiver1@ca.ibm.com');
  v_reply := UTL_SMTP.OPEN_DATA (v_connection);
  UTL_SMTP.WRITE_DATA (v_connection, 'Test message');
  UTL_SMTP.CLOSE_DATA (v_connection);
  UTL_SMTP.QUIT(v_connection);
END;
/
```

Functions (PL/SQL)

The Db2 data server supports the compilation and execution of scalar and pipelined PL/SQL functions. Scalar PL/SQL functions can be invoked in contexts where expressions are valid. When evaluated, a scalar PL/SQL function returns a value that is substituted within the expression in which the function is embedded. Pipelined PL/SQL functions can be invoked in the FROM clause of SELECT statements and compute a table one row at a time.

PL/SQL functions are created by executing the CREATE FUNCTION statement. Such functions can be dropped from the database by using the Db2 SQL DROP statement. If you want to replace the implementation for a function, you do not need to drop it. You can use the CREATE FUNCTION statement and specify the OR REPLACE option to replace the function implementation.

CREATE FUNCTION statement (PL/SQL)

The CREATE FUNCTION statement defines a scalar or pipelined function that is stored in the database.

Invocation

A scalar function returns a single value each time it is invoked, and is generally valid wherever an SQL expression is valid. A pipelined function computes a table one row at a time and can be referenced in the FROM clause of SELECT statements.

This statement can be executed from the Db2 command line processor, any supported interactive SQL interface, an application, or routine.

Authorization

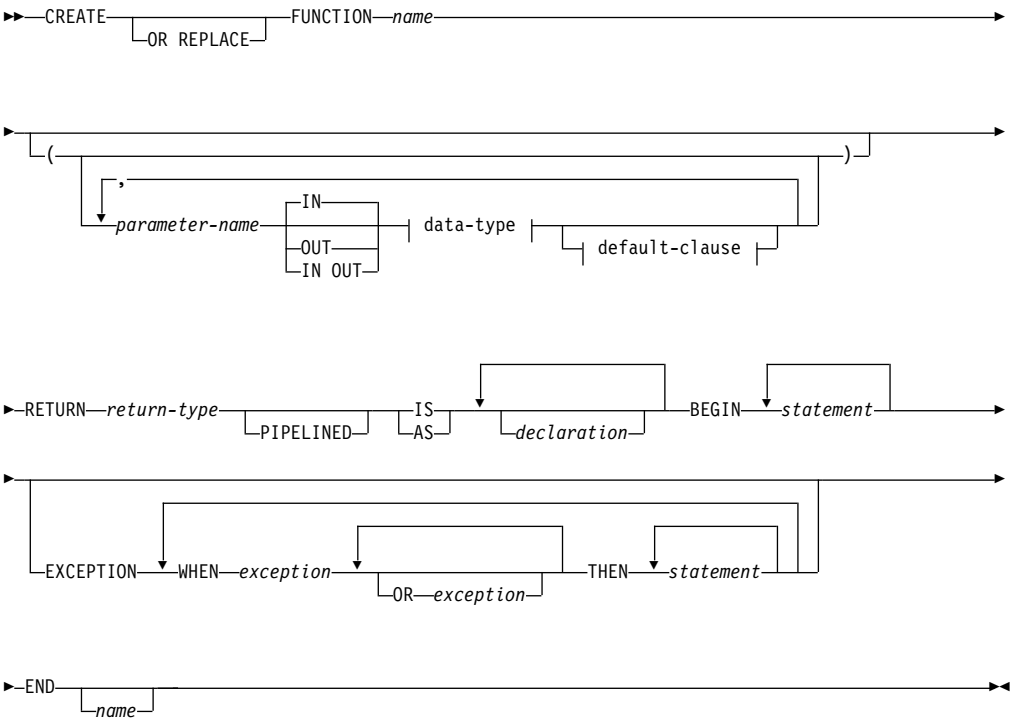
The privileges held by the authorization ID of the statement must include at least one of the following:

- If the schema name of the function does not exist, IMPLICIT_SCHEMA authority on the database
- If the schema name of the function refers to an existing schema, CREATEIN privilege on the schema
- DBADM authority

The privileges held by the authorization ID of the statement must also include all of the privileges necessary to invoke the SQL statements that are specified in the function body.

The authorization ID of the statement must be the owner of the matched function if OR REPLACE is specified (SQLSTATE 42501).

Syntax



Description

The CREATE FUNCTION statement specifies the name of the function, the optional parameters, the return type of the function, and the body of the function. The body of the function is a block that is enclosed by the BEGIN and END keywords. It can contain an optional EXCEPTION section that defines an action to be taken when a defined exception condition occurs.

OR REPLACE
Indicates that if a function with the same name already exists in the schema, the new function is to replace the existing one. If this option is not specified, the new function cannot replace an existing one with the same name in the same schema.

FUNCTION *name*
Specifies an identifier for the function.

parameter-name
Specifies the name of a parameter. The name cannot be the same as any other parameter-name in the parameter list (SQLSTATE 42734).

data-type
Specifies one of the supported PL/SQL data types.

RETURN *return-type*
Specifies the data type of the scalar value that is returned by the function.

PIPELINED
Specifies that the function being created is a pipelined function.

IS or AS
Introduces the block that defines the function body.

declaration
Specifies one or more variable, cursor, or REF CURSOR type declarations.

statement
Specifies one or more PL/SQL program statements. Each statement must be terminated by a semicolon.

exception
Specifies an exception condition name.

Notes

A PL/SQL function cannot take any action that changes the state of an object that the database manager does not manage.

The CREATE FUNCTION statement can be submitted in obfuscated form. In an obfuscated statement, only the function name is readable. The rest of the statement is encoded in such a way that it is not readable, but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

Examples

The following example shows a basic function that takes no parameters:

```
CREATE OR REPLACE FUNCTION simple_function
    RETURN VARCHAR2
IS
BEGIN
    RETURN 'That''s All Folks!';
END simple_function;
```

The following example shows a function that takes two input parameters:

```
CREATE OR REPLACE FUNCTION emp_comp (
     p_sal           NUMBER,
     p_comm          NUMBER )
RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp;
```

The following example shows a pipelined function that returns a table with two rows:

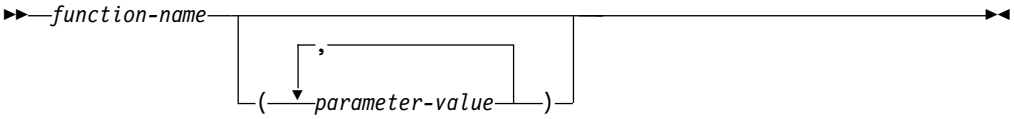
```
CREATE TYPE row_typ as OBJECT(f1 NUMBER, f2 VARCHAR2(10))

CREATE TYPE arr_typ as TABLE OF row_typ

CREATE FUNCTION pipe_func
RETURN arr_typ
PIPELINED
IS
BEGIN
```

```
    PIPE ROW (1, 'one');
    PIPE ROW (2, 'two');
    RETURN;
END pipe_func;
```

Function references (PL/SQL)

Scalar PL/SQL functions can be referenced wherever an expression is supported. Pipelined PL/SQL functions can be referenced in the FROM clause of SELECT statements.

Syntax



Description

function-name
Specifies an identifier for the function.

parameter-value
Specifies a value for a parameter.

Examples

The following example shows how a function named SIMPLE_FUNCTION, defined in the PL/SQL sample schema, can be called from a PL/SQL anonymous block:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(simple_function);
END;
```

The following example shows how a scalar function and a pipelined function can be used within an SQL statement:

```
SELECT
   emp.empno, emp.ename, emp.sal, emp.comm,
   emp_comp(sal, comm)+bon.bonus "YEARLY COMPENSATION"
FROM emp, TABLE(bonuses()) as bon(category, bonus)
WHERE bon.category = emp.category
```

Collection, record, and object types (PL/SQL)

The use of PL/SQL collections is supported by the Db2 data server. A PL/SQL *collection* is a set of ordered data elements with the same data type. Individual data items in the set can be referenced by using subscript notation within parentheses.

In PL/SQL contexts, the Db2 server supports varrays, associative arrays, and record types. The Db2 server accepts the syntax for the creation of PL/SQL nested tables and object types, but maps nested tables to associative arrays and object types to records.

VARRAY collection type declaration (PL/SQL)

A VARRAY is a type of collection in which each element is referenced by a positive integer called the *array index*. The maximum cardinality of the VARRAY is specified in the type definition.

The TYPE IS VARRAY statement is used to define a VARRAY collection type.

Syntax

Description

varraytype
An identifier that is assigned to the array type.

n The maximum number of elements in the array type.

datatype
A supported data type, such as NUMBER, VARCHAR2, RECORD, VARRAY, or associative array type. The %TYPE attribute and the %ROWTYPE attribute are also supported.

Example

The following example reads employee names from the EMP table, stores the names in an array variable of type VARRAY, and then displays the results. The EMP table contains one column named ENAME. The code is executed from a Db2 script (`script.db2`). The following commands should be issued from the Db2 command window before executing the script (`db2 -tvf script.db2`):

```
db2set DB2_COMPATIBILITY_VECTOR=FFF
db2stop
db2start
```

The script contains the following code:

```
SET SQLCOMPAT PLSQL;

connect to mydb
/

CREATE PACKAGE foo
AS
    TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);
END;
/

SET SERVEROUTPUT ON
/

DECLARE
    emp_arr         foo.emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 5;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
```

```
    END LOOP;
END;
/

DROP PACKAGE foo
/

connect reset
/
```

This script produces the following sample output:

```
Curly
Larry
Moe
Shemp
Joe
```

CREATE TYPE (VARRAY) statement (PL/SQL)

The CREATE TYPE (VARRAY) statement defines a VARRAY data type.

Invocation

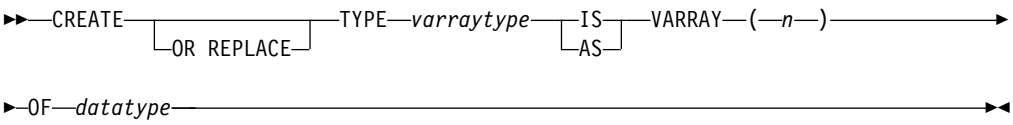
This statement can be executed from the Db2 command line processor (CLP), any supported interactive SQL interface, an application, or a routine.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- If the schema name of the VARRAY type does not exist, IMPLICIT_SCHEMA authority on the database
- If the schema name of the VARRAY type refers to an existing schema, CREATEIN privilege on the schema
- DBADM authority

Syntax



Description

OR REPLACE
Indicates that if a user-defined data type with the same name already exists in the schema, the new data type is to replace the existing one. If this option is not specified, the new data type cannot replace an existing one with the same name in the same schema.

varraytype
Specifies an identifier for the VARRAY type. The unqualified form of *varraytype* is an SQL identifier with a maximum length of 128. The value of the CURRENT SCHEMA special register is used to qualify an unqualified object name. The qualified form of *varraytype* is a schema name followed by a period character and an SQL identifier. If a two-part name is specified, the schema name cannot begin with 'SYS'; otherwise, an error is returned (SQLSTATE

42939). The name (including an implicit or explicit qualifier) must not identify a user-defined data type that is described in the catalog (SQLSTATE 42723). The unqualified name is unique within its schema, but does not need to be unique across schemas.

n Specifies the maximum number of elements in the array type. The maximum cardinality of an array on a given system is limited by the total amount of memory that is available to Db2 applications. As such, although arrays of large cardinalities (up to 2,147,483,647) can be created, not all elements might be available for use.

datatype
A supported data type, such as NUMBER, VARCHAR2, RECORD, VARRAY, or associative array type. The %TYPE attribute and the %ROWTYPE attribute are also supported.

Example

The following example creates a VARRAY data type with a maximum of 10 elements, where each element has the data type NUMBER:

```
CREATE TYPE NUMARRAY1 AS VARRAY (10) OF NUMBER
```

CREATE TYPE (Nested table) statement (PL/SQL)

The CREATE TYPE (Nested table) statement defines an associative array indexed by INTEGER data type.

Invocation

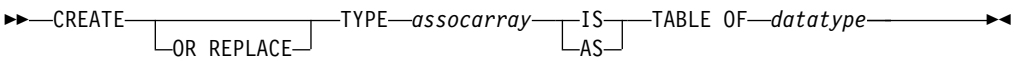
This statement can be executed from the Db2 command line processor (CLP), any supported interactive SQL interface, an application, or a routine.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- If the schema name of the nested table type does not exist, IMPLICIT_SCHEMA authority on the database
- If the schema name of the nested table type refers to an existing schema, CREATEIN privilege on the schema
- DBADM authority

Syntax



Description

OR REPLACE
Indicates that if a user-defined data type with the same name already exists in the schema, the new data type is to replace the existing one. If this option is not specified, the new data type cannot replace an existing one with the same name in the same schema.

assocarray
Specifies an identifier for the associative array type.

datatype
Specifies a supported data type, such as NUMBER, VARCHAR2, RECORD, VARRAY, or associative array type.

Example

The following example reads the first ten employee names from the EMP table, stores them in a nested table, and then displays its contents:

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE TYPE emp_arr_typ IS TABLE OF VARCHAR2(10)
/

DECLARE
   emp_arr         emp_arr_typ;
   CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
   i               INTEGER := 0;
BEGIN
   FOR r_emp IN emp_cur LOOP
      i := i + 1;
      emp_arr(i) := r_emp.ename;
   END LOOP;
   FOR j IN 1..10 LOOP
      DBMS_OUTPUT.PUT_LINE(emp_arr(j));
   END LOOP;
END
/
```

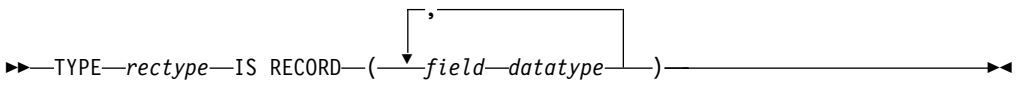
Records (PL/SQL)

A record type is a composite data type that consists of one or more identifiers and their corresponding data types.

You can create user-defined record types by using the TYPE IS RECORD statement within a package or by using the CREATE TYPE (Object) statement.

Dot notation is used to reference fields in a record. For example, `record.field`.

Syntax



Description

TYPE *rectype* **IS RECORD**
Specifies an identifier for the record type.

field
Specifies an identifier for a field of the record type.

datatype
Specifies the corresponding data type of the *field*. The %TYPE attribute, RECORD, VARRAY, associative array types, and the %ROWTYPE attributes are supported.

Example

The following example shows a package that references a user-defined record type:

```
CREATE OR REPLACE PACKAGE pkg7a
IS
TYPE t1_typ IS RECORD (
  c1 T1.C1%TYPE,
  c2 VARCHAR(10)
);
END;
```

CREATE TYPE (Object) statement (PL/SQL)

The CREATE TYPE (Object) statement defines a record data type.

Invocation

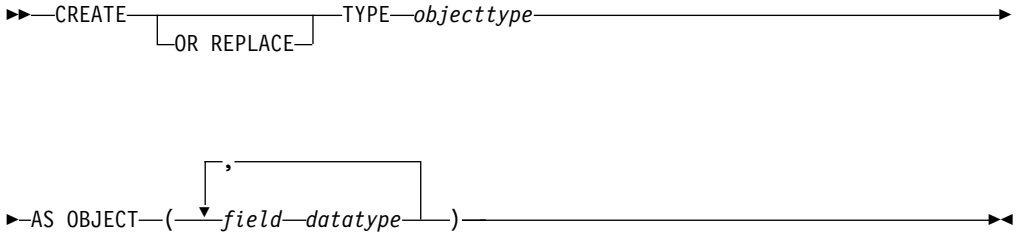
This statement can be executed from the Db2 command line processor (CLP), any supported interactive SQL interface, an application, or a routine.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- If the schema name of the object type does not exist, IMPLICIT_SCHEMA authority on the database
- If the schema name of the object type refers to an existing schema, CREATEIN privilege on the schema
- DBADM authority

Syntax



Description

OR REPLACE
Indicates that if a user-defined data type with the same name already exists in the schema, the new data type is to replace the existing one. If this option is not specified, the new data type cannot replace an existing one with the same name in the same schema.

objecttype
Specifies an identifier for the record type.

field
Specifies an identifier for a field of the record type.

datatype
Specifies a supported data type, such as NUMBER, VARCHAR2, RECORD, VARRAY, or associative array type.

Example

The following example shows a the definition of a record type with two fields:

```
CREATE TYPE objtyp AS OBJECT
(
   c1    NUMBER,
   c2    VARCHAR2(10)
);
```

Associative arrays (PL/SQL)

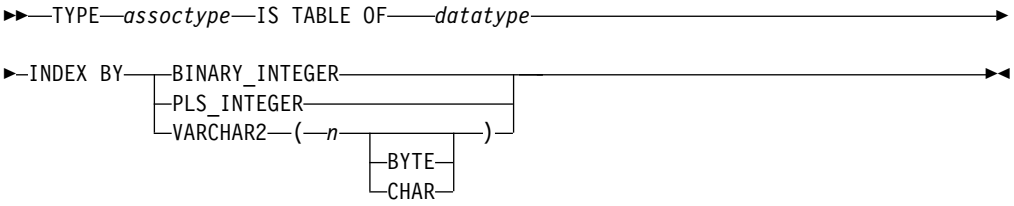
A PL/SQL associative array is a collection type that associates a unique key with a value.

An associative array has the following characteristics:

- An associative array type must be defined before array variables of that array type can be declared. Data manipulation occurs in the array variable.
- The array does not need to be initialized; simply assign values to array elements.
- There is no defined limit on the number of elements in the array; it grows dynamically as elements are added.
- The array can be *sparse*; there can be gaps in the assignment of values to keys.
- An attempt to reference an array element that has not been assigned a value results in an exception.

Use the TYPE IS TABLE OF statement to define an associative array type.

Syntax



Description

TYPE *assoctype*
Specifies an identifer for the array type.

datatype
Specifies a supported data type, such as VARCHAR2, NUMBER, RECORD, VARRAY, or associative array type. The %TYPE attribute and the %ROWTYPE attribute are also supported.

INDEX BY
Specifies that the associative array is to be indexed by one of the data types introduced by this clause.

BINARY INTEGER
Integer numeric data.

PLS_INTEGER
Integer numeric data.

VARCHAR2 (*n***[BYTE|CHAR])**
A variable-length character string of maximum length *n* code units, which

may range from 1 to 32 672 BYTE or from 1 to 8 168 CHAR. The %TYPE attribute is also supported if the object to which the %TYPE attribute is being applied is of the BINARY_INTEGER, PLS_INTEGER, or VARCHAR2 data type.

To declare a variable with an associative array type, specify *array-name assoctype*, where *array-name* represents an identifier that is assigned to the associative array, and *assoctype* represents the identifier for a previously declared array type.

To reference a particular element of the array, specify *array-name*(*n*), where *array-name* represents the identifier for a previously declared array, and *n* represents a value of INDEX BY data type of *assoctype*. If the array is defined from a record type, the reference becomes *array-name*(*n*).*field*, where *field* is defined within the record type from which the array type is defined. To reference the entire record, omit *field*.

Examples

The following example reads the first ten employee names from the EMP table, stores them in an array, and then displays the contents of the array.

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type1
IS
    TYPE emp_arr_typ IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
END pkg_test_type1
/

DECLARE
    emp_arr         pkg_test_type1.emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END
/
```

This code generates the following sample output:

```
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
```

The example can be modified to use a record type in the array definition.

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type2
```

```
IS
    TYPE emp_rec_typ IS RECORD (
        empno       INTEGER,
        ename       VARCHAR2(10)
    );
END pkg_test_type2
/

CREATE OR REPLACE PACKAGE pkg_test_type3
IS
    TYPE emp_arr_typ IS TABLE OF pkg_test_type2.emp_rec_typ INDEX BY BINARY_INTEGER;
END pkg_test_type3
/

DECLARE
    emp_arr         pkg_test_type3.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

The modified code generates the following sample output:

```
EMPNO    ENAME
-----    -------
1001     SMITH
1002     ALLEN
1003     WARD
1004     JONES
1005     MARTIN
1006     BLAKE
1007     CLARK
1008     SCOTT
1009     KING
1010     TURNER
```

This example can be further modified to use the emp%ROWTYPE attribute to define `emp_arr_typ`, instead of using the `emp_rec_typ` record type.

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type4
IS
    TYPE emp_arr_typ IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
END pkg_test_type4
/

DECLARE
    emp_arr         pkg_test_type4.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
```

```
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

In this case, the sample output is identical to that of the previous example.

Finally, instead of assigning each field of the record individually, a record-level assignment can be made from `r_emp` to `emp_arr`:

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type5
IS
    TYPE emp_rec_typ IS RECORD (
        empno       INTEGER,
        ename       VARCHAR2(10)
    );
END pkg_test_type5
/

CREATE OR REPLACE PACKAGE pkg_test_type6
IS
    TYPE emp_arr_typ IS TABLE OF pkg_test_type5.emp_rec_typ INDEX BY BINARY_INTEGER;
END pkg_test_type6
/

DECLARE
    emp_arr         pkg_test_type6.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

Collection methods (PL/SQL)

Collection methods can be used to obtain information about collections or to modify collections.

The following commands should be executed before attempting to run the examples in Table 1 on page 32.

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
db2 connect to mydb
```

The MYDB database has one table, EMP, which has one column, ENAME (defined as VARCHAR(10)):

```
db2 select * from emp

ENAME
----------
Curly
Larry
Moe
Shemp
Joe

  5 record(s) selected.
```

Table 1. Collection methods that are supported (or tolerated) by the Db2 data server in a PL/SQL context

Collection method	Description	Example

COUNT	Returns the number of elements in a collection.	`CREATE PACKAGE foo`
		`AS`
		`  TYPE sparse_arr_typ IS TABLE OF NUMBER`
		`  INDEX BY BINARY_INTEGER;`
		`END;`
		`/`
		
		`SET SERVEROUTPUT ON`
		`/`
		
		`DECLARE`
		`  sparse_arr  foo.sparse_arr_typ;`
		`BEGIN`
		`  sparse_arr(-10)  := -10;`
		`  sparse_arr(0)  := 0;`
		`  sparse_arr(10)  := 10;`
		`  DBMS_OUTPUT.PUT_LINE('COUNT: ' ||`
		`  sparse_arr.COUNT);`
		`END;`
		`/`

Table 1. Collection methods that are supported (or tolerated) by the Db2 data server in a PL/SQL context (continued)

Collection method	Description	Example

DELETE	Removes all elements from a collection.	`CREATE PACKAGE foo`
		`AS`
		`  TYPE names_typ IS TABLE OF VARCHAR2(10)`
		`  INDEX BY BINARY_INTEGER;`
		`END;`
		`/`
		
		`SET SERVEROUTPUT ON`
		`/`
		
		`DECLARE`
		`  actor_names  foo.names_typ;`
		
		`BEGIN`
		
		`  actor_names(1) := 'Chris';`
		`  actor_names(2) := 'Steve';`
		`  actor_names(3) := 'Kate';`
		`  actor_names(4) := 'Naomi';`
		`  actor_names(5) := 'Peter';`
		`  actor_names(6) := 'Philip';`
		`  actor_names(7) := 'Michael';`
		`  actor_names(8) := 'Gary';`
		
		`  DBMS_OUTPUT.PUT_LINE('COUNT: ' ||`
		`  actor_names.COUNT);`
		
		`  actor_names.DELETE(2);`
		`  DBMS_OUTPUT.PUT_LINE('COUNT: ' ||`
		`  actor_names.COUNT);`
		
		`  actor_names.DELETE(3, 5);`
		`  DBMS_OUTPUT.PUT_LINE('COUNT: ' ||`
		`  actor_names.COUNT);`
		
		`  actor_names.DELETE;`
		`  DBMS_OUTPUT.PUT_LINE('COUNT: ' ||`
		`  actor_names.COUNT);`
		
		`END;`
		`/`

DELETE (*n*)	Removes element *n* from an associative array. You cannot delete individual elements from a VARRAY collection type.	See “DELETE”.
DELETE (*n1, n2*)	Removes all elements from *n1* to *n2* from an associative array. You cannot delete individual elements from a VARRAY collection type.	See “DELETE”.

Table 1. Collection methods that are supported (or tolerated) by the Db2 data server in a PL/SQL context (continued)

Collection method	Description	Example

EXISTS (*n*)	Returns TRUE if the specified element exists.	`CREATE PACKAGE foo`
		`AS`
		`  TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);`
		`END;`
		`/`
		
		`SET SERVEROUTPUT ON`
		`/`
		
		`DECLARE`
		`  emp_arr  foo.emp_arr_typ;`
		`  CURSOR emp_cur IS SELECT ename FROM emp`
		`  WHERE ROWNUM <= 5;`
		`  i  INTEGER := 0;`
		`BEGIN`
		`  FOR r_emp IN emp_cur LOOP`
		`  i := i + 1;`
		`  emp_arr.EXTEND;`
		`  emp_arr(i) := r_emp.ename;`
		`  END LOOP;`
		`  emp_arr.TRIM;`
		`  FOR j IN 1..5 LOOP`
		`  IF emp_arr.EXISTS(j) = true THEN`
		`  DBMS_OUTPUT.PUT_LINE(emp_arr(j));`
		`  ELSE`
		`  DBMS_OUTPUT.PUT_LINE('THIS ELEMENT`
		`  HAS BEEN DELETED');`
		`  END IF;`
		`  END LOOP;`
		`END;`
		`/`

EXTEND	Appends a single NULL element to a collection.	See "EXISTS (*n*)".
EXTEND (*n*)	Appends *n* NULL elements to a collection.	See "EXISTS (*n*)".
EXTEND (*n1, n2*)	Appends *n1* copies of the $n2^{th}$ element to a collection.	See "EXISTS (*n*)".

Table 1. Collection methods that are supported (or tolerated) by the Db2 data server in a PL/SQL context (continued)

Collection method	Description	Example

FIRST	Returns the smallest index number in a collection.	CREATE PACKAGE foo
		AS
		  TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);
		END;
		/
		
		SET SERVEROUTPUT ON
		/
		
		DECLARE
		  emp_arr foo.emp_arr_typ;
		  CURSOR emp_cur IS SELECT ename FROM emp
		  WHERE ROWNUM <= 5;
		  i INTEGER := 0;
		  k INTEGER := 0;
		  l INTEGER := 0;
		BEGIN
		
		  FOR r_emp IN emp_cur LOOP
		  i := i + 1;
		  emp_arr(i) := r_emp.ename;
		  END LOOP;
		
		  -- Use FIRST and LAST to specify the lower and
		  -- upper bounds of a loop range:
		  FOR j IN emp_arr.FIRST..emp_arr.LAST LOOP
		  DBMS_OUTPUT.PUT_LINE(emp_arr(j));
		  END LOOP;
		
		  -- Use NEXT(n) to obtain the subscript of
		  -- the next element:
		  k := emp_arr.FIRST;
		  WHILE k IS NOT NULL LOOP
		  DBMS_OUTPUT.PUT_LINE(emp_arr(k));
		  k := emp_arr.NEXT(k);
		  END LOOP;
		
		  -- Use PRIOR(n) to obtain the subscript of
		  -- the previous element:
		  l := emp_arr.LAST;
		  WHILE l IS NOT NULL LOOP
		  DBMS_OUTPUT.PUT_LINE(emp_arr(l));
		  l := emp_arr.PRIOR(l);
		  END LOOP;
		
		  DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);
		
		  emp_arr.TRIM;
		  DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);
		
		  emp_arr.TRIM(2);
		  DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);
		
		  DBMS_OUTPUT.PUT_LINE('Max. no. elements = ' ||
		  emp_arr.LIMIT);
		
		END;
		/

LAST	Returns the largest index number in a collection.	See “FIRST”.

Table 1. Collection methods that are supported (or tolerated) by the Db2 data server in a PL/SQL context (continued)

Collection method	Description	Example
LIMIT	Returns the maximum number of elements for a VARRAY, or NULL for nested tables.	See "FIRST".
NEXT (*n*)	Returns the index number of the element immediately following the specified index.	See "FIRST".
PRIOR (*n*)	Returns the index number of the element immediately prior to the specified index.	See "FIRST".
TRIM	Removes a single element from the end of a collection. You cannot trim elements from an associative array collection type.	See "FIRST".
TRIM (*n*)	Removes *n* elements from the end of a collection. You cannot trim elements from an associative array collection type.	See "FIRST".

Variables (PL/SQL)

Variables must be declared before they are referenced.

Variables that are used in a block must generally be defined in the declaration section of the block unless they are global variables or package-level variables. The declaration section contains the definitions of variables, cursors, and other types that can be used in PL/SQL statements within the block. A variable declaration consists of a name that is assigned to the variable and the data type of the variable. Optionally, the variable can be initialized to a default value within the variable declaration.

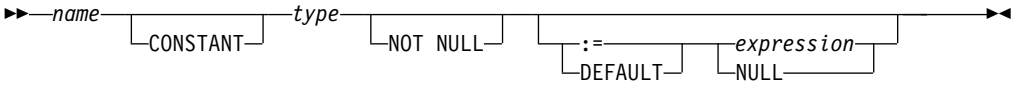
Procedures and functions can have parameters for passing input values. Procedures can also have parameters for passing output values, or parameters for passing both input and output values.

PL/SQL also includes variable data types to match the data types of existing columns, rows, or cursors using the %TYPE and %ROWTYPE qualifiers.

Variable declarations (PL/SQL)

Variables that are used in a block must generally be defined in the declaration section of the block unless they are global variables or package-level variables. A variable declaration consists of a name that is assigned to the variable and the data type of the variable. Optionally, the variable can be initialized to a default value within the variable declaration.

Syntax



Description

name
Specifies an identifier that is assigned to the variable.

CONSTANT
Specifies that the variable value is constant. A default expression must be assigned, and a new value cannot be assigned to the variable within the application program.

type
Specifies a data type for the variable.

NOT NULL
Currently ignored by Db2. Routines that specify NOT NULL for variable declarations compile successfully. However, such routines behave as though NOT NULL has not been specified. No run-time checking is performed to disallow null values in variables declared NOT NULL. See the following example, if your application needs to restrict null values in PL/SQL variables.

DEFAULT
Specifies a default value for the variable. This default is evaluated every time that the block is entered. For example, if SYSDATE has been assigned to a variable of type DATE, the variable resolves to the current invocation time, not to the time at which the procedure or function was precompiled.

:= The assignment operator is a synonym for the DEFAULT keyword. However, if this operator is specified without *expression*, the variable is initialized to the value NULL.

expression
Specifies the initial value that is to be assigned to the variable when the block is entered.

NULL
Specifies the SQL value NULL, which has a null value.

Examples

1. The following procedure shows variable declarations that utilize defaults consisting of string and numeric expressions:

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno        NUMBER
)
IS
    todays_date     DATE := SYSDATE;
    rpt_title       VARCHAR2(60) := 'Report For Department # ' || p_deptno
```

```
            || ' on ' || todays_date;
    base_sal        INTEGER := 35525;
    base_comm_rate  NUMBER := 1.33333;
    base_annual     NUMBER := ROUND(base_sal * base_comm_rate, 2);
BEGIN
    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

The following sample output was obtained by calling this procedure:

```
CALL dept_salary_rpt(20);

Report For Department # 20 on 10-JUL-07 16:44:45
Base Annual Salary: 47366.55
```

2. The following example restricts null values by adding explicit checks using IS NULL or IS NOT NULL and handles error cases as required:

```
create table T(col1 integer);
insert into T values null;

declare
  N integer not null := 0;
  null_variable exception;
begin
  select col1 into N from T;
  if N is null then
    raise null_variable;
  end if;
exception
  when null_variable then
    -- Handle error condition here.
    dbms_output.put_line('Null variable detected');
end;
```

Parameter modes (PL/SQL)

PL/SQL procedure parameters can have one of three possible modes: IN, OUT, or IN OUT. PL/SQL function parameters can only be IN.

- An IN formal parameter is initialized to the actual parameter with which it was called, unless it was explicitly initialized with a default value. The IN parameter can be referenced within the called program; however, the called program cannot assign a new value to the IN parameter. After control returns to the calling program, the actual parameter always contains the value to which it was set prior to the call.
- An OUT formal parameter is initialized to the actual parameter with which it was called. The called program can reference and assign new values to the formal parameter. If the called program terminates without an exception, the actual parameter takes on the value to which the formal parameter was last set. If a handled exception occurs, the actual parameter takes on the last value to which the formal parameter was set. If an unhandled exception occurs, the value of the actual parameter remains what it was prior to the call.
- Like an IN parameter, an IN OUT formal parameter is initialized to the actual parameter with which it was called. Like an OUT parameter, an IN OUT formal parameter is modifiable by the called program, and the last value of the formal parameter is passed to the calling program's actual parameter if the called program terminates without an exception. If a handled exception occurs, the actual parameter takes on the last value to which the formal parameter was set. If an unhandled exception occurs, the value of the actual parameter remains what it was prior to the call.

Table 2 summarizes this behavior.

Table 2. Parameter modes

Mode property	IN	IN OUT	OUT
Formal parameter initialized to:	Actual parameter value	Actual parameter value	Actual parameter value
Formal parameter modifiable by the called program?	No	Yes	Yes
After normal termination of the called program, actual parameter contains:	Original actual parameter value prior to the call	Last value of the formal parameter	Last value of the formal parameter
After a handled exception in the called program, actual parameter contains:	Original actual parameter value prior to the call	Last value of the formal parameter	Last value of the formal parameter
After an unhandled exception in the called program, actual parameter contains:	Original actual parameter value prior to the call	Original actual parameter value prior to the call	Original actual parameter value prior to the call

Data types (PL/SQL)

A wide range of data types are supported and can be used to declare variables in a PL/SQL block.

Table 3. Supported scalar data types that are available in PL/SQL

PL/SQL data type	Db2 SQL data type	Description
BINARY_INTEGER	INTEGER	Integer numeric data
BLOB	BLOB (4096)	Binary data
BLOB (*n*)	BLOB (*n*) *n* = 1 to 2 147 483 647	Binary large object data
BOOLEAN	BOOLEAN	Logical Boolean (true or false)
CHAR	CHAR (*n*) *n* = 63 if the string units of the environment is set to CODEUNITS32 *n* = 255 otherwise	Fixed-length character string data of length *n*
CHAR (*n*)	CHAR (*n*) *n* = 1 to 255	Fixed-length character string data of length *n*
CHAR (*n* CHAR)	CHAR (*n* CODEUNITS32) *n* = 1 to 63	Fixed-length character string data of length *n* UTF-32 code units[1]
CHAR VARYING (*n*)	VARCHAR (*n*)	Variable-length character string data of maximum length *n*
CHAR VARYING (*n* CHAR)	VARCHAR (*n* CODEUNITS32) *n* = 1 to 8 168	Variable-length character string data of maximum length n UTF-32 code units[1]

Table 3. Supported scalar data types that are available in PL/SQL (continued)

PL/SQL data type	Db2 SQL data type	Description
CHARACTER	CHARACTER (*n*) *n* = 63 if the string units of the environment is set to CODEUNITS32 *n* = 255 otherwise	Fixed-length character string data of length *n*
CHARACTER (*n*)	CHARACTER (*n*) *n* = 1 to 255	Fixed-length character string data of length *n*
CHARACTER (*n* CHAR)	CHARACTER (*n* CODEUNITS32) *n* = 1 to 63	Fixed-length character string data of length *n* UTF-32 code units[1]
CHARACTER VARYING (*n*)	VARCHAR (*n*) *n* = 1 to 32 672	Variable-length character string data of maximum length *n*
CHARACTER VARYING (*n* CHAR)	VARCHAR (*n* CODEUNITS32) *n* = 1 to 8 168	Variable-length character string data of maximum length *n* UTF-32 code units[1]
CLOB	CLOB (1M)	Character large object data
CLOB (*n*)	CLOB (*n*) *n* = 1 to 2 147 483 647	Character large object data of length *n*
CLOB (*n* CHAR)	CLOB (*n* CODEUNITS32) *n* = 1 to 536 870 911	Character large object string data of length *n* UTF-32 code units[1]
DATE	DATE [2]	Date and time data (expressed to the second)
DEC	DEC (9, 2)	Decimal numeric data
DEC (*p*)	DEC (*p*) *p* = 1 to 31	Decimal numeric data of precision *p*
DEC (*p*, *s*)	DEC (*p*, *s*) *p* = 1 to 31; *s* = 1 to 31	Decimal numeric data of precision *p* and scale *s*
DECIMAL	DECIMAL (9, 2)	Decimal numeric data
DECIMAL (*p*)	DECIMAL (*p*) *p* = 1 to 31	Decimal numeric data of precision *p*
DECIMAL (*p*, *s*)	DECIMAL (*p*, *s*) *p* = 1 to 31; *s* = 1 to 31	Decimal numeric data of precision *p* and scale *s*
DOUBLE	DOUBLE	Double precision floating-point number
DOUBLE PRECISION	DOUBLE PRECISION	Double precision floating-point number
FLOAT	FLOAT	Float numeric data
FLOAT (*n*) *n* = 1 to 24	REAL	Real numeric data
FLOAT (*n*) *n* = 25 to 53	DOUBLE	Double numeric data
INT	INT	Signed four-byte integer numeric data
INTEGER	INTEGER	Signed four-byte integer numeric data
LONG	CLOB (32760)	Character large object data
LONG RAW	BLOB (32760)	Binary large object data
LONG VARCHAR	CLOB (32760)	Character large object data

Table 3. Supported scalar data types that are available in PL/SQL (continued)

PL/SQL data type	Db2 SQL data type	Description
NATURAL	INTEGER	Signed four-byte integer numeric data
NCHAR	NCHAR (*n*) [3] *n* = 63 if the NCHAR_MAPPING configuration parameter is set to GRAPHIC_CU32 or CHAR_CU32 *n* = 127 otherwise	Fixed-length national character string data of length *n*
NCHAR (*n*) *n* = 1 to 2000	NCHAR (*n*) [3]	Fixed-length national character string data of length *n*
NCLOB [4]	NCLOB(1M) [3]	National character large object data
NCLOB (*n*)	NCLOB (*n*) [3]	National character large object data of maximum length *n*
NVARCHAR2	NVARCHAR [3]	Variable-length national character string data
NVARCHAR2 (*n*)	NVARCHAR (*n*) [3]	Variable-length national character string data of maximum length *n*
NUMBER	NUMBER [5]	Exact numeric data
NUMBER (*p*)	NUMBER (*p*) [5]	Exact numeric data of maximum precision *p*
NUMBER (*p*, *s*)	NUMBER (*p*, *s*) [5] *p* = 1 to 31; *s* = 1 to 31	Exact numeric data of maximum precision *p* and scale *s*
NUMERIC	NUMERIC (9.2)	Exact numeric data
NUMERIC (*p*)	NUMERIC (*p*) *p* = 1 to 31	Exact numeric data of maximum precision *p*
NUMERIC (*p*, s)	NUMERIC (*p*, *s*) *p* = 1 to 31; *s* = 0 to 31	Exact numeric data of maximum precision *p* and scale *s*
PLS_INTEGER	INTEGER	Integer numeric data
RAW	VARBINARY(32672)	Variable-length binary string data
RAW (*n*)	VARBINARY(*n*) *n* = 1 to 32 672	Variable-length binary string data
SMALLINT	SMALLINT	Signed two-byte integer data
TIMESTAMP (0)	TIMESTAMP (0)	Date data with timestamp information
TIMESTAMP (*p*)	TIMESTAMP (*p*)	Date and time data with optional fractional seconds and precision *p*
VARCHAR	VARCHAR (4096)	Variable-length character string data with a maximum length of 4096
VARCHAR (*n*)	VARCHAR (*n*)	Variable-length character string data with a maximum length of *n*
VARCHAR (*n* CHAR)	VARCHAR (*n* CODEUNITS32) *n* = 1 to 8 168	Variable-length character string data of maximum length *n* UTF-32 code units[1]
VARCHAR2 (*n*)	VARCHAR2 (*n*) [6]	Variable-length character string data with a maximum length of *n*

Table 3. Supported scalar data types that are available in PL/SQL (continued)

PL/SQL data type	Db2 SQL data type	Description
VARCHAR2 (*n* CHAR)	VARCHAR2 (*n* CODEUNITS32) *n* = 1 to 8 168 [6]	Variable-length character string data of maximum length *n* UTF-32 code units[1]

1. If the string units of the environment is set to CODEUNITS32, the CHAR attribute of the length is implicit. This behavior is similar to NLS_LENGTH_SEMANTICS=CHAR in an Oracle database.
2. When the DB2_COMPATIBILITY_VECTOR registry variable is set for the DATE data type, DATE is equivalent to TIMESTAMP (0).
3. National character strings are synonyms for character strings or graphic strings with the mapping of data types determined by the **NCHAR_MAPPING** configuration parameter. See "National character strings" for details.
4. For restrictions on the NCLOB data type in certain database environments, see "Restrictions on PL/SQL support".
5. This data type is supported when the **number_compat** database configuration parameter set to ON.
6. This data type is supported when the **varchar2_compat** database configuration parameter set to ON.

In addition to the scalar data types described in Table 3 on page 39, the Db2 data server also supports collection types, record types, and REF CURSOR types.

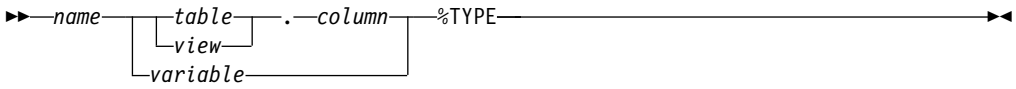
%TYPE attribute in variable declarations (PL/SQL)

The %TYPE attribute, used in PL/SQL variable and parameter declarations, is supported by the Db2 data server. Use of this attribute ensures that type compatibility between table columns and PL/SQL variables is maintained.

A qualified column name in dot notation or the name of a previously declared variable must be specified as a prefix to the %TYPE attribute. The data type of this column or variable is assigned to the variable being declared. If the data type of the column or variable changes, there is no need to modify the declaration code.

The %TYPE attribute can also be used with formal parameter declarations.

Syntax



Description

name
Specifies an identifier for the variable or formal parameter that is being declared.

table
Specifies an identifier for the table whose column is to be referenced.

view
Specifies an identifier for the view whose column is to be referenced.

column
Specifies an identifier for the table or view column that is to be referenced.

variable
Specifies an identifier for a previously declared variable that is to be referenced. The variable does not inherit any other column attributes, such as, for example, the nullability attribute.

Example

The following example shows a procedure that queries the EMP table using an employee number, displays the employee's data, finds the average salary of all employees in the department to which the employee belongs, and then compares the chosen employee's salary with the department average.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN NUMBER
)
IS
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_deptno        NUMBER(2);
    v_avgsal        NUMBER(7,2);
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

This procedure could be rewritten without explicitly coding the EMP table data types in the declaration section.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_avgsal        v_sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
```

```
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

The p_empno parameter is an example of a formal parameter that is defined using the %TYPE attribute. The v_avgsal variable is an example of the %TYPE attribute referring to another variable instead of a table column.

The following sample output is generated by a call to the EMP_SAL_QUERY procedure:

```
CALL emp_sal_query(7698);

Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
```

SUBTYPE definitions (PL/SQL)

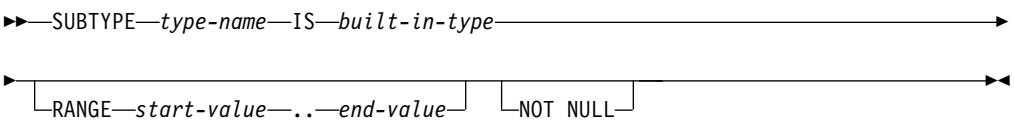
A *subtype* is a definition of a type based on a built-in type.

Subtypes provide a layer of abstraction between variables and parameters and the data types that they use. This layer allows you to concentrate any changes to the data types in one location. You can add constraints to subtypes so that they cannot be nullable or limited to a specific range of values.

Subtypes can be defined in:

- CREATE PACKAGE statement (PL/SQL)
- CREATE PACKAGE BODY statement (PL/SQL)
- CREATE PROCEDURE (PL/SQL)
- CREATE FUNCTION (PL/SQL)
- CREATE TRIGGER (PL/SQL)
- Anonymous block (PL/SQL)

Syntax



Description

SUBTYPE *type-name*
Specifies an identifier for the subtype. You cannot specify BOOLEAN as the built-in type.

built-in-type
Specifies the built-in data type that the subtype is based on.

RANGE *start-value .. end-value*
Optionally defines a range of values within the domain of the subtype that is valid for the subtype.

NOT NULL
Optionally defines that the subtype is not nullable.

Example

The following example shows a package that defines a subtype for small integers:

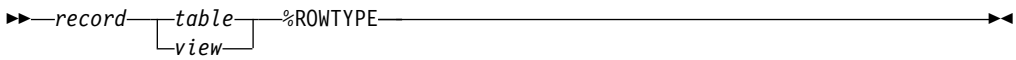
```
CREATE OR REPLACE PACKAGE math
IS
SUBTYPE tinyint IS INTEGER RANGE -256 .. 255 NOT NULL
END;
```

%ROWTYPE attribute in record type declarations (PL/SQL)

The %ROWTYPE attribute, used to declare PL/SQL variables of type record with fields that correspond to the columns of a table or view, is supported by the Db2 data server. Each field in a PL/SQL record assumes the data type of the corresponding column in the table.

A *record* is a named, ordered collection of fields. A *field* is similar to a variable; it has an identifier and a data type, but it also belongs to a record, and must be referenced using dot notation, with the record name as a qualifier.

Syntax



Description

record
Specifies an identifier for the record.

table
Specifies an identifier for the table whose column definitions will be used to define the fields in the record.

view
Specifies an identifier for the view whose column definitions will be used to define the fields in the record.

%ROWTYPE
Specifies that the record field data types are to be derived from the column data types that are associated with the identified table or view. Record fields do not inherit any other column attributes, such as, for example, the nullability attribute.

Example

The following example shows how to use the %ROWTYPE attribute to create a record (named r_emp) instead of declaring individual variables for the columns in the EMP table.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

Basic statements (PL/SQL)

The programming statements that can be used in a PL/SQL application include: assignment, DELETE, EXECUTE IMMEDIATE, INSERT, NULL, SELECT INTO, and UPDATE.

NULL statement (PL/SQL)

The NULL statement is an executable statement that does nothing. The NULL statement can act as a placeholder whenever an executable statement is required, but no SQL operation is wanted; for example, within a branch of the IF-THEN-ELSE statement.

Syntax

Examples

The following example shows the simplest valid PL/SQL program that the Db2 data server can compile:

```
BEGIN
    NULL;
END;
```

The following example shows the NULL statement within an IF...THEN...ELSE statement:

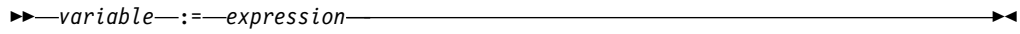
```
CREATE OR REPLACE PROCEDURE divide_it (
    p_numerator     IN  NUMBER,
    p_denominator   IN  NUMBER,
    p_result        OUT NUMBER
)
IS
```

```
BEGIN
    IF p_denominator = 0 THEN
        NULL;
    ELSE
        p_result := p_numerator / p_denominator;
    END IF;
END;
```

Assignment statement (PL/SQL)

The assignment statement sets a previously-declared variable or formal OUT or IN OUT parameter to the value of an expression.

Syntax



Description

variable
Specifies an identifier for a previously-declared variable, OUT formal parameter, or IN OUT formal parameter.

expression
Specifies an expression that evaluates to a single value. The data type of this value must be compatible with the data type of *variable*.

Example

The following example shows assignment statements in the executable section of a procedure:

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno       IN   NUMBER,
    p_base_annual  OUT  NUMBER
)
IS
    todays_date     DATE;
    rpt_title       VARCHAR2(60);
    base_sal        INTEGER;
    base_comm_rate  NUMBER;
BEGIN
    todays_date := SYSDATE;
    rpt_title := 'Report For Department # ' || p_deptno || ' on '
        || todays_date;
    base_sal := 35525;
    base_comm_rate := 1.33333;
    p_base_annual := ROUND(base_sal * base_comm_rate, 2);

    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || p_base_annual);
END
/
```

EXECUTE IMMEDIATE statement (PL/SQL)

The EXECUTE IMMEDIATE statement prepares an executable form of an SQL statement from a character string form of the statement and then executes the SQL statement. EXECUTE IMMEDIATE combines the basic functions of the PREPARE and EXECUTE statements.

Invocation

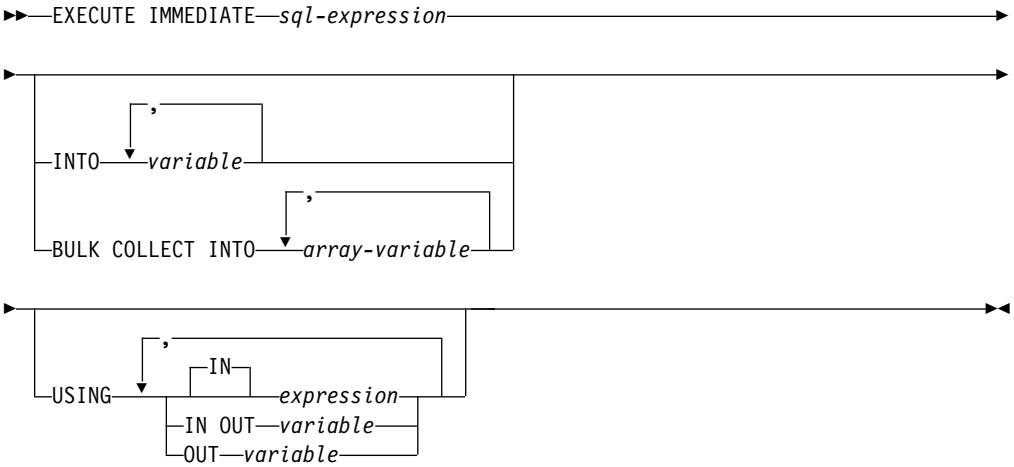
This statement can only be specified in a PL/SQL context.

Authorization

The authorization rules are those defined for the specified SQL statement.

The authorization ID of the statement might be affected by the DYNAMICRULES bind option.

Syntax



Description

sql-expression
An expression returning the statement string to be executed. The expression must return a character-string type that is less than the maximum statement size of 2 097 152 bytes. Note that a CLOB(2097152) can contain a maximum size statement, but a VARCHAR cannot.

The statement string must be one of the following SQL statements:

- ALTER
- CALL
- COMMENT
- COMMIT
- Compound SQL (compiled)
- Compound SQL (inlined)
- CREATE
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP
- EXPLAIN
- FLUSH EVENT MONITOR
- FLUSH PACKAGE CACHE
- GRANT

- INSERT
- LOCK TABLE
- MERGE
- REFRESH TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE
- ROLLBACK
- SAVEPOINT
- SELECT (only when the EXECUTE IMMEDIATE statement also specifies the BULK COLLECT INTO clause)
- SET COMPILATION ENVIRONMENT
- SET CURRENT DECFLOAT ROUNDING MODE
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE
- SET CURRENT FEDERATED ASYNCHRONY
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT IMPLICIT XMLPARSE OPTION
- SET CURRENT ISOLATION
- SET CURRENT LOCALE LC_TIME
- SET CURRENT LOCK TIMEOUT
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT MDC ROLLOUT MODE
- SET CURRENT OPTIMIZATION PROFILE
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET CURRENT SQL_CCFLAGS
- SET ROLE (only if DYNAMICRULES run behavior is in effect for the package)
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE (only if DYNAMICRULES run behavior is in effect for the package)
- SET INTEGRITY
- SET PASSTHRU
- SET PATH
- SET SCHEMA
- SET SERVER OPTION
- SET SESSION AUTHORIZATION
- SET variable
- TRANSFER OWNERSHIP (only if DYNAMICRULES run behavior is in effect for the package)
- TRUNCATE (only if DYNAMICRULES run behavior is in effect for the package)
- UPDATE

The statement string must not contain a statement terminator, with the exception of compound SQL statements which can contain semicolons (;) to separate statements within the compound block. A compound SQL statement is used within some CREATE and ALTER statements which, therefore, can also contain semicolons.

When an EXECUTE IMMEDIATE statement is executed, the specified statement string is parsed and checked for errors. If the SQL statement is invalid, it is not executed, and an exception is thrown.

INTO *variable*
Specifies the name of a variable that is to receive an output value from the corresponding parameter marker.

BULK COLLECT INTO *array-variable*
Identifies one or more variables with an array data type. Each row of the query is assigned to an element in each array in the order of the result set, with the array index assigned in sequence.

- If exactly one *array-variable* is specified:
  - If the data type of the *array-variable* element is not a record type, the SELECT list must have exactly one column and the column data type must be assignable to the array element data type.
  - If the data type of the *array-variable* element is a record type, the SELECT list must be assignable to the record type.
- If multiple array variables are specified:
  - The data type of the *array-variable* element must not be a record type.
  - There must be an *array-variable* for each column in the SELECT list.
  - The data type of each column in the SELECT list must be assignable to the array element data type of the corresponding *array-variable*.

If the data type of *array-variable* is an ordinary array, the maximum cardinality must be greater than or equal to the number of rows that are returned by the query.

This clause can only be used if the *sql-expression* is a SELECT statement.

USING

IN *expression*
Specifies a value that is passed to an input parameter marker. IN is the default.

IN OUT *variable*
Specifies the name of a variable that provides an input value to, or receives an output value from the corresponding parameter marker. This option is not supported when the `INTO` or `BULK COLLECT INTO` clause is used.

OUT *variable*
Specifies the name of a variable that receives an output value from the corresponding parameter marker. This option is not supported when the `INTO` or `BULK COLLECT INTO` clause is used.

The number and order of evaluated expressions or variables must match the number and order of-and be type-compatible with-the parameter markers in *sql-expression*.

Notes

- Statement caching affects the behavior of an EXECUTE IMMEDIATE statement.

Example

```
CREATE OR REPLACE PROCEDURE proc1( p1 IN NUMBER, p2 IN OUT NUMBER, p3 OUT NUMBER )
IS
BEGIN
  p3 := p1 + 1;
  p2 := p2 + 1;
END;
/

EXECUTE IMMEDIATE 'BEGIN proc1( :1, :2, :3 ); END' USING IN p1 + 10, IN OUT p3,
  OUT p2;

EXECUTE IMMEDIATE 'BEGIN proc1( :1, :2, :3 ); END' INTO p3, p2 USING p1 + 10, p3;
```

SQL statements (PL/SQL)

SQL statements that are supported within PL/SQL contexts can be used to modify data or to specify the manner in which statements are to be executed.

Table 4 lists these statements. The behavior of these statements when executed in PL/SQL contexts is equivalent to the behavior of the corresponding Db2 SQL statements.

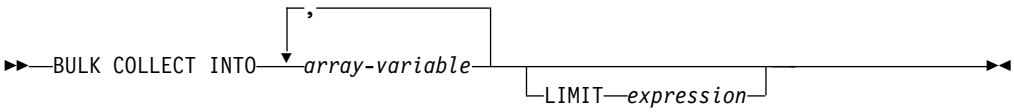
Table 4. SQL statements that can be executed by the Db2 data server within PL/SQL contexts

Command	Description
DELETE	Deletes rows from a table
INSERT	Inserts rows into a table
MERGE	Updates a target (a table or view) using data from a source (result of a table reference)
SELECT INTO	Retrieves rows from a table
UPDATE	Updates rows in a table

BULK COLLECT INTO clause (PL/SQL)

A SELECT INTO statement with the optional BULK COLLECT keywords preceding the INTO keyword retrieves multiple rows into an array.

Syntax



Description

BULK COLLECT INTO *array-variable*

Identifies one or more variables with an array data type. Each row of the result is assigned to an element in each array in the order of the result set, with the array index assigned in sequence.

- If exactly one *array-variable* is specified:
  - If the data type of the *array-variable* element is not a record type, the SELECT list must have exactly one column, and the column data type must be assignable to the array element data type.

  - If the data type of the *array-variable* element is a record type, the SELECT list must be assignable to the record type.
- If multiple array variables are specified:
  - The data type of the *array-variable* element must not be a record type.
  - There must be an *array-variable* for each column in the SELECT list.
  - The data type of each column in the SELECT list must be assignable to the array element data type of the corresponding *array-variable*.

  If the data type of *array-variable* is an ordinary array, the maximum cardinality must be greater than or equal to the number of rows that are returned by the query.

LIMIT *expression*
Provides an upper bound for how many rows are being fetched. The expression can be a numeric literal, a variable, or a complex expression, but it cannot depend on any column from the select statement.

Notes

- Variations of the BULK COLLECT INTO clause are also supported with the FETCH statement and the EXECUTE IMMEDIATE statement.

Example

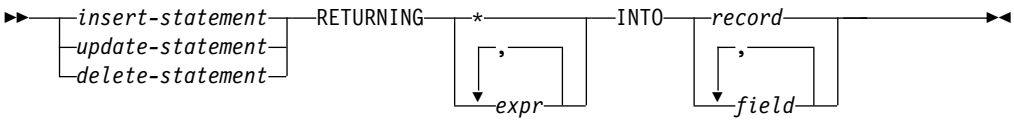
The following example shows a procedure that uses the BULK COLLECT INTO clause to return an array of rows from the procedure. The procedure and the type for the array are defined in a package.

```
CREATE OR REPLACE PACAKGE bci_sample
IS
 TYPE emps_array IS VARRAY (30) OF VARCHAR2(6);

 PROCEDURE get_dept_empno (
  dno       IN   emp.deptno%TYPE,
  emps_dno  OUT  emps_array
  );
END bci_sample;

CREATE OR REPLACE PACKAGE BODY bci_sample
IS
 PROCEDURE get_dept_empno (
  dno       IN   emp.deptno%TYPE,
  emps_dno  OUT  emps_array
  )
 IS
  BEGIN
   SELECT empno BULK COLLECT INTO emps_dno
    FROM emp
    WHERE deptno=dno;
  END get_dept_empno;
END bci_sample;
```

RETURNING INTO clause (PL/SQL)

INSERT, UPDATE, and DELETE statements that are appended with the optional RETURNING INTO clause can be compiled by the Db2 data server. When used in PL/SQL contexts, this clause captures the newly added, modified, or deleted values from executing INSERT, UPDATE, or DELETE statements, respectively.

Syntax



Description

insert-statement
Specifies a valid INSERT statement. An exception is raised if the INSERT statement returns a result set that contains more than one row.

update-statement
Specifies a valid UPDATE statement. An exception is raised if the UPDATE statement returns a result set that contains more than one row.

delete-statement
Specifies a valid DELETE statement. An exception is raised if the DELETE statement returns a result set that contains more than one row.

RETURNING *
Specifies that all of the values from the row that is affected by the INSERT, UPDATE, or DELETE statement are to be made available for assignment.

RETURNING *expr*
Specifies an expression to be evaluated against the row that is affected by the INSERT, UPDATE, or DELETE statement. The evaluated results are assigned to a specified record or fields.

INTO *record*
Specifies that the returned values are to be stored in a record with compatible fields and data types. The fields must match in number, order, and data type those values that are specified with the RETURNING clause. If the result set contains no rows, the fields in the record are set to the null value.

INTO *field*
Specifies that the returned values are to be stored in a set of variables with compatible fields and data types. The fields must match in number, order, and data type those values that are specified with the RETURNING clause. If the result set contains no rows, the fields are set to the null value.

Examples

The following example shows a procedure that uses the RETURNING INTO clause:

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno         IN emp.empno%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_deptno        emp.deptno%TYPE;
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
    RETURNING
```

```
        empno,
        ename,
        job,
        sal,
        comm,
        deptno
    INTO
        v_empno,
        v_ename,
        v_job,
        v_sal,
        v_comm,
        v_deptno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || v_empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || v_ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || v_job);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || v_deptno);
        DBMS_OUTPUT.PUT_LINE('New Salary         : ' || v_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission     : ' || v_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

This procedure returns the following sample output:

```
EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Department         : 40
New Salary         : 6540.00
New Commission     : 1200.00
```

The following example shows a procedure that uses the RETURNING INTO clause with record types:

```
CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
BEGIN
    DELETE FROM emp WHERE empno = p_empno
    RETURNING
        *
    INTO
        r_emp;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || r_emp.empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || r_emp.ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || r_emp.job);
        DBMS_OUTPUT.PUT_LINE('Manager            : ' || r_emp.mgr);
        DBMS_OUTPUT.PUT_LINE('Hire Date          : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary             : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Commission         : ' || r_emp.comm);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || r_emp.deptno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

This procedure returns the following sample output:

```
EXEC emp_delete(9503);

Deleted Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Manager            : 7902
Hire Date          : 31-MAR-05 00:00:00
Salary             : 6540.00
Commission         : 1200.00
Department         : 40
```

Statement attributes (PL/SQL)

SQL%FOUND, SQL%NOTFOUND, and SQL%ROWCOUNT are PL/SQL attributes that can be used to determine the effect of an SQL statement.

- The SQL%FOUND attribute has a Boolean value that returns TRUE if at least one row was affected by an INSERT, UPDATE, or DELETE statement, or if a SELECT INTO statement retrieved one row. The following example shows an anonymous block in which a row is inserted and a status message is displayed.

```
BEGIN
    INSERT INTO emp (empno,ename,job,sal,deptno)
        VALUES (9001, 'JONES', 'CLERK', 850.00, 40);
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Row has been inserted');
    END IF;
END;
```

- The SQL%NOTFOUND attribute has a Boolean value that returns TRUE if no rows were affected by an INSERT, UPDATE, or DELETE statement, or if a SELECT INTO statement did not retrieve a row. For example:

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9000;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No rows were updated');
    END IF;
END;
```

- The SQL%ROWCOUNT attribute has an integer value that represents the number of rows that were affected by an INSERT, UPDATE, or DELETE statement. For example:

```
BEGIN
    UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9001;
    DBMS_OUTPUT.PUT_LINE('# rows updated: ' || SQL%ROWCOUNT);
END;
```

Control statements (PL/SQL)

Control statements are the programming statements that make PL/SQL a full procedural complement to SQL.

A number of PL/SQL control statements can be compiled by the Db2 data server.

IF statement (PL/SQL)

Use the IF statement within PL/SQL contexts to execute SQL statements on the basis of certain criteria.

The four forms of the IF statement are:

- IF...THEN...END IF
- IF...THEN...ELSE...END IF
- IF...THEN...ELSE IF...END IF

- IF...THEN...ELSIF...THEN...ELSE...END IF

IF...THEN...END IF

The syntax of this statement is:

```
IF boolean-expression THEN
  statements
END IF;
```

IF...THEN statements are the simplest form of IF. The statements between THEN and END IF are executed only if the condition evaluates to TRUE. In the following example, an IF...THEN statement is used to test for and to display those employees who have a commission.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

This program generates the following sample output:

```
EMPNO    COMM
-----    -------
7499     $300.00
7521     $500.00
7654    $1400.00
```

IF...THEN...ELSE...END IF

The syntax of this statement is:

```
IF boolean-expression THEN
  statements
ELSE
  statements
END IF;
```

IF...THEN...ELSE statements specify an alternative set of statements that should be executed if the condition evaluates to FALSE. In the following example, the previous example is modified so that an IF...THEN...ELSE statement is used to display the text “Non-commission” if an employee does not have a commission.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
```

```
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || 'Non-commission');
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

This program generates the following sample output:

```
EMPNO    COMM
-----    -------
7369     Non-commission
7499  $   300.00
7521  $   500.00
7566     Non-commission
7654  $  1400.00
7698     Non-commission
7782     Non-commission
7788     Non-commission
7839     Non-commission
7844     Non-commission
7876     Non-commission
7900     Non-commission
7902     Non-commission
7934     Non-commission
```

IF...THEN...ELSE IF...END IF

The syntax of this statement is:

```
IF boolean-expression THEN
  IF boolean-expression THEN
    statements
ELSE
  IF boolean-expression THEN
    statements
END IF;
```

You can nest IF statements so that alternative IF statements are invoked, depending on whether the conditions of an outer IF statement evaluate to TRUE or FALSE. In the following example, the outer IF...THEN...ELSE statement tests whether or not an employee has a commission. The inner IF...THEN...ELSE statements subsequently test whether the employee's total compensation exceeds or is less than the company average. When you use this form of the IF statement, you are actually nesting an IF statement inside of the ELSE part of an outer IF statement. You therefore need one END IF for each nested IF and one for the parent IF...ELSE. (Note that the logic in this program can be simplified considerably by calculating each employee's yearly compensation using an NVL function within the SELECT statement of the cursor declaration; however, the purpose of this example is to demonstrate how IF statements can be used.)

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_sal           emp.sal%TYPE;
```

```
    v_comm          emp.comm%TYPE;
    v_avg           NUMBER(7,2);
    CURSOR emp_cursor IS SELECT empno, sal, comm FROM emp;
BEGIN
--
--  Calculate the average yearly compensation
--
    SELECT AVG((sal + NVL(comm,0)) * 24) INTO v_avg FROM emp;
    DBMS_OUTPUT.PUT_LINE('Average Yearly Compensation: ' ||
        TO_CHAR(v_avg,'$999,999.99'));
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    YEARLY COMP');
    DBMS_OUTPUT.PUT_LINE('-----    -----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_sal, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
--
--  Test whether the employee's compensation with commission exceeds
--  the company average
--
            IF (v_sal + v_comm) * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Below Average');
            END IF;
        ELSE
--
--  Test whether the employee's compensation without commission exceeds
--  the company average
--
            IF v_sal * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') || ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') || ' Below Average');
            END IF;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

This program generates the following sample output:

```
Average Yearly Compensation: $  53,528.57
EMPNO    YEARLY COMP
-----    -----------
7369  $  19,200.00 Below Average
7499  $  45,600.00 Below Average
7521  $  42,000.00 Below Average
7566  $  71,400.00 Exceeds Average
7654  $  63,600.00 Exceeds Average
7698  $  68,400.00 Exceeds Average
7782  $  58,800.00 Exceeds Average
7788  $  72,000.00 Exceeds Average
7839  $ 120,000.00 Exceeds Average
7844  $  36,000.00 Below Average
```

```
7876  $  26,400.00 Below Average
7900  $  22,800.00 Below Average
7902  $  72,000.00 Exceeds Average
7934  $  31,200.00 Below Average
```

IF...THEN...ELSIF...THEN...ELSE...END IF

The syntax of this statement is:

```
IF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements ] ...]
[ ELSE
  statements ]
END IF;
```

IF...THEN...ELSIF...ELSE statements provide the means for checking many alternatives in one statement. Formally, this statement is equivalent to nested IF...THEN...ELSE...IF...THEN statements, but only one END IF is needed. The following example uses an IF...THEN...ELSIF...ELSE statement to count the number of employees by compensation, in steps of $25,000.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comp          NUMBER(8,2);
    v_lt_25K        SMALLINT := 0;
    v_25K_50K       SMALLINT := 0;
    v_50K_75K       SMALLINT := 0;
    v_75K_100K      SMALLINT := 0;
    v_ge_100K       SMALLINT := 0;
    CURSOR emp_cursor IS SELECT empno, (sal + NVL(comm,0)) * 24 FROM emp;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_empno, v_comp;
        EXIT WHEN emp_cursor%NOTFOUND;
        IF v_comp < 25000 THEN
            v_lt_25K := v_lt_25K + 1;
        ELSIF v_comp < 50000 THEN
            v_25K_50K := v_25K_50K + 1;
        ELSIF v_comp < 75000 THEN
            v_50K_75K := v_50K_75K + 1;
        ELSIF v_comp < 100000 THEN
            v_75K_100K := v_75K_100K + 1;
        ELSE
            v_ge_100K := v_ge_100K + 1;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
    DBMS_OUTPUT.PUT_LINE('Number of employees by yearly compensation');
    DBMS_OUTPUT.PUT_LINE('Less than 25,000 : ' || v_lt_25K);
    DBMS_OUTPUT.PUT_LINE('25,000 - 49,9999 : ' || v_25K_50K);
    DBMS_OUTPUT.PUT_LINE('50,000 - 74,9999 : ' || v_50K_75K);
    DBMS_OUTPUT.PUT_LINE('75,000 - 99,9999 : ' || v_75K_100K);
    DBMS_OUTPUT.PUT_LINE('100,000 and over : ' || v_ge_100K);
END;
```

This program generates the following sample output:

```
Number of employees by yearly compensation
Less than 25,000 : 2
25,000 - 49,9999 : 5
50,000 - 74,9999 : 6
75,000 - 99,9999 : 0
100,000 and over : 1
```

CASE statement (PL/SQL)

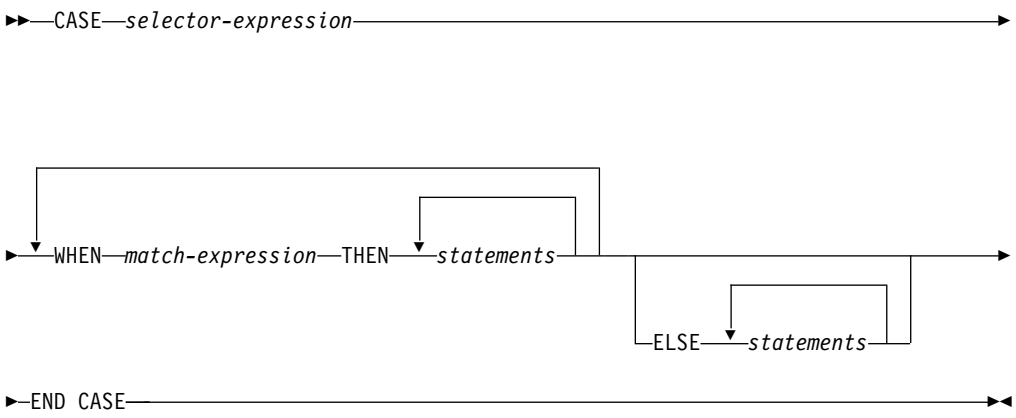
The CASE statement executes a set of one or more statements when a specified search condition is true. CASE is a standalone statement that is distinct from the CASE expression, which must appear as part of an expression.

There are two forms of the CASE statement: the simple CASE statement and the searched CASE statement.

Simple CASE statement (PL/SQL)

The simple CASE statement attempts to match an expression (known as the *selector*) to another expression that is specified in one or more WHEN clauses. A match results in the execution of one or more corresponding statements.

Syntax



Description

CASE *selector-expression*

Specifies an expression whose value has a data type that is compatible with each *match-expression*. If the value of *selector-expression* matches the first *match-expression*, the statements in the corresponding THEN clause are executed. If there are no matches, the statements in the corresponding ELSE clause are executed. If there are no matches and there is no ELSE clause, an exception is thrown.

WHEN *match-expression*

Specifies an expression that is evaluated within the CASE statement. If *selector-expression* matches a *match-expression*, the statements in the corresponding THEN clause are executed.

THEN

A keyword that introduces the statements that are to be executed when the corresponding Boolean expression evaluates to TRUE.

statements
Specifies one or more SQL or PL/SQL statements, each terminated with a semicolon.

ELSE
A keyword that introduces the default case of the CASE statement.

Example

The following example uses a simple CASE statement to assign a department name and location to a variable that is based upon the department number.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE v_deptno
            WHEN 10 THEN v_dname := 'Accounting';
                         v_loc   := 'New York';
            WHEN 20 THEN v_dname := 'Research';
                         v_loc   := 'Dallas';
            WHEN 30 THEN v_dname := 'Sales';
                         v_loc   := 'Chicago';
            WHEN 40 THEN v_dname := 'Operations';
                         v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                         v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

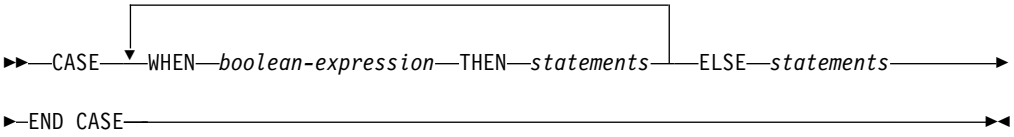
This program returns the following sample output:

```
EMPNO    ENAME     DEPTNO    DNAME          LOC
-----    -------   ------    ----------     ---------
7369     SMITH       20      Research       Dallas
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
7698     BLAKE       30      Sales          Chicago
7782     CLARK       10      Accounting     New York
7788     SCOTT       20      Research       Dallas
7839     KING        10      Accounting     New York
7844     TURNER      30      Sales          Chicago
7876     ADAMS       20      Research       Dallas
7900     JAMES       30      Sales          Chicago
7902     FORD        20      Research       Dallas
7934     MILLER      10      Accounting     New York
```

Searched CASE statement (PL/SQL)

A searched CASE statement uses one or more Boolean expressions to determine which statements to execute.

Syntax



Description

CASE
A keyword that introduces the first WHEN clause in the CASE statement.

WHEN *boolean-expression*
Specifies an expression that is evaluated when control flow enters the WHEN clause in which the expression is defined. If *boolean-expression* evaluates to TRUE, the statements in the corresponding THEN clause are executed. If *boolean-expression* does not evaluate to TRUE, the statements in the corresponding ELSE clause are executed.

THEN
A keyword that introduces the statements that are to be executed when the corresponding Boolean expression evaluates to TRUE.

statements
Specifies one or more SQL or PL/SQL statements, each terminated with a semicolon.

ELSE
A keyword that introduces the default case of the CASE statement.

Example

The following example uses a searched CASE statement to assign a department name and location to a variable that is based upon the department number.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE
            WHEN v_deptno = 10 THEN v_dname := 'Accounting';
                                    v_loc   := 'New York';
            WHEN v_deptno = 20 THEN v_dname := 'Research';
```

```
                                    v_loc   := 'Dallas';
            WHEN v_deptno = 30 THEN v_dname := 'Sales';
                                    v_loc   := 'Chicago';
            WHEN v_deptno = 40 THEN v_dname := 'Operations';
                                    v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                                    v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

This program returns the following sample output:

```
EMPNO     ENAME      DEPTNO     DNAME           LOC
-----     -------    ------     ----------      ---------
7369      SMITH        20       Research        Dallas
7499      ALLEN        30       Sales           Chicago
7521      WARD         30       Sales           Chicago
7566      JONES        20       Research        Dallas
7654      MARTIN       30       Sales           Chicago
7698      BLAKE        30       Sales           Chicago
7782      CLARK        10       Accounting      New York
7788      SCOTT        20       Research        Dallas
7839      KING         10       Accounting      New York
7844      TURNER       30       Sales           Chicago
7876      ADAMS        20       Research        Dallas
7900      JAMES        30       Sales           Chicago
7902      FORD         20       Research        Dallas
7934      MILLER       10       Accounting      New York
```

Loops (PL/SQL)

Use the EXIT, FOR, LOOP, and WHILE statements to repeat a series of commands in your PL/SQL program.

FOR (cursor variant) statement (PL/SQL)

The cursor FOR loop statement opens a previously declared cursor, fetches all rows in the cursor result set, and then closes the cursor.

Use this statement instead of separate SQL statements to open a cursor, define a loop construct to retrieve each row of the result set, test for the end of the result set, and then finally close the cursor.

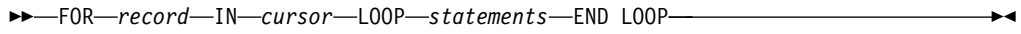
Invocation

This statement can be invoked within a PL/SQL procedure, function, trigger, or anonymous block.

Authorization

No specific authorization is required to reference a row expression within an SQL statement; however, for successful statement execution, all other authorization requirements for processing a cursor are required.

Syntax



Description

FOR
Introduces the condition that must be true if the FOR loop is to proceed.

record
Specifies an identifier that was assigned to an implicitly declared record with definition cursor%ROWTYPE.

IN *cursor*
Specifies the name of a previously declared cursor.

LOOP and END LOOP
Starts and ends the loop containing SQL statements that are to be executed during each iteration through the loop.

statements
One or more PL/SQL statements. A minimum of one statement is required.

Example

The following example shows a procedure that contains a cursor FOR loop:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  FOR v_emp_rec IN emp_cur_1 LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
  END LOOP;
END;
```

FOR (integer variant) statement (PL/SQL)

Use the FOR statement to execute a set of SQL statements more than once.

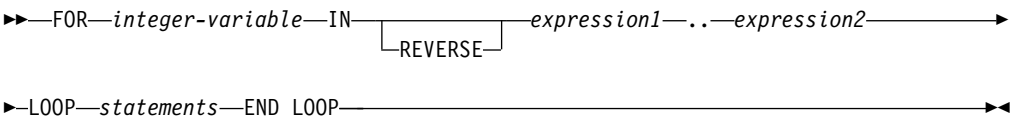
Invocation

This statement can be embedded within a PL/SQL procedure, function, or anonymous block statement.

Authorization

No privileges are required to invoke the FOR statement; however, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded in the FOR statement.

Syntax



Description

integer-variable
An automatically defined integer variable that is used during loop processing. The initial value of *integer-variable* is *expression1*. After the initial iteration, the value of *integer-variable* is incremented at the beginning of each subsequent

iteration. Both *expression1* and *expression2* are evaluated when entering the loop, and loop processing stops when *integer-variable* is equal to *expression2*.

IN Introduces the optional REVERSE keyword and expressions that define the range of integer variables for the loop.

REVERSE
Specifies that the iteration is to proceed from *expression2* to *expression1*. Note that *expression2* must have a higher value than *expression1*, regardless of whether the REVERSE keyword is specified, if the statements in the loop are to be processed.

expression1
Specifies the initial value of the range of integer variables for the loop. If the REVERSE keyword is specified, *expression1* specifies the end value of the range of integer variables for the loop.

expression2
Specifies the end value of the range of integer variables for the loop. If the REVERSE keyword is specified, *expression2* specifies the initial value of the range of integer variables for the loop.

statements
Specifies the PL/SQL and SQL statements that are executed each time that the loop is processed.

Examples

The following example shows a basic FOR statement within an anonymous block:

```
BEGIN
  FOR i IN 1 .. 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

This example generates the following output:

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

If the start value is greater than the end value, the loop body is not executed at all, but no error is returned, as shown by the following example:

```
BEGIN
  FOR i IN 10 .. 1 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

This example generates no output, because the loop body is never executed.

The following example uses the REVERSE keyword:

```
BEGIN
  FOR i IN REVERSE 1 .. 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

This example generates the following output:

```
Iteration # 10
Iteration # 9
Iteration # 8
Iteration # 7
Iteration # 6
Iteration # 5
Iteration # 4
Iteration # 3
Iteration # 2
Iteration # 1
```

FORALL statement (PL/SQL)

The FORALL statement executes a data change statement for all elements of an array or for a range of elements of an array.

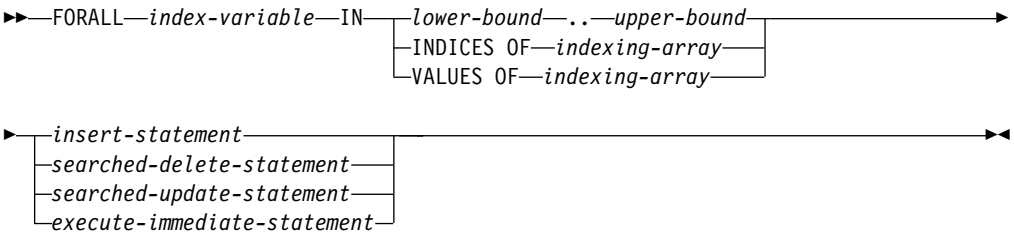
Invocation

This statement can only be specified in a PL/SQL block.

Authorization

The privileges held by the authorization ID of the statement must include all of the privileges necessary to invoke the data change statement that is specified in the FORALL statement.

Syntax



Description

index-variable
: Identifies a name to be used as an array index. It is implicitly declared as an INTEGER and it can only be referenced in the FORALL statement.

lower-bound **..** *upper-bound*
: Identifies a range of index values that are assignable to the *index-variable* with *lower-bound* less than *upper-bound*. The range represents every integer value starting with *lower-bound* and incrementing by 1 up to and including *upper-bound*.

INDICES OF *indexing-array*
: Identifies the set of array index values of the array identified by *indexing-array*. If *indexing-array* is an associative array, array index values must be assignable to *index-variable* and could be a sparse set.

VALUES OF *indexing-array*
Identifies the set of element values of the array identified by *indexing-array*. The element values must be assignable to *index-variable* and could be an unordered sparse set.

insert-statement
Specifies an INSERT statement that is effectively executed for each *index-variable* value.

searched-delete-statement
Specifies a searched DELETE statement that is effectively executed for each *index-variable* value.

searched-update-statement
Specifies a searched UPDATE statement that is effectively executed for each *index-variable* value.

execute-immediate-statement
Specifies an EXECUTE IMMEDIATE statement that is effectively executed for each *index-variable* value.

Notes

- FORALL statement processing is not atomic. If an error occurs while iterating in the FORALL statement, any data change operations that have already been processed are not implicitly rolled back. An application can use a ROLLBACK statement to roll back the entire transaction when an error occurs in the FORALL statement.

Example

The following example shows a basic FORALL statement:

```
FORALL x
  IN in_customer_list.FIRST..in_customer_list.LAST
  DELETE FROM customer
    WHERE cust_id IN in_customer_list(x);
```

EXIT statement (PL/SQL)

The EXIT statement terminates execution of a loop within a PL/SQL code block.

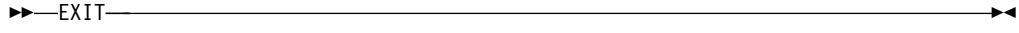
Invocation

This statement can be embedded within a FOR, LOOP, or WHILE statement in a PL/SQL procedure, function, or anonymous block.

Authorization

No privileges are required to invoke the EXIT statement. However, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded within the FOR, LOOP, or WHILE statement.

Syntax



Example

The following example shows a basic LOOP statement with an EXIT statement within an anonymous block:

```
DECLARE
  sum PLS_INTEGER := 0;
BEGIN
  LOOP
    sum := sum + 1;
    IF sum > 10 THEN
       EXIT;
    END IF;
  END LOOP;
END
```

LOOP statement (PL/SQL)

The LOOP statement executes a sequence of statements within a PL/SQL code block multiple times.

Invocation

This statement can be embedded in a PL/SQL procedure, function, or anonymous block statement.

Authorization

No privileges are required to invoke the LOOP statement. However, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded within the LOOP statement.

Syntax

Description

statements
Specifies one or more PL/SQL or SQL statements. These statements are executed during each iteration of the loop.

Example

The following example shows a basic LOOP statement within an anonymous block:

```
DECLARE
  sum INTEGER := 0;
BEGIN
  LOOP
    sum := sum + 1;
    IF sum > 10 THEN
       EXIT;
    END IF;
  END LOOP;
END
```

WHILE statement (PL/SQL)

The WHILE statement repeats a set of SQL statements as long as a specified expression is true. The condition is evaluated immediately before each entry into the loop body.

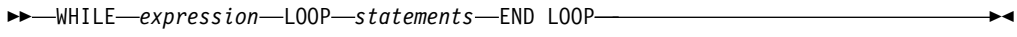
Invocation

This statement can be embedded within a PL/SQL procedure, function, or anonymous block statement.

Authorization

No privileges are required to invoke the WHILE statement; however, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded in the WHILE statement.

Syntax



Description

expression
Specifies an expression that is evaluated immediately before each entry into the loop body to determine whether or not the loop is to be executed. If the expression is logically true, the loop is executed. If the expression is logically false, loop processing ends. An EXIT statement can be used to terminate the loop while the expression is true.

statements
Specifies the PL/SQL and SQL statements that are executed each time that the loop is processed.

Example

The following example shows a basic WHILE statement within an anonymous block:

```
DECLARE
  sum INTEGER := 0;
BEGIN
  WHILE sum < 11 LOOP
    sum := sum + 1;
  END LOOP;
END
```

The WHILE statement within this anonymous block executes until *sum* is equal to 11; loop processing then ends, and processing of the anonymous block proceeds to completion.

CONTINUE statement (PL/SQL)

The CONTINUE statement terminates the current iteration of a loop within a PL/SQL code block, and moves to the next iteration of the loop.

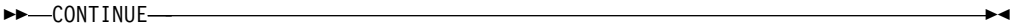
Invocation

This statement can be embedded within a FOR, LOOP, or WHILE statement, or within a PL/SQL procedure, function, or anonymous block statement.

Authorization

No privileges are required to invoke the CONTINUE statement. However, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded within the FOR, LOOP, or WHILE statement.

Syntax



Example

The following example shows a basic LOOP statement with an CONTINUE statement within an anonymous block:

```
BEGIN
  FOR i IN 1 .. 5 LOOP
    IF i = 3 THEN
       CONTINUE;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

This example generates the following output:

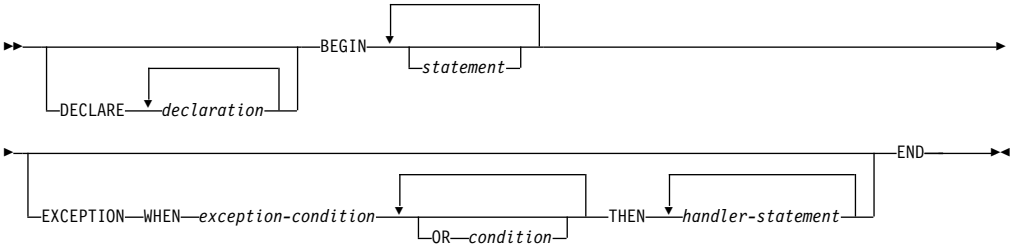
```
Iteration # 1
Iteration # 2
Iteration # 4
Iteration # 5
```

Exception handling (PL/SQL)

By default, any error encountered in a PL/SQL program stops execution of the program. You can trap and recover from errors by using an EXCEPTION section.

The syntax for exception handlers is an extension of the syntax for a BEGIN block.

Syntax



If no error occurs, the block simply executes *statement*, and control passes to the statement after END. However, if an error occurs while executing a *statement*, further processing of the *statement* is abandoned, and control passes to the EXCEPTION section. The WHEN clauses are searched for the first exception matching the error that occurred. If a match is found, the corresponding *handler-statement* is executed, and control passes to the statement after END. If no match is found, the program stops executing.

If a new error occurs during execution of the *handler-statement*, it can only be caught by a surrounding EXCEPTION clause.

Exceptions in a WHEN clause can be either user-defined or built-in. User-defined exceptions can be defined in the DECLARE section of either the current block or its surrounding block, or in the DECLARE section of a PL/SQL package. The syntax PRAGMA EXCEPTION_INIT or PRAGMA DB2_EXCEPTION_INIT can be used immediately after the definition of an exception, specifying the sqlcode or sqlstate that corresponds to the user-defined exception.

In the following example, the DECLARE section contains the definitions of three named exceptions. The body of the block is a call to procedure `MyApp.Main`. The EXCEPTION section contains handlers for the three exceptions:

1. `exception1` is not associated with an sqlcode or sqlstate .
2. `exception2` is associated with sqlcode -942 (Undefined name).
3. `exception3` is associated with sqlstate 42601 (syntax error).

```
DECLARE
  exception1 EXCEPTION;
  exception2 EXCEPTION;
  PRAGMA EXCEPTION_INIT(exception2,-942);
  exception3 EXCEPTION;
  PRAGMA DB2_EXCEPTION_INIT(exception3,'42601');
BEGIN
  MyApp.Main(100);
EXCEPTION
  WHEN exception1 THEN
      DBMS_OUTPUT.PUT_LINE('User-defined exception1 caught');
  WHEN exception2 THEN
      DBMS_OUTPUT.PUT_LINE('User-defined exception2 (Undefined name) caught');
  WHEN exception3 THEN
      DBMS_OUTPUT.PUT_LINE('User-defined exception3 (Syntax error) caught');
END
```

Note: A limited number of Oracle sqlcodes are accepted by the Db2 data server as arguments to PRAGMA EXCEPTION_INIT. Refer to "Oracle-Db2 error mapping (PL/SQL)" on page 74 for the full list.

When an exception initialized with PRAGMA EXCEPTION_INIT is caught, the value returned by the SQLCODE function is the sqlcode associated with the exception, not the Oracle value. In the previous example, when `exception2` is caught, the value returned by SQLCODE will be -204, which is the sqlcode corresponding to Oracle sqlcode -942. If the Oracle sqlcode specified in PRAGMA EXCEPTION_INIT is not listed in the Oracle-Db2 error mapping table, then compilation fails. You can avoid this by replacing PRAGMA EXCEPTION_INIT with PRAGMA DB2_EXCEPTION_INIT and specifying the Db2 sqlstate corresponding to the error that you want identified.

Table 5 summarizes the built-in exceptions that you can use. The special exception name OTHERS matches every exception. Condition names are not case sensitive.

Table 5. Built-in exception names

Exception name	Description
CASE_NOT_FOUND	None of the cases in a CASE statement evaluates to "true", and there is no ELSE condition.
CURSOR_ALREADY_OPEN	An attempt was made to open a cursor that is already open.

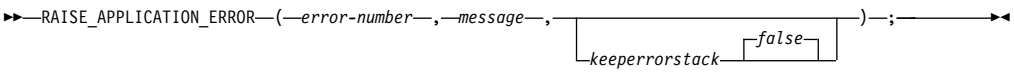
Table 5. Built-in exception names (continued)

Exception name	Description
DUP_VAL_ON_INDEX	There are duplicate values for the index key.
INVALID_CURSOR	An attempt was made to access an unopened cursor.
INVALID_NUMBER	The numeric value is invalid.
LOGIN_DENIED	The user name or password is invalid.
NO_DATA_FOUND	No rows satisfied the selection criteria.
NOT_LOGGED_ON	A database connection does not exist.
OTHERS	For any exception that has not been caught by a prior condition in the exception section.
SUBSCRIPT_BEYOND_COUNT	An array index is out of range or does not exist.
SUBSCRIPT_OUTSIDE_LIMIT	The data type of an array index expression is not assignable to the array index type.
TOO_MANY_ROWS	More than one row satisfied the selection criteria, but only one row is allowed to be returned.
VALUE_ERROR	The value is invalid.
ZERO_DIVIDE	Division by zero was attempted.

Raise application error (PL/SQL)

The RAISE_APPLICATION_ERROR procedure raises an exception based on a user-provided error code and message. This procedure is only supported in PL/SQL contexts.

Syntax



Description

error-number
A vendor-specific number that is mapped to an error code before it is stored in a variable named SQLCODE. The RAISE_APPLICATION_ERROR procedure accepts user-defined *error-number* values from -20000 to -20999. The SQLCODE that is returned in the error message is SQL0438N. The SQLSTATE contains class 'UD' plus three characters that correspond to the last three digits of the *error-number* value.

message
A user-defined message with a maximum length of 70 bytes.

keeperrorstack
An optional boolean value indicating whether the error stack should be preserved. Currently, only the default value of *false* is supported.

Example

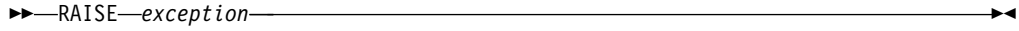
The following example uses the RAISE_APPLICATION_ERROR procedure to display error codes and messages that are specific to missing employee information:

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno         NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_mgr           emp.mgr%TYPE;
    v_hiredate      emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, mgr, hiredate
        INTO v_ename, v_job, v_mgr, v_hiredate FROM emp
        WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        RAISE_APPLICATION_ERROR(-20010, 'No name for ' || p_empno);
    END IF;
    IF v_job IS NULL THEN
        RAISE_APPLICATION_ERROR(-20020, 'No job for' || p_empno);
    END IF;
    IF v_mgr IS NULL THEN
        RAISE_APPLICATION_ERROR(-20030, 'No manager for ' || p_empno);
    END IF;
    IF v_hiredate IS NULL THEN
        RAISE_APPLICATION_ERROR(-20040, 'No hire date for ' || p_empno);
    END IF;
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' validated without errors');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
END;

CALL verify_emp(7839);

SQLCODE: -438
SQLERRM: SQL0438N  Application raised error or warning with
 diagnostic text: "No manager for 7839".  SQLSTATE=UD030
```

RAISE statement (PL/SQL)

The RAISE statement raises a previously-defined exception.

Syntax



Description

exception
Specifies a previously-defined exception.

Example

The following example shows a procedure that raises an exception of `oddno` or `evenno` based on the value provided as argument in the invocation::

```
CREATE OR REPLACE PROCEDURE raise_demo (inval NUMBER) IS
  evenno EXCEPTION;
  oddno  EXCEPTION;
BEGIN
  IF MOD(inval, 2) = 1 THEN
    RAISE oddno;
  ELSE
    RAISE evenno;
  END IF;
EXCEPTION
  WHEN evenno THEN
    dbms_output.put_line(TO_CHAR(inval) || ' is even');
  WHEN oddno THEN
    dbms_output.put_line(TO_CHAR(inval) || ' is odd');
END raise_demo;
/

SET SERVEROUTPUT ON
/

CALL raise_demo (11)
/
```

The output of the CALL statement would be:

```
CALL raise_demo (11)

 Return Status = 0

11 is odd
```

Oracle-Db2 error mapping (PL/SQL)

PL/SQL error codes and exception names have corresponding Db2 error codes and SQLSTATE values.

These error codes, exception names, and SQLSTATE values are summarized in Table 6.

Table 6. Mapping of PL/SQL error codes and exception names to Db2 data server error codes and SQLSTATE values

plsqlCode	plsqlName	db2Code	db2State
-1	DUP_VAL_ON_INDEX	-803	23505
+100	NO_DATA_FOUND	+100	02000
-1012	NOT_LOGGED_ON	-1024	08003
-1017	LOGIN_DENIED	-30082	08001
-1476	ZERO_DIVIDE	-801	22012
-1722	INVALID_NUMBER	-420	22018
-1001	INVALID_CURSOR	-501	24501
-1422	TOO_MANY_ROWS	-811	21000
-6502	VALUE_ERROR	-433	22001
-6511	CURSOR_ALREADY_OPEN	-502	24502
-6532	SUBSCRIPT_OUTSIDE_LIMIT	-20439	428H1
-6533	SUBSCRIPT_BEYOND_COUNT	-20439	2202E
-6592	CASE_NOT_FOUND	-773	20000
-54		-904	57011

Table 6. Mapping of PL/SQL error codes and exception names to Db2 data server error codes and SQLSTATE values (continued)

plsqlCode	plsqlName	db2Code	db2State
-60		-911	40001
-310		-206	42703
-595		-390	42887
-597		-303	42806
-598		-407	23502
-600		-30071	58015
-603		-119	42803
-604		-119	42803
-610		-20500	428HR
-611		-117	42802
-612		-117	42802
-613		-811	21000
-615		-420	22018
-616		-420	22018
-617		-418	42610
-618		-420	22018
-619		-418	42610
-620		-171	42815
-622		-304	22003
-623		-604	42611
-904		-206	42703
-911		-7	42601
-942		-204	42704
-955		-601	42710
-996		-1022	57011
-1119		-292	57047
-1002		+231	02000
-1403		-100	02000
-1430		-612	42711
-1436		-20451	560CO
-1438		-413	22003
-1450		-614	54008
-1578		-1007	58034
-2112		-811	21000
-2261		+605	01550
-2291		-530	23503
-2292		-532	23001
-3113		-30081	08001
-3114		-1024	08003

Table 6. Mapping of PL/SQL error codes and exception names to Db2 data server error codes and SQLSTATE values (continued)

plsqlCode	plsqlName	db2Code	db2State
-3214		-20170	57059
-3297		-20170	57059
-4061		-727	56098
-4063		-727	56098
-4091		-723	09000
-6502		-304	22003
-6508		-440	42884
-6550		-104	42601
-6553		-104	42601
-14028		-538	42830
-19567		-1523	55039
-30006		-904	57011
-30041		-1139	54047

Cursors (PL/SQL)

A *cursor* is a named control structure used by an application program to point to and select a row of data from a result set. Instead of executing a query all at once, you can use a cursor to read and process the query result set one row at a time.

A cursor in a PL/SQL context is treated as a WITH HOLD cursor. For more information about WITH HOLD cursors, see "DECLARE CURSOR statement".

The Db2 data server supports both PL/SQL static cursors and cursor variables.

Static cursors (PL/SQL)

A *static cursor* is a cursor whose associated query is fixed at compile time. Declaring a cursor is a prerequisite to using it. Declarations of static cursors using PL/SQL syntax within PL/SQL contexts are supported by the Db2 data server.

Syntax

Description

cursor-name
Specifies an identifier for the cursor that can be used to reference the cursor and its result set.

query
Specifies a SELECT statement that determines a result set for the cursor.

Example

The following example shows a procedure that contains multiple static cursor declarations:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
     CURSOR emp_cur_1 IS SELECT * FROM emp;

     CURSOR emp_cur_2 IS SELECT empno, ename FROM emp;

     CURSOR emp_cur_3 IS SELECT empno, ename
                         FROM emp
                         WHERE deptno = 10
                         ORDER BY empno;
BEGIN
    OPEN emp_cur_1;
        ...
END;
```

Parameterized cursors (PL/SQL)

Parameterized cursors are static cursors that can accept passed-in parameter values when they are opened.

The following example includes a parameterized cursor. The cursor displays the name and salary of each employee in the EMP table whose salary is less than that specified by a passed-in parameter value.

```
DECLARE
    my_record       emp%ROWTYPE;
    CURSOR c1 (max_wage NUMBER) IS
        SELECT * FROM emp WHERE sal < max_wage;
BEGIN
    OPEN c1(2000);
    LOOP
        FETCH c1 INTO my_record;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = '
            || my_record.sal);
    END LOOP;
    CLOSE c1;
END;
```

If 2000 is passed in as the value of *max_wage*, only the name and salary data for those employees whose salary is less than 2000 is returned:

```
Name = SMITH, salary = 800.00
Name = ALLEN, salary = 1600.00
Name = WARD, salary = 1250.00
Name = MARTIN, salary = 1250.00
Name = TURNER, salary = 1500.00
Name = ADAMS, salary = 1100.00
Name = JAMES, salary = 950.00
Name = MILLER, salary = 1300.00
```

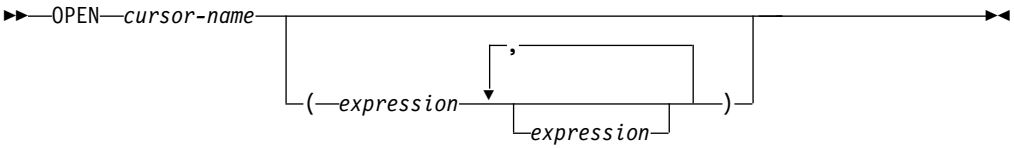
Parameterized cursors can only reference its own parameters. Parameterized cursors cannot reference local variables. In this example, `cursor_id` must be used in the select statement because `in_id` is not within the scope of the cursor.

```
CREATE OR REPLACE PROCEDURE myproc (in_id IN NUMBER) IS
  CURSOR c(cursor_id in NUMBER) IS
  SELECT id,emp_name FROM employee WHERE id = cursor_id;
  empName VARCHAR2(100);

  BEGIN
    FOR r IN c(in_id) LOOP
      empName := r.emp_name;
      DBMS_OUTPUT.PUT_LINE(empName);
    END LOOP;
  END;
```

Opening a cursor (PL/SQL)

The result set that is associated with a cursor cannot be referenced until the cursor has been opened.

Syntax



Description

cursor-name
Specifies an identifier for a cursor that was previously declared within a PL/SQL context. The specified cursor cannot already be open.

expression
When *cursor-name* is a parameterized cursor, specifies one or more optional actual parameters. The number of actual parameters must match the number of corresponding formal parameters.

Example

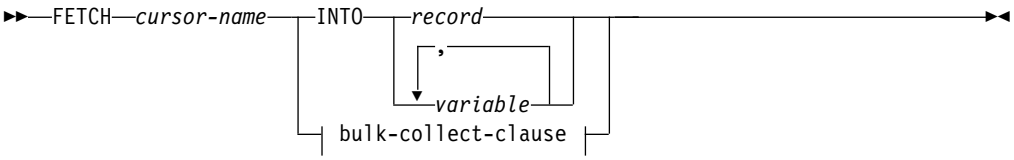
The following example shows an OPEN statement for a cursor that is part of the CURSOR_EXAMPLE procedure:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_3 IS SELECT empno, ename
                        FROM emp
                        WHERE deptno = 10
                        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
        ...
END;
```

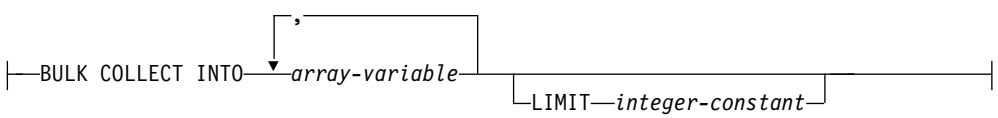
Fetching rows from a cursor (PL/SQL)

The FETCH statement that is required to fetch rows from a PL/SQL cursor is supported by the Db2 data server in PL/SQL contexts.

Syntax



bulk-collect-clause:



Description

cursor-name
Name of a static cursor or cursor variable.

record
Identifier for a previously-defined record. This can be a user-defined record or a record definition that is derived from a table using the %ROWTYPE attribute.

variable
A PL/SQL variable that will hold the field data from the fetched row. One or more variables can be defined, but they must match in order and number the fields that are returned in the select list of the query that was specified in the cursor declaration. The data types of the fields in the select list must match or be implicitly convertible to the data types of the fields in the record or the data types of the variables.

The variable data types can be defined explicitly or by using the %TYPE attribute.

BULK COLLECT INTO *array-variable*
Identifies one or more variables with an array data type. Each row of the result is assigned to an element in each array in the order of the result set, with the array index assigned in sequence.

- If exactly one *array-variable* is specified:
  - If the data type of the *array-variable* element is not a record type, the result row of the cursor must have exactly one column, and the column data type must be assignable to the array element data type.
  - If the data type of the *array-variable* element is a record type, the result row of the cursor must be assignable to the record type.
- If multiple array variables are specified:
  - The data type of the *array-variable* element must not be a record type.
  - There must be an *array-variable* for each column in the result row of the cursor.
  - The data type of each column in the result row of the cursor must be assignable to the array element data type of the corresponding *array-variable*.

If the data type of *array-variable* is an ordinary array, the maximum cardinality must be greater than or equal to the number of rows that are returned by the query, or greater than or equal to the *integer-constant* that is specified in the LIMIT clause.

LIMIT *integer-constant*
Identifies a limit for the number of rows stored in the target array. The cursor position is moved forward *integer-constant* rows or to the end of the result set.

Example

The following example shows a procedure that contains a FETCH statement.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
```

```
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

If the %TYPE attribute is used to define the data type of a target variable, the target variable declaration in a PL/SQL application program does not need to change if the data type of the database column changes. The following example shows a procedure with variables that are defined using the %TYPE attribute.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

If all of the columns in a table are retrieved in the order in which they are defined, the %ROWTYPE attribute can be used to define a record into which the FETCH statement will place the retrieved data. Each field within the record can then be accessed using dot notation. The following example shows a procedure with a record definition that uses %ROWTYPE. This record is used as the target of the FETCH statement.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
     v_emp_rec       emp%ROWTYPE;
     CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
     OPEN emp_cur_1;
     FETCH emp_cur_1 INTO v_emp_rec;
     DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
     DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
END;
```

Closing a cursor (PL/SQL)

After all rows have been retrieved from the result set that is associated with a cursor, the cursor must be closed. The result set cannot be referenced after the cursor has been closed.

However, the cursor can be reopened and the rows of the new result set can be fetched.

Syntax

Description

cursor-name
Specifies an identifier for an open cursor that was previously declared within a PL/SQL context.

Example

The following example shows a CLOSE statement for a cursor that is part of the CURSOR_EXAMPLE procedure:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    FETCH emp_cur_1 INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
    DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
    CLOSE emp_cur_1;
END;
```

Using %ROWTYPE with cursors (PL/SQL)

The %ROWTYPE attribute is used to define a record with fields corresponding to all of the columns that are fetched from a cursor or cursor variable. Each field assumes the data type of its corresponding column.

The %ROWTYPE attribute is prefixed by a cursor name or a cursor variable name. The syntax is `record cursor%ROWTYPE`, where *record* is an identifier that is assigned to the record, and *cursor* is an explicitly declared cursor within the current scope.

The following example shows how to use a cursor with the %ROWTYPE attribute to retrieve department information about each employee in the EMP table.

```
CREATE OR REPLACE PROCEDURE emp_info
IS
    CURSOR empcur IS SELECT ename, deptno FROM emp;
    myvar           empcur%ROWTYPE;
BEGIN
    OPEN empcur;
    LOOP
        FETCH empcur INTO myvar;
        EXIT WHEN empcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( myvar.ename || ' works in department '
            || myvar.deptno );
    END LOOP;
    CLOSE empcur;
END;
```

A call to this procedure (`CALL emp_info;`) returns the following sample output:

```
SMITH works in department 20
ALLEN works in department 30
WARD works in department 30
JONES works in department 20
MARTIN works in department 30
BLAKE works in department 30
CLARK works in department 10
SCOTT works in department 20
KING works in department 10
TURNER works in department 30
ADAMS works in department 20
JAMES works in department 30
FORD works in department 20
MILLER works in department 10
```

Cursor attributes (PL/SQL)

Each cursor has a set of attributes that enables an application program to test the state of the cursor.

These attributes are %ISOPEN, %FOUND, %NOTFOUND, and %ROWCOUNT.

%ISOPEN
This attribute is used to determine whether a cursor is in the open state. When a cursor is passed as a parameter to a function or procedure, it is useful to know (before attempting to open the cursor) whether the cursor is already open.

%FOUND
This attribute is used to determine whether a cursor contains rows after the execution of a FETCH statement. If FETCH statement execution was successful, the %FOUND attribute has a value of true. If FETCH statement execution was not successful, the %FOUND attribute has a value of false. The result is unknown when:

- The value of *cursor-variable-name* is null
- The underlying cursor of *cursor-variable-name* is not open
- The %FOUND attribute is evaluated before the first FETCH statement was executed against the underlying cursor
- FETCH statement execution returns an error

The %FOUND attribute provides an efficient alternative to using a condition handler that checks for the error that is returned when no more rows remain to be fetched.

%NOTFOUND
This attribute is the logical opposite of the %FOUND attribute.

%ROWCOUNT
This attribute is used to determine the number of rows that have been fetched since a cursor was opened.

Table 7 summarizes the attribute values that are associated with certain cursor events.

Table 7. Summary of cursor attribute values

Cursor attribute	%ISOPEN	%FOUND	%NOTFOUND	%ROWCOUNT
Before OPEN	False	Undefined	Undefined	"Cursor not open" exception
After OPEN and before 1st FETCH	True	Undefined	Undefined	0
After 1st successful FETCH	True	True	False	1
After *n*th successful FETCH (last row)	True	True	False	*n*
After *n*+1st FETCH (after last row)	True	False	True	*n*
After CLOSE	False	Undefined	Undefined	"Cursor not open" exception

Cursor variables (PL/SQL)

A *cursor variable* is a cursor that contains a pointer to a query result set. The result set is determined by execution of the OPEN FOR statement using the cursor variable.

A cursor variable, unlike a static cursor, is not associated with a particular query. The same cursor variable can be opened a number of times with separate OPEN FOR statements containing different queries. A new result set is created each time and made available through the cursor variable.

SYS_REFCURSOR cursor variables (PL/SQL)

The Db2 data server supports the declaration of cursor variables of the SYS_REFCURSOR built-in data type, which can be associated with any result set.

The SYS_REFCURSOR data type is known as a weakly-typed REF CURSOR type. Strongly-typed cursor variables of the REF CURSOR type require a result set specification.

Syntax

Description

cursor-variable-name
Specifies an identifier for the cursor variable.

SYS_REFCURSOR
Specifies that the data type of the cursor variable is the built-in SYS_REFCURSOR data type.

Example

The following example shows a SYS_REFCURSOR variable declaration:

```
DECLARE emprefcur SYS_REFCURSOR;
```

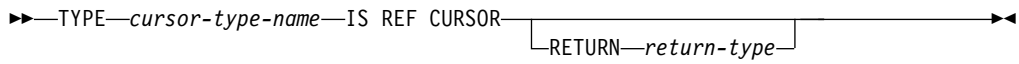
User-defined REF CURSOR type variables (PL/SQL)

The Db2 data server supports the user-defined REF CURSOR data type and cursor variable declarations.

The user-defined REF CURSOR type can be defined by executing the TYPE declaration in a PL/SQL context. After the type has been defined, you can declare a cursor variable of that type.

Restriction: REF CURSOR types can be declared only within a package and are not supported in routines, triggers, or anonymous blocks.

Syntax



Description

TYPE *cursor-type-name*
Specifies an identifier for the cursor data type.

IS REF CURSOR
Specifies that the cursor is of a user-defined REF CURSOR data type.

RETURN *return-type*
Specifies the return type that is associated with the cursor. If a *return-type* is specified, this REF CURSOR type is strongly typed; otherwise, it is weakly typed.

Example

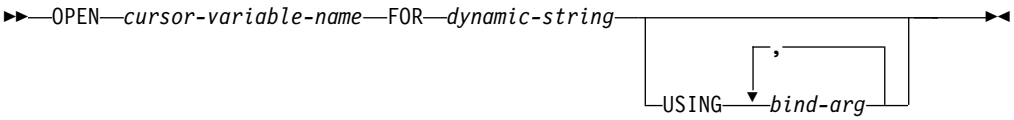
The following example shows a cursor variable declaration in a package:

```
CREATE OR REPLACE PACKAGE my_pkg
AS
TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
my_rec emp_cur_type;
END my_pkg
```

Dynamic queries with cursor variables (PL/SQL)

The Db2 data server supports dynamic queries through the OPEN FOR statement in PL/SQL contexts.

Syntax



Description

OPEN *cursor-variable-name*
Specifies an identifier for a cursor variable that was previously declared within a PL/SQL context.

FOR *dynamic-string*
Specifies a string literal or string variable that contains a SELECT statement (without the terminating semicolon). The statement can contain named parameters, such as, for example, *:param1*.

USING *bind-arg*
Specifies one or more bind arguments whose values are substituted for placeholders in *dynamic-string* when the cursor opens.

Examples

The following example shows a dynamic query that uses a string literal:

```
CREATE OR REPLACE PROCEDURE dept_query
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = 30' ||
        ' AND sal >= 1500';
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
```

```
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following example output is generated by the DEPT_QUERY procedure:

```
CALL dept_query;

EMPNO    ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

The query in the previous example can be modified with bind arguments to pass the query parameters:

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = :dept'
        || ' AND sal >= :sal' USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following CALL statement generates the same output that was generated in the previous example:

```
CALL dept_query(30, 1500);
```

A string variable to pass the SELECT statement provides the most flexibility:

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    p_query_string  VARCHAR2(100);
BEGIN
    p_query_string := 'SELECT empno, ename FROM emp WHERE ' ||
        'deptno = :dept AND sal >= :sal';
    OPEN emp_refcur FOR p_query_string USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
```

```
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

This version of the DEPT_QUERY procedure generates the following example output:

```
CALL dept_query(20, 1500);

EMPNO    ENAME
-----    -------
7566     JONES
7788     SCOTT
7902     FORD
```

Example: Returning a REF CURSOR from a procedure (PL/SQL)

This example demonstrates how to define and open a REF CURSOR variable, and then pass it as a procedure parameter.

The cursor variable is specified as an IN OUT parameter so that the result set is made available to the caller of the procedure:

```
CREATE OR REPLACE PROCEDURE emp_by_job (
    p_job           VARCHAR2,
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
END;
```

The EMP_BY_JOB procedure is invoked in the following anonymous block by assigning the procedure's IN OUT parameter to a cursor variable that was declared in the anonymous block's declaration section. The result set is fetched using this cursor variable.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE := 'SALESMAN';
    v_emp_refcur    SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES WITH JOB ' || v_job);
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    emp_by_job(v_job, v_emp_refcur);
    LOOP
        FETCH v_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN v_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE v_emp_refcur;
END;
```

The following example output is generated when the anonymous block executes:

```
EMPLOYEES WITH JOB SALESMAN
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
```

Example: Modularizing cursor operations (PL/SQL)

This example demonstrates how various operations on cursor variables can be modularized into separate programs or PL/SQL components.

The following example shows a procedure that opens a cursor variable whose query retrieves all rows in the EMP table:

```
CREATE OR REPLACE PROCEDURE open_all_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp;
END;
```

In the next example, a procedure opens a cursor variable whose query retrieves all rows for a given department:

```
CREATE OR REPLACE PROCEDURE open_emp_by_dept (
    p_emp_refcur    IN OUT SYS_REFCURSOR,
    p_deptno        emp.deptno%TYPE
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp
        WHERE deptno = p_deptno;
END;
```

The following example shows a procedure that opens a cursor variable whose query retrieves all rows in the DEPT table:

```
CREATE OR REPLACE PROCEDURE open_dept (
    p_dept_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_dept_refcur FOR SELECT deptno, dname FROM dept;
END;
```

In the next example, a procedure fetches and displays a cursor variable result set consisting of employee number and name:

```
CREATE OR REPLACE PROCEDURE fetch_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH p_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN p_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
END;
```

The following example shows a procedure that fetches and displays a cursor variable result set consisting of department number and name:

```
CREATE OR REPLACE PROCEDURE fetch_dept (
    p_dept_refcur   IN SYS_REFCURSOR
)
IS
    v_deptno        dept.deptno%TYPE;
```

```
    v_dname         dept.dname%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('DEPT   DNAME');
    DBMS_OUTPUT.PUT_LINE('----   ---------');
    LOOP
        FETCH p_dept_refcur INTO v_deptno, v_dname;
        EXIT WHEN p_dept_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_deptno || '     ' || v_dname);
    END LOOP;
END;
```

The following example shows a procedure that closes a cursor variable:

```
CREATE OR REPLACE PROCEDURE close_refcur (
    p_refcur        IN OUT SYS_REFCURSOR
)
IS
BEGIN
    CLOSE p_refcur;
END;
```

The following example shows an anonymous block that executes these procedures:

```
DECLARE
    gen_refcur      SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('ALL EMPLOYEES');
    open_all_emp(gen_refcur);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('***************');

    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #10');
    open_emp_by_dept(gen_refcur, 10);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('***************');

    DBMS_OUTPUT.PUT_LINE('DEPARTMENTS');
    open_dept(gen_refcur);
    fetch_dept(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('****************');

    close_refcur(gen_refcur);
END;
```

The following example output is generated when the anonymous block executes:

```
ALL EMPLOYEES
EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
***************
EMPLOYEES IN DEPT #10
EMPNO    ENAME
-----    -------
7782     CLARK
```

```
7839     KING
7934     MILLER
****************
DEPARTMENTS
DEPT   DNAME
----   ---------
10     ACCOUNTING
20     RESEARCH
30     SALES
40     OPERATIONS
*****************
```

Triggers (PL/SQL)

A PL/SQL trigger is a named database object that encapsulates and defines a set of actions that are to be performed in response to an insert, update, or delete operation against a table. Triggers are created using the PL/SQL CREATE TRIGGER statement.

Types of triggers (PL/SQL)

The Db2 data server supports row-level and statement-level triggers within a PL/SQL context.

A *row-level trigger* fires once for each row that is affected by a triggering event. For example, if deletion is defined as a triggering event for a particular table, and a single DELETE statement deletes five rows from that table, the trigger fires five times, once for each row.

A *statement-level trigger* fires only once for each statement. Using the previous example, if deletion is defined as a triggering event for a particular table, and a single DELETE statement deletes five rows from that table, the trigger fires once. Statement-level trigger granularity cannot be specified for BEFORE triggers or INSTEAD OF triggers.

The trigger code block is executed either before or after each row is affected by the triggering statement, except for INSTEAD OF triggers which execute the trigger code block instead of affecting each row based on the activating statement.

Trigger variables (PL/SQL)

NEW and OLD are special variables that you can use with PL/SQL triggers without explicitly defining them.

- NEW is a pseudo-record name that refers to the new table row for insert and update operations in row-level triggers. Its usage is `:NEW.column`, where *column* is the name of a column in the table on which the trigger is defined.
  - When used in a *before row-level trigger*, the initial content of :NEW.*column* is the column value in the new row that is to be inserted or in the row that is to replace the old row.
  - When used in an *after row-level trigger*, the new column value has already been stored in the table.
  - When a trigger is activated by a DELETE operation, the :NEW.*column* used in that trigger is null.

  In the trigger code block, :NEW.*column* can be used like any other variable. If a value is assigned to :NEW.*column* in the code block of a before row-level trigger, the assigned value is used in the inserted or updated row.

- OLD is a pseudo-record name that refers to the old table row for update and delete operations in row-level triggers. Its usage is `:OLD.column`, where *column* is the name of a column in the table on which the trigger is defined.
  - When used in a *before row-level trigger*, the initial content of :OLD.*column* is the column value in the row that is to be deleted or in the old row that is to be replaced by the new row.
  - When used in an *after row-level trigger*, the old column value is no longer stored in the table.
  - When a trigger is activated by an INSERT operation, the :OLD.*column* used in that trigger is null.

  In the trigger code block, :OLD.*column* can be used like any other variable. If a value is assigned to :OLD.*column* in the code block of a before row-level trigger, the assigned value has no affect on the action of the trigger.

Trigger event predicates (PL/SQL)

The trigger event predicates, UPDATING, DELETING, and INSERTING can only be used in a trigger to identify the event that activated the trigger.

DELETING
True if the trigger was activated by a delete operation. False otherwise.

INSERTING
True if the trigger was activated by an insert operation. False otherwise.

UPDATING
True if the trigger was activated by an update operation. False otherwise.

These predicates can be specified as a single search condition, or as a boolean factor within a complex search condition in a WHEN clause or PL/SQL statement.

Transactions and exceptions (PL/SQL)

A trigger is always executed as part of the same transaction within which the triggering statement is executing.

If no exceptions occur within the trigger code block, the effects of data manipulation language (DML) within the trigger are committed only if the transaction that contains the triggering statement commits. If the transaction is rolled back, the effects of DML within the trigger are also rolled back.

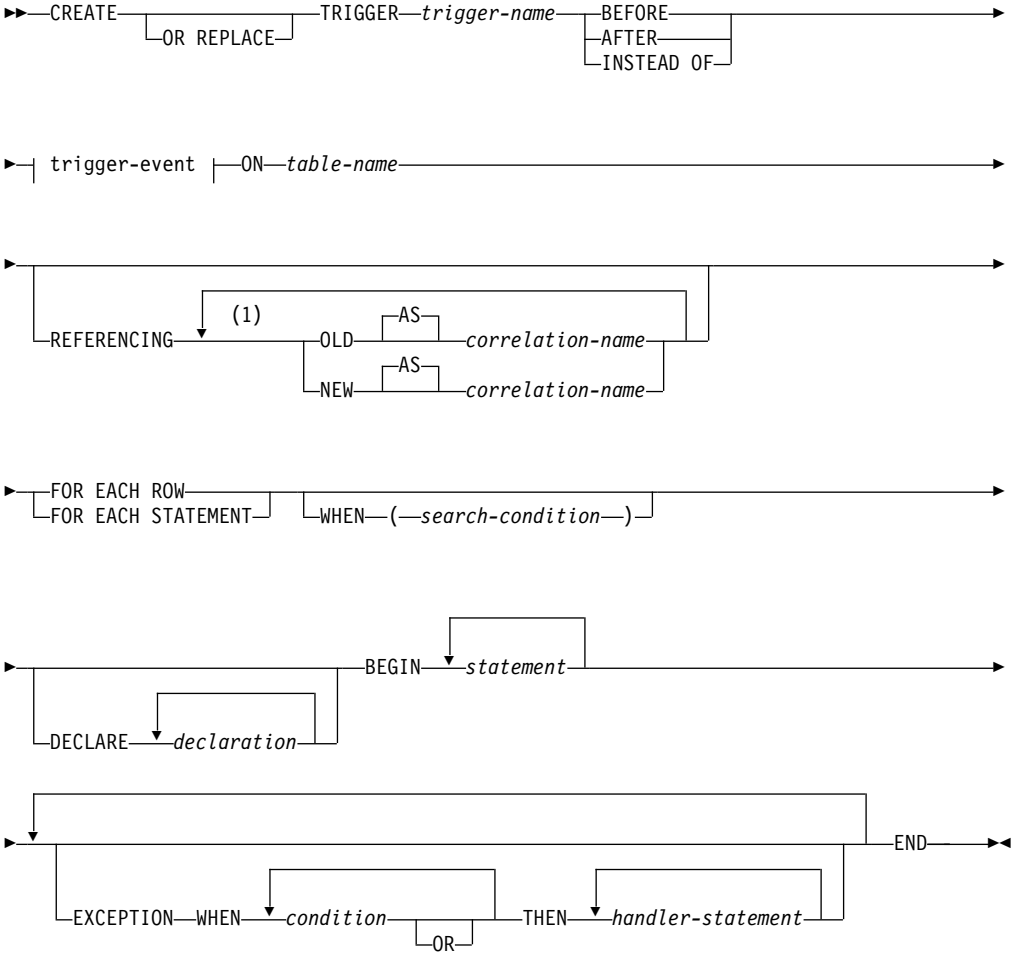
A Db2 rollback can only occur within an atomic block or by using an UNDO handler. The triggering statement itself is not rolled back unless the application forces a rollback of the encapsulating transaction.

If an unhandled exception occurs within the trigger code block, the calling statement is rolled back.

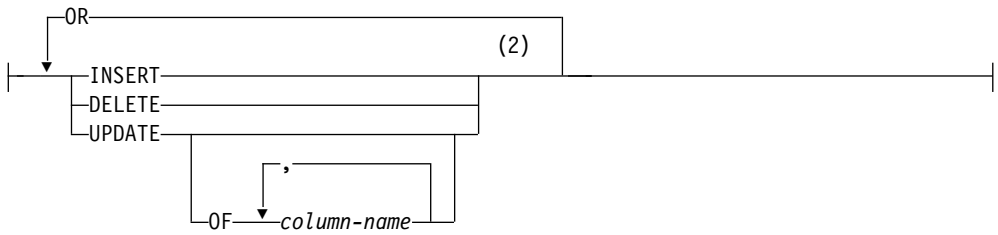
CREATE TRIGGER statement (PL/SQL)

The CREATE TRIGGER statement defines a PL/SQL trigger in the database.

Syntax



trigger-event:



Notes:

1 OLD and NEW can only be specified once each.

2 A trigger event must not be specified more than once for the same operation. For example, INSERT OR DELETE is allowed, but INSERT OR INSERT is not allowed.

Description

OR REPLACE
Specifies to replace the definition for the trigger if one exists at the current

server. The existing definition is effectively dropped before the new definition is replaced in the catalog. This option is ignored if a definition for the trigger does not exist at the current server.

trigger-name
Names the trigger. The name, including the implicit or explicit schema name, must not identify a trigger already described in the catalog (SQLSTATE 42710). If a two-part name is specified, the schema name cannot begin with 'SYS' (SQLSTATE 42939).

BEFORE
Specifies that the associated triggered action is to be applied before any changes caused by the actual update of the subject table are applied to the database.

AFTER
Specifies that the associated triggered action is to be applied after the changes caused by the actual update of the subject table are applied to the database.

INSTEAD OF
Specifies that the associated triggered action replaces the action against the subject view.

trigger-event
Specifies that the triggered action associated with the trigger is to be executed whenever one of the events is applied to the subject table. Any combination of the events can be specified, but each event (INSERT, DELETE, and UPDATE) can only be specified once (SQLSTATE 42613).

INSERT
Specifies that the triggered action associated with the trigger is to be executed whenever an INSERT operation is applied to the subject table.

DELETE
Specifies that the triggered action associated with the trigger is to be executed whenever a DELETE operation is applied to the subject table.

UPDATE
Specifies that the triggered action associated with the trigger is to be executed whenever an UPDATE operation is applied to the subject table, subject to the columns specified or implied.

If the optional *column-name* list is not specified, every column of the table is implied. Therefore, omission of the *column-name* list implies that the trigger will be activated by the update of any column of the table.

OF *column-name*,...
Each *column-name* specified must be a column of the base table (SQLSTATE 42703). If the trigger is a BEFORE trigger, the *column-name* specified cannot be a generated column other than the identity column (SQLSTATE 42989). No *column-name* can appear more than once in the *column-name* list (SQLSTATE 42711). The trigger will only be activated by the update of a column that is identified in the *column-name* list. This clause cannot be specified for an INSTEAD OF trigger (SQLSTATE 42613).

ON *table-name*
Designates the subject table of the BEFORE trigger or AFTER trigger definition. The name must specify a base table or an alias that resolves to a base table (SQLSTATE 42704 or 42809). The name must not specify a catalog table

(SQLSTATE 42832), a materialized query table (SQLSTATE 42997), a created temporary table, a declared temporary table (SQLSTATE 42995), or a nickname (SQLSTATE 42809).

REFERENCING
Specifies the correlation names for the *transition variables*. Correlation names identify a specific row in the set of rows affected by the triggering SQL operation. Each row affected by the triggering SQL operation is available to the triggered action by qualifying columns with *correlation-names* specified as follows.

OLD AS *correlation-name*
Specifies a correlation name that identifies the row state prior to the triggering SQL operation. If the trigger event is INSERT, the values in the row are null values.

NEW AS *correlation-name*
Specifies a correlation name that identifies the row state as modified by the triggering SQL operation and by any SET statement in a BEFORE trigger that has already executed. If the trigger event is DELETE, the values in the row are null values.

If the REFERENCING clause is not invoked, then trigger variables NEW and OLD can optionally be used without explicitly defining them.

FOR EACH ROW
Specifies that the triggered action is to be applied once for each row of the subject table that is affected by the triggering SQL operation.

FOR EACH STATEMENT
Specifies that the triggered action is to be applied only once for the whole statement.

WHEN

(*search-condition***)**
Specifies a condition that is true, false, or unknown. The *search-condition* provides a capability to determine whether or not a certain triggered action should be executed. The associated action is performed only if the specified search condition evaluates as true.

declaration
Specifies a variable declaration.

statement **or** *handler-statement*
Specifies a PL/SQL program statement. The trigger body can contain nested blocks.

condition
Specifies an exception condition name, such as NO_DATA_FOUND.

Example

The following example shows a before row-level trigger that calculates the commission of every new employee belonging to department 30 before a record for that employee is inserted into the EMP table. It also records any salary increases that exceed 50% in an exception table:

```
CREATE TABLE emp (
    name            VARCHAR2(10),
    deptno          NUMBER,
    sal             NUMBER,
```

```
    comm            NUMBER
)
/

CREATE TABLE exception (
    name            VARCHAR2(10),
    old_sal         NUMBER,
    new_sal         NUMBER
)
/

CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW
BEGIN
    IF (:NEW.deptno = 30 and INSERTING) THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;

    IF (UPDATING and (:NEW.sal - :OLD.sal) > :OLD.sal * .5) THEN
       INSERT INTO exception VALUES (:NEW.name, :OLD.sal, :NEW.sal);
    END IF;
END
/
```

Dropping triggers (PL/SQL)

You can remove a trigger from the database by using the DROP TRIGGER statement.

Syntax

Description

trigger-name
Specifies the name of the trigger that is to be dropped.

Examples: Triggers (PL/SQL)

PL/SQL trigger definitions can be compiled by the Db2 data server. These examples will help you to create valid triggers and to troubleshoot PL/SQL trigger compilation errors.

Before row-level triggers

The following example shows a before row-level trigger that calculates the commission of every new employee belonging to department 30 before a record for that employee is inserted into the EMP table:

```
CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
BEGIN
    IF :NEW.deptno = 30 THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;
END;
```

The trigger computes the commissions for two new employees and inserts those values as part of the new employee rows:

```
INSERT INTO emp VALUES (9005,'ROBERS','SALESMAN',7782,SYSDATE,3000.00,NULL,30);

INSERT INTO emp VALUES (9006,'ALLEN','SALESMAN',7782,SYSDATE,4500.00,NULL,30);

SELECT * FROM emp WHERE empno IN (9005, 9006);

     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
      9005 ROBERS     SALESMAN        7782 01-APR-05       3000       1200         30
      9006 ALLEN      SALESMAN        7782 01-APR-05       4500       1800         30
```

After row-level triggers

The following example shows three after row-level triggers.

- When a new employee row is inserted into the EMP table, one trigger (EMP_INS_TRIG) adds a new row to the JOBHIST table for that employee and adds a row to the EMPCHGLOG table with a description of the action.
- When an existing employee row is updated, the second trigger (EMP_CHG_TRIG) sets the ENDDATE column of the latest JOBHIST row (assumed to be the one with a null ENDDATE) to the current date and inserts a new JOBHIST row with the employee's new information. This trigger also adds a row to the EMPCHGLOG table with a description of the action
- When an employee row is deleted from the EMP table, the third trigger (EMP_DEL_TRIG) adds a row to the EMPCHGLOG table with a description of the action.

```
CREATE TABLE empchglog (
    chg_date        DATE,
    chg_desc        VARCHAR2(30)
);
CREATE OR REPLACE TRIGGER emp_ins_trig
    AFTER INSERT ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    v_action := 'Added';
    v_empno := :NEW.empno;
    v_deptno := :NEW.deptno;
    INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
        :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, 'New Hire');

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;

CREATE OR REPLACE TRIGGER emp_chg_trig
    AFTER UPDATE ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    v_action := 'Updated';
    v_empno := :NEW.empno;
    v_deptno := :NEW.deptno;
    v_chgdesc := '';
    IF NVL(:OLD.ename, '-null-') != NVL(:NEW.ename, '-null-') THEN
```

```
        v_chgdesc := v_chgdesc || 'name, ';
    END IF;
    IF NVL(:OLD.job, '-null-') != NVL(:NEW.job, '-null-') THEN
        v_chgdesc := v_chgdesc || 'job, ';
    END IF;
    IF NVL(:OLD.sal, -1) != NVL(:NEW.sal, -1) THEN
        v_chgdesc := v_chgdesc || 'salary, ';
    END IF;
    IF NVL(:OLD.comm, -1) != NVL(:NEW.comm, -1) THEN
        v_chgdesc := v_chgdesc || 'commission, ';
    END IF;
    IF NVL(:OLD.deptno, -1) != NVL(:NEW.deptno, -1) THEN
        v_chgdesc := v_chgdesc || 'department, ';
    END IF;
    v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ', ');
    UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
        AND enddate IS NULL;
    INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
        :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, v_chgdesc);

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;

CREATE OR REPLACE TRIGGER emp_del_trig
    AFTER DELETE ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    v_action := 'Deleted';
    v_empno := :OLD.empno;
    v_deptno := :OLD.deptno;

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;
```

In the following example, two employee rows are added using two separate INSERT statements, and then both rows are updated using a single UPDATE statement. The JOBHIST table shows the action of the trigger for each affected row: two new hire entries for the two new employees and two changed commission records. The EMPCHGLOG table also shows that the trigger was fired a total of four times, once for each action against the two rows.

```
INSERT INTO emp VALUES (9003,'PETERS','ANALYST',7782,SYSDATE,5000.00,NULL,40);

INSERT INTO emp VALUES (9004,'AIKENS','ANALYST',7782,SYSDATE,4500.00,NULL,40);

UPDATE emp SET comm = sal * 1.1 WHERE empno IN (9003, 9004);

SELECT * FROM jobhist WHERE empno IN (9003, 9004);

     EMPNO STARTDATE ENDDATE   JOB             SAL       COMM     DEPTNO CHGDESC
---------- --------- --------- --------- ---------- ---------- ---------- ------------------
      9003 31-MAR-05 31-MAR-05 ANALYST         5000                    40 New Hire
      9004 31-MAR-05 31-MAR-05 ANALYST         4500                    40 New Hire
      9003 31-MAR-05           ANALYST         5000       5500         40 Changed commission
      9004 31-MAR-05           ANALYST         4500       4950         40 Changed commission

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
```

After both employees are deleted with a single DELETE statement, the EMPCHGLOG table shows that the trigger was fired twice, once for each deleted employee:

```
DELETE FROM emp WHERE empno IN (9003, 9004);

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
31-MAR-05 Deleted employee # 9003
31-MAR-05 Deleted employee # 9004
```

Packages (PL/SQL)

PL/SQL package definitions are supported by the Db2 data server. A PL/SQL package is a named collection of functions, procedures, variables, cursors, user-defined types, and records that are referenced using a common qualifier, the package name.

Packages have the following characteristics:

- Packages provide a convenient way of organizing the functions and procedures that have a related purpose. Permission to use the package functions and procedures is dependent upon one privilege that is granted to the entire package.
- Certain items in a package can be declared public. Public entities are visible and can be referenced by other programs that hold the EXECUTE privilege on the package. In the case of public functions and procedures, only their signatures are visible. The PL/SQL code for these function and procedures is not accessible to others; therefore, applications that utilize such a package are dependent upon only the information that is available in the signatures.
- Other items in a package can be declared private. Private entities can be referenced and used by functions and procedures within the package, but not by external applications.

Package components (PL/SQL)

Packages consist of two main components: the package specification and the package body.

- The *package specification* is the public interface, comprising the elements that can be referenced outside of the package. A package specification is created by executing the CREATE PACKAGE statement.
- The *package body* contains the actual implementation of all of the procedures and functions that are declared within the package specification, as well as any declaration of private types, variables, and cursors. A package body is created by executing the CREATE PACKAGE BODY statement.

Creating packages (PL/SQL)

Creating a package specification enables you to encapsulate related data type, procedure, and function definitions within a single context in the database.

Packages are extensions of schemas that provide namespace support for the objects that they reference. They are repositories in which executable code can be defined.

Using a package involves referencing or executing objects that are defined in the package specification and implemented within the package.

Creating package specifications (PL/SQL)

A package specification establishes which package objects can be referenced from outside of the package. Objects that can be referenced from outside of a package are called the public elements of that package.

The following example shows how to create a package specification named EMP_ADMIN, consisting of two functions and two stored procedures.

```
CREATE OR REPLACE PACKAGE emp_admin
IS

   FUNCTION get_dept_name (
      p_deptno        NUMBER DEFAULT 10
   )
   RETURN VARCHAR2;
   FUNCTION update_emp_sal (
      p_empno         NUMBER,
      p_raise         NUMBER
   )
   RETURN NUMBER;
   PROCEDURE hire_emp (
      p_empno         NUMBER,
      p_ename         VARCHAR2,
      p_job           VARCHAR2,
      p_sal           NUMBER,
      p_hiredate      DATE DEFAULT sysdate,
      p_comm          NUMBER DEFAULT 0,
      p_mgr           NUMBER,
      p_deptno        NUMBER DEFAULT 10
   );
   PROCEDURE fire_emp (
      p_empno         NUMBER
   );

END emp_admin;
```

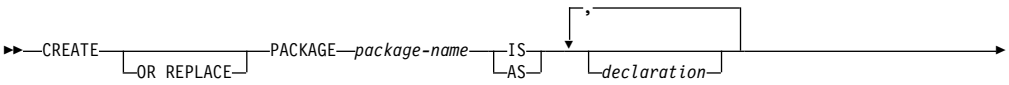
Attention:

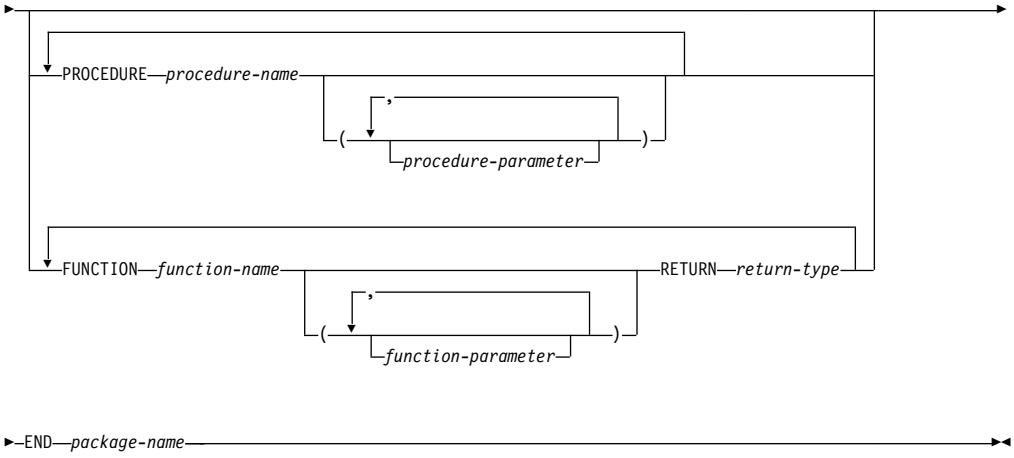
When you use **CREATE** or **REPLACE** syntax for package specification of package whose body was created by a **CREATE PACKAGE BODY** statement, the existing package body is dropped and needs to be re-created before any of package objects are invoked.

CREATE PACKAGE statement (PL/SQL)

The CREATE PACKAGE statement creates a package specification, which defines the interface to a package.

Syntax





Description

package-name
Specifies an identifier for the package.

declaration
Specifies an identifier for a public item. The public item can be accessed from outside of the package using the syntax *package-name.item-name*. There can be zero or more public items. Public item declarations must come before procedure or function declarations. The *declaration* can be any of the following:

- Collection declaration
- EXCEPTION declaration
- Record declaration
- REF CURSOR and cursor variable declaration
- TYPE definition for a collection, record, or REF CURSOR type variable
- SUBTYPE definition
- Variable declaration

procedure-name
Specifies an identifier for a public procedure. The public procedure can be invoked from outside of the package using the syntax *package-name.procedure-name*().

procedure-parameter
Specifies an identifier for a formal parameter of the procedure.

function-name
Specifies an identifier for a public function. The public function can be invoked from outside of the package using the syntax *package-name.function-name*().

function-parameter
Specifies an identifier for a formal parameter of the function. Input (IN mode) parameters can be initialized with a default value.

return-type
Specifies a data type for the value that is returned by the function.

Notes

The CREATE PACKAGE statement can be submitted in obfuscated form. In an obfuscated statement, only the package name is readable. The rest of the statement is encoded in such a way that it is not readable, but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

Creating the package body (PL/SQL)

A package body contains the implementation of all of the procedures and functions that are declared within the package specification.

The following example shows how to create a package body for the EMP_ADMIN package specification.

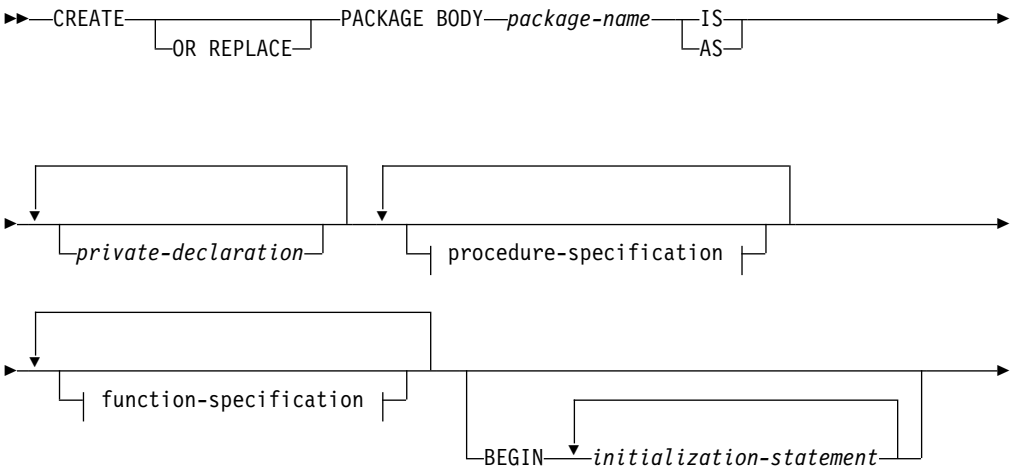
```
--
--  Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
   --
   --  Function that queries the 'dept' table based on the department
   --  number and returns the corresponding department name.
   --
   FUNCTION get_dept_name (
      p_deptno        IN NUMBER DEFAULT 10
   )
   RETURN VARCHAR2
   IS
      v_dname         VARCHAR2(14);
   BEGIN
      SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
      RETURN v_dname;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
         RETURN '';
   END;
   --
   --  Function that updates an employee's salary based on the
   --  employee number and salary increment/decrement passed
   --  as IN parameters.  Upon successful completion the function
   --  returns the new updated salary.
   --
   FUNCTION update_emp_sal (
      p_empno         IN NUMBER,
      p_raise         IN NUMBER
   )
   RETURN NUMBER
   IS
      v_sal           NUMBER := 0;
   BEGIN
      SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
      v_sal := v_sal + p_raise;
      UPDATE emp SET sal = v_sal WHERE empno = p_empno;
      RETURN v_sal;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
         RETURN -1;
      WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
         DBMS_OUTPUT.PUT_LINE(SQLERRM);
         DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
         DBMS_OUTPUT.PUT_LINE(SQLCODE);
```

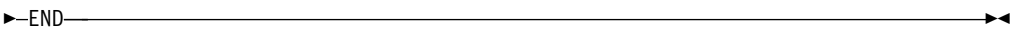
```
      RETURN -1;
   END;
   --
   --  Procedure that inserts a new employee record into the 'emp' table.
   --
   PROCEDURE hire_emp (
      p_empno         NUMBER,
      p_ename         VARCHAR2,
      p_job           VARCHAR2,
      p_sal           NUMBER,
      p_hiredate      DATE    DEFAULT sysdate,
      p_comm          NUMBER  DEFAULT 0,
      p_mgr           NUMBER,
      p_deptno        NUMBER  DEFAULT 10
   )
   AS
   BEGIN
      INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
         VALUES(p_empno, p_ename, p_job, p_sal,
                p_hiredate, p_comm, p_mgr, p_deptno);
   END;
   --
   --  Procedure that deletes an employee record from the 'emp' table based
   --  on the employee number.
   --
   PROCEDURE fire_emp (
      p_empno         NUMBER
   )
   AS
   BEGIN
      DELETE FROM emp WHERE empno = p_empno;
   END;
END;
```

CREATE PACKAGE BODY statement (PL/SQL)

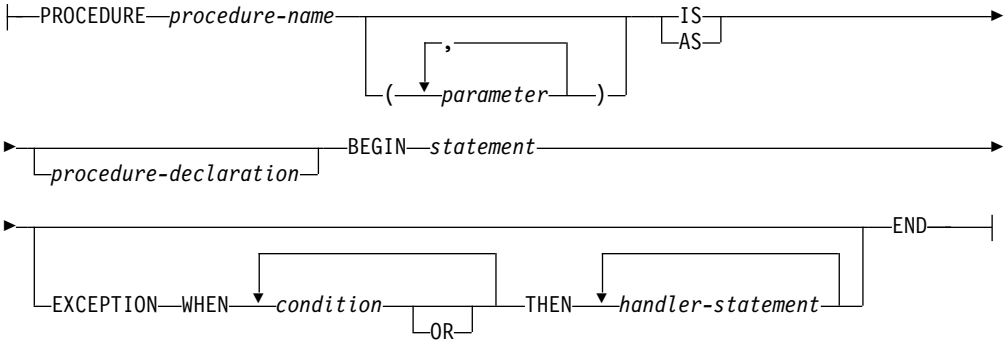
The CREATE PACKAGE BODY statement creates a package body, which contains the implementation of all of the procedures and functions that are declared within the package specification, as well as any declaration of private types, variables, and cursors.

Syntax

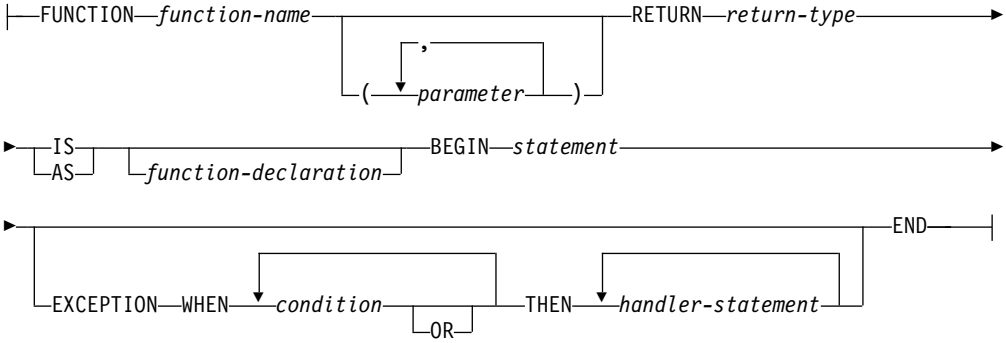




procedure-specification:



function-specification:



Description

package-name
Specifies the name of the package whose body is to be created. A package specification with the same name must exist.

private-declaration
Specifies the name of a private object that can be accessed by any procedure or function within the package. There can be zero or more private variables. The *private-declaration* can be any of the following:

- Variable declaration
- Record declaration
- Collection declaration
- REF CURSOR and cursor variable declaration
- TYPE definitions for records, collections, or variables of the REF CURSOR type
- SUBTYPE definition over a base type

procedure-name
Specifies the name of a public procedure that is declared in the package specification and its signature. The signature can specify any one of the following: the formal parameter names, data types, parameter modes, the order

of the formal parameters, or the number of formal parameters. When the procedure name and package specification exactly match the signature of the public procedure's declaration, *procedure-name* defines the body of this public procedure.

If none of these conditions is true, *procedure-name* defines a new private procedure.

parameter
Specifies a formal parameter of the procedure.

procedure-declaration
Specifies a declaration that can be accessed only from within procedure *procedure-name*. This is a PL/SQL statement.

statement
Specifies a PL/SQL program statement.

function-name
Specifies the name of a public function that is declared in the package specification and its signature. The signature can specify any one of the following: the formal parameter names, data types, parameter modes, the order of the formal parameters, or the number of formal parameters. When the function name and package specification exactly match the signature of the public function's declaration, *function-name* defines the body of this public function.

If none of these conditions is true, *function-name* defines a new private function.

parameter
Specifies a formal parameter of the function.

return-type
Specifies the data type of the value that is returned by the function.

function-declaration
Specifies a declaration that can be accessed only from within function *function-name*. This is a PL/SQL statement.

statement
Specifies a PL/SQL program statement.

initialization-statement
Specifies a statement in the initialization section of the package body. The initialization section, if specified, must contain at least one statement. The statements in the initialization section are executed once per user session when the package is first referenced.

Notes

The CREATE PACKAGE BODY statement can be submitted in obfuscated form. In an obfuscated statement, only the package name is readable. The rest of the statement is encoded in such a way that it is not readable, but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

Referencing package objects (PL/SQL)

References to objects that are defined within a package must sometimes be qualified with the package name.

To reference the objects that are declared in a package specification, specify the package name, a period character, and then the name of the object. If the package is not defined in the current schema, specify the schema name as well. For example:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
schema.package_name.subprogram_name
```

Example

The following example contains a reference to a function named GET_DEPT_NAME that is defined in a package named EMP_ADMIN:

```
select emp_admin.get_dept_name(10) from dept
```

Packages with user-defined types (PL/SQL)

User-defined types can be declared and referenced in packages.

The following example shows a package specification for the EMP_RPT package. This definition includes the following declarations:

- A publicly accessible record type, EMPREC_TYP
- A publicly accessible weakly-typed REF CURSOR type, EMP_REFCUR
- A publicly accessible subtype, DEPT_NUM, restricted to the range of values from 1 to 99
- Two functions, GET_DEPT_NAME and OPEN_EMP_BY_DEPT; both functions have an input parameter of the subtype DEPT_NUM; the latter function returns the REF CURSOR type EMP_REFCUR
- Two procedures, FETCH_EMP and CLOSE_REFCUR; both declare a weakly-typed REF CURSOR type as a formal parameter

```
CREATE OR REPLACE PACKAGE emp_rpt
IS
    TYPE emprec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR(10)
    );
    TYPE emp_refcur IS REF CURSOR;
    SUBTYPE dept_num IS dept.deptno%TYPE RANGE 1..99;

    FUNCTION get_dept_name (
        p_deptno    IN dept_num
    ) RETURN VARCHAR2;
    FUNCTION open_emp_by_dept (
        p_deptno    IN dept_num
    ) RETURN EMP_REFCUR;
    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    );
    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    );
END emp_rpt;
```

The definition of the associated package body includes the following private variable declarations:

- A static cursor, DEPT_CUR
- An associative array type, DEPTTAB_TYP
- An associative array variable, T_DEPT

- An integer variable, T_DEPT_MAX
- A record variable, R_EMP

```
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
    CURSOR dept_cur IS SELECT * FROM dept;
    TYPE depttab_typ IS TABLE of dept%ROWTYPE
        INDEX BY BINARY_INTEGER;
    t_dept          DEPTTAB_TYP;
    t_dept_max      INTEGER := 1;
    r_emp           EMPREC_TYP;

    FUNCTION get_dept_name (
        p_deptno    IN dept_num
    ) RETURN VARCHAR2
    IS
    BEGIN
        FOR i IN 1..t_dept_max LOOP
            IF p_deptno = t_dept(i).deptno THEN
                RETURN t_dept(i).dname;
            END IF;
        END LOOP;
        RETURN 'Unknown';
    END;

    FUNCTION open_emp_by_dept(
        p_deptno    IN dept_num
    ) RETURN EMP_REFCUR
    IS
        emp_by_dept EMP_REFCUR;
    BEGIN
        OPEN emp_by_dept FOR SELECT empno, ename FROM emp
            WHERE deptno = p_deptno;
        RETURN emp_by_dept;
    END;

    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
        DBMS_OUTPUT.PUT_LINE('-----    -------');
        LOOP
            FETCH p_refcur INTO r_emp;
            EXIT WHEN p_refcur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' || r_emp.ename);
        END LOOP;
    END;

    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        CLOSE p_refcur;
    END;
BEGIN
    OPEN dept_cur;
    LOOP
        FETCH dept_cur INTO t_dept(t_dept_max);
        EXIT WHEN dept_cur%NOTFOUND;
        t_dept_max := t_dept_max + 1;
    END LOOP;
    CLOSE dept_cur;
    t_dept_max := t_dept_max - 1;
END emp_rpt;
```

This package contains an initialization section that loads the private associative array variable T_DEPT, using the private static cursor DEPT_CUR. T_DEPT serves as a department name lookup table in function GET_DEPT_NAME. The function OPEN_EMP_BY_DEPT returns a REF CURSOR variable for the result set of employee numbers and names for a given department. This REF CURSOR variable can then be passed to procedure FETCH_EMP to retrieve and list the individual rows of the result set. Finally, procedure CLOSE_REFCUR can be used to close the REF CURSOR variable that is associated with this result set.

The following anonymous block runs the package functions and procedures. The declaration section includes the declaration of a scalar variable V_DEPTNO, using the public SUBTYPE DEPT_NUM and a cursor variable V_EMP_CUR, using the public REF CURSOR type, EMP_REFCUR. V_EMP_CUR contains a pointer to the result set that is passed between the package function and procedures.

```
DECLARE
    v_deptno        emp_rpt.DEPT DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    emp_rpt.fetch_emp(v_emp_cur);
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    emp_rpt.close_refcur(v_emp_cur);
END;
```

This anonymous block produces the following sample output:

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

The following anonymous block shows another way of achieving the same result. Instead of using the package procedures FETCH_EMP and CLOSE_REFCUR, the logic is coded directly into the anonymous block. Note the declaration of record variable R_EMP, using the public record type EMPREC_TYP.

```
DECLARE
    v_deptno        emp_rpt.DEPT DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
    r_emp           emp_rpt.EMPREC_TYP;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH v_emp_cur INTO r_emp;
        EXIT WHEN v_emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' ||
            r_emp.ename);
    END LOOP;
```

```
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    CLOSE v_emp_cur;
END;
```

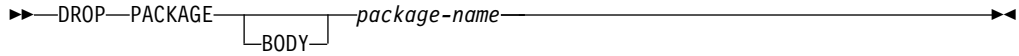
This anonymous block produces the following sample output:

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

Dropping packages (PL/SQL)

You can drop a package if it is no longer needed. Alternatively, if you want to reuse the package, you have the option to drop only the package body.

Syntax



Description

BODY
Specifies that only the package body is to be dropped. If this keyword is omitted, both the package specification and the package body are dropped.

package-name
Specifies the name of a package.

Examples

The following example shows how to drop only the body of a package named EMP_ADMIN:

```
DROP PACKAGE BODY emp_admin
```

The following example shows how to drop both the specification and the body of the package:

```
DROP PACKAGE emp_admin
```

Index

R

S

T

U

V

Printed in USA