Procedures, Functions and Triggers

Slides

- **Anonymous PL/SQL programs**: un-named database objects, submitted to PL/SQL interpreter and run but not available to other users or called by other procedures.

- **Named PL/SQL programs**: Database objects that can be referenced by other programs and can be used by other database users.

- **Stored procedures** – groups of SQL and PL/SQL statements – allow you to move code that enforces business rules from your application to the database.

Performance gains due to two reasons:

1. Processing of complex business rules may be performed with the database – and therefore by the server.

2. Since the procedural code is stored within the database and is fairly static, you may benefit from reuse of the same queries within the database.

Named Program Units

Server-side program units: are stored in the database as database objects and execute on the database server.

Advantages:
1. stored in a central location accessible to all database users,
2. always available whenever a database connection is made.

Disadvantages
1. forces all processing to be done on the database server.
2. If database server is very busy, the response time will be very slow

Client-side program units are stored in the file system of the client workstation and execute on the client workstation.

Program Unit Type	Description	Where Stored	Where Executed
Procedure	Can accept multiple input parameters, and returns multiple output values	Operating system file or database	Client-side or server-side
Function	Can accept multiple input parameters and returns a single output value	Operating system file or database	Client-side or server-side
Library	Contains code for multiple related procedures or functions	Operating system file or database server	Client-side
Package	Contains code for multiple related procedures, functions and variables and can be made available to other database users	Operating system file or database server	Client-side or server-side
Trigger	Contains code that executes when a specific database action occurs, such as inserting, updating or deleting records	Database server	Server-side

Procedures and Functions

PROCEDURE procedure_name(parameter1, mode datatype, …, parameterN mode datatype)
IS
    Statements

FUNCTION procedure_name(parameter1, mode datatype, …, parameterN mode datatype)
IS
    Statements

Mode: how the parameter value can be changed in the program unit

Mode	Description
IN	Parameter is passed to the program unit as a read-only value that cannot be changed with the program unit
OUT	Parameter is a write-only value that can only appear on the left side of an assignment statement in the program unit
IN OUT	Combination of IN and OUT; the parameter is passed to the program unit and its value can be changed within the program unit

Required System Privileges

- In order to create a procedural object you must have the CREATE PROCEDURE system privilege (part of the RESOURCE role)
- If in another user's schema, must have CREATE ANY PROCEDURE system privilege.

Calling Program Units and Passing Parameters

- From within SQL*PLUS, a procedure can be executed by using EXECUTE command, followed by the procedure name.

  EXECUTE procedure_name(parameter1_value, parameter2_value,…);

  EXECUTE New_Worker('Adah Talbot');

- From within another procedure, function, package, or trigger, the procedure can be called without the EXECUTE command.

- **Formal parameters** are the parameters that are declared in the header of the procedure.

- **Actual parameters** are the values placed in the procedure parameter list when the procedure is called.

PROCEDURE cal_gpa(student_id IN NUMBER, current_term_id IN NUMBER, calculated_gpa OUT NUMBER) IS

Formal parameters: student_id, current_term_id, calculated_gpa

Execute cal_gpa(current_s_id, 4, current_gpa);

Actual parameters: current_s_id, 4, current_gpa

Procedures Vs. Functions

- Unlike procedures, functions can return a value to the caller.

Procedures Vs. Packages

- Packages are groups of procedures, functions, variables and SQL statements grouped together into a single unit.

- To EXECUTE a procedure within a package, you must first list the package name, then the procedure name:

  EXECUTE Ledger_Package.New_Worker('Adah Talbot');

- Packages allow multiple procedures to use the same variables and cursors.

- Procedures within packages may be available to the PUBLIC or they may be PRIVATE, in which case they are only accessible via commands from within the package.

- Packages may also include commands that are to be executed each time the package is called, regardless of the procedure or function called within the package.

Creating Stored Procedures in SQL*Plus

Create Procedure Syntax

```
CREATE [or REPLACE] procedure  [user.]procedure [(argument [IN | OUT | IN OUT]
                    Datatype [, argument [IN | OUT | IN OUT ] datatype] …)]  {IS | AS}
                      declarations
BEGIN
        program statements
EXCEPTION
        Exception handlers
END;
```

```
    CREATE PROCEDURE  new_Worker((Person_Name IN VARCHAR2) AS
     BEGIN
        INSERT into Worker (Name, Age, Lodging)
                values(Person_Name, NULL,NULL);
    END;
    /
```

```
CREATE OR REPLACE PROCEDURE update_inventory(current_inv_id IN
       INVENTORY.INV_ID%TYPE, update_quantity IN NUMBER, updated_qoh OUT
       NUMBER) AS
BEGIN
  -- update item QOH
  DBMS_OUTPUT.PUT_LINE('current_inv_id in procedure: ' || current_inv_id);
  UPDATE inventory
  SET qoh = qoh + update_quantity
  WHERE inv_id = current_inv_id;
  COMMIT;
  -- retrieve updated QOH into output parameter variable
END;
/
```

```
SELECT qoh
FROM inventory
WHERE inv_id = 11668;

QOH
--------
   16

EXECUTE update_inventory(11668, -3);


SELECT qoh FROM inventory
WHERE inv_id = 11668;

QOH
--------
   13
```

Debugging Named Program Units in SQL*PLUS

```
CREATE OR REPLACE PROCEDURE update_inventory(current_inv_id IN
       INVENTORY.INV_ID%TYPE; update_quantity IN NUMBER) AS
BEGIN
  -- update item QOH
  UPDATE inventory
  SET qoh = qoh + update_quantity
  WHRE inv_id = current_inv_id;
  COMMIT;
  -- retrieve updated QOH into output parameter variable
END;
/

Warning: Procedure created with compilation errors
```

Figure 5-4. Named program unit with compile error

USER_ERRORS data dictionary view can be queried to get the compilations errors.

A summary of the listing of compile errors generated by the last program unit that was compiled, can be displayed using **SHOW ERRORS** command.

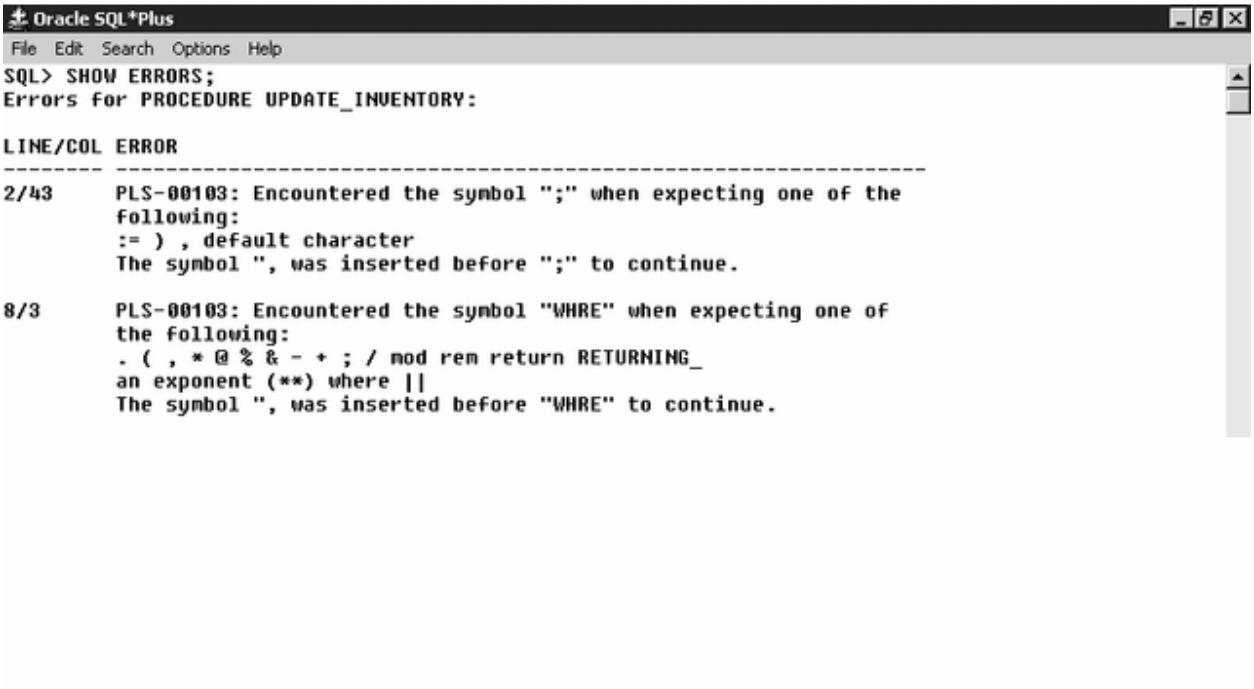


Figure 5-5. Using the SHOW ERRORS command to view compile error details

Place DBMS_OUTPUT.PUT_LINE statements in the procedure code to display variable values during execution.

```
CREATE OR REPLACE PROCEDURE update_inventory(current_inv_id IN
      INVENTORY.INV_ID%TYPE, update_quantity IN NUMBER) AS
BEGIN
  -- update item QOH
  DBMS_OUTPUT.PUT_LINE('current_inv_id in procedure: ' || current_inv_id);
  DBMS_OUTPUT.PUT_LINE('current update_quantity value = ‘ || update_quantity);

  UPDATE inventory
  SET qoh = qoh + update_quantity
  WHERE inv_id = current_inv_id;
  COMMIT;
  -- retrieve updated QOH into output parameter variable
END;
```

Creating Functions in SQL*PLUS

CREATE OR REPLACE FUNTION function_name(parameter1 mode datatype, parameter2 mode datatype, …)
RETURN function_return_value_datatype IS variable declarations

Calling a funtion:

Variable_name := function_name(parameter1, parameter2, …);

Syntax for the body of a function

```
BEGIN
        Program statements
        RETURN return_value;
EXCEPTION
        Exception handlers
        RETURN exception_notice;
END;
```

The RETRUN EXCEPTION_NOTICE command instructs the function to display the exception notice in the program that calls the function.

```
CREATE OR REPLACE FUNCTION age
  (input_dob IN DATE
  )
  RETURN NUMBER IS
  calculated_age NUMBER;
BEGIN
  calculated_age := TRUNC((SYSDATE - input_dob)/365.25);
  RETURN calculated_age;
END;
/
```

Figure 5-7. Creating a user-defined function

Anonymous program block calling function AGE

```
DECLARE
  current_s_dob DATE;
  current_age NUMBER;
BEGIN
  -- retrieve the student's DOB from the database
  SELECT s_dob
  INTO current_s_dob
  FROM student
  WHERE s_last = 'Black'
  AND s_first = 'Daniel';
  -- call the AGE function to determine the student's age
  current_age := AGE(current_s_dob);
  -- display the calculated value
  DBMS_OUTPUT.PUT_LINE('Student''s age is ' || current_age || ' years.');
END;
/
```

Student's age is 21 years.

Figure 5-8. Calling the user-defined function

Function Purity Levels

The functions can be used directly in SQL commands, depending upon the purity level.

Inline functions – can be directly used within SQL commands, such as ROUND, TO_CHAR, etc.

To be used as an inline function, a user-defined function must follow these basic rules:

- The function can only use IN mode parameter, since to be used in a SQL command, it must return only a single value to the calling program.

- The data types of the function input variables and the function return value must be the PL/SQL data type that correspond to the Oracle database data types (VARCHAR2, CHAR, NUMBER, DATE, and so forth). You cannot use the PL/SQL data types that have no corresponding database data types, such as BOOLEAN and BINARY_INTEGER.

- The function must be stored in the database as a database object.

Purity Level	Abbreviation	Description
Writes No Database State	WNDS	Function does not perform any DML commands
Reads No Database State	RNDS	Function does not perform any SELECT commands
Writes No Package State	WNPS	Function does not change values of any package variables
Reads No Package State	RNPS	Function does not read any package variables

The function purity levels place the following restrictions on whether or not a function can be called within a SQL command:

- All inline functions must meet the WNDS purity level.

- Inline functions stored on the database and executed from a SQL query in a program running on the user's workstation (like anonymous PL/SQL program) must meet the RNPS and WNDS purity levels. In contrast, an inline function called from a SQL query in a stored procedure does not have to meet these two purity levels.

- Functions called from the SELECT, VALUES, or SET clauses of a SQL query can write package variables, so they do not need to meet the WNPS purity level. Functions called from any other clause of a SQL query must meet the WNPS purity level.

- A function is only as pure as the purity of a subprogram it calls.

The AGE function can be used an inline function as it meets the criteria: it only contains IN parameters, it returns a database data type, and it satisfies the RNPS and WNDS function purity levels.

```
SELECT s_first, s_last, AGE(s_DOB)
FROM student;

S_FIRST      S_LAST       AGE(S_DOB)
-----------  -----------  ----------
Sarah        Miller             18
Brian        Umato              18
Daniel       Black              21
Amanda       Mobley             19
Ruben        Sanchez            19
Michael      Cannoly            17

6 rows selected
```

Figure 5-9. Using the AGE function as an inline function

Stored Program Unit Object Privileges

GRANT EXECUTE ON unit_name TO username;

GRANT EXECUTE ON MY_Procedure TO Dora;

- If you do not grant EXECUTE privilege to users, then they must have the EXECUTE ANY PROCEDURE system privilege.

GRANT EXECUTE ON age TO PUBLIC;

```
SELECT s_last, lhoward.age(s_dob)
FROM student;

S_LAST       LHOWARD.AGE(S_DOB)
-----------  ------------------
Miller                 18
Umato                  18
Black                  21
Mobley                 19
Sanchez                19
Cannoly                17

6 rows selected.
```

Figure 5-10. Executing a function owned by another user

Creating a Client-side Procedure in Procedure Builder

```
    ALTER TABLE inventory
          ADD  Inv_Value  NUMBER(11,2);


PROCEDURE update_inv_value IS
   CURSOR  inventory_cursor IS
      SELECT * from Inventory;

   Inventory_row          inventory_cursor%ROWTYPE;
   Current_inv_value        NUMBER(11,2);

BEGIN
  For   Inventory_row  IN  inventory_cursor   LOOP
     Current_inv_value := inventory_row.price * inventory_row.qoh;
      UPDATE  Inventory
         SET inv_value := current_inv_value
       WHERE   inv_id = inventory_row.inv_id;
      COMMIT;
   END LOOP;
END;
/
```

Calling a Procedure and Passing Parameters

```
PROCEDURE Update_Inv_Value_Record (Current_Inv_Value  IN NUMBER)
IS
  Current_Price                    NUMBER(5,2);
  Current_QOH                      NUMBER;
  New_Inv_Value                    NUMBER(9,2);

BEGIN
-- retrieve the current values
    SELECT  price, qoh
        INTO     Current_Price, Current_QOH
    FROM   Inventory
        WHERE    inv_id = Current_Inv_Value;

-- calculate the new inv_value
     New_Inv_Value := Current_Price* Current_QOH;
-- update the record
     UPDATE  inventory
          SET    inv_value = New_Inv_Value
               WHERE  inv_id = Current_Inv_Value;
    COMMIT;
END;
```

Calling Procedure

```
PROCEDURE Update_QOH(Current_Inv_ID NUMBER, New_QOH  NUMBER)
IS
BEGIN
  -- update QOH
   UPDATE  inventory
   SET qoh = New_QOH
   WHERE inv_id = Current_Inv_ID;
   COMMIT;
  -- call procedure to update INV_VALUE
   Update_Inv_Value_Record(Current_Inv_ID);
END;
```

Clearwater Traders Example:

Procedure named CREATE_NEW_ORDER, received customer ID, order source ID, method of payment, inventory ID, and order quantity, then inserts the order information into the CUST_ORDER table The procedure calls another procedure named CREATE_NEW_ORDER_LINE and passes to it the values for the order ID, inventory ID, and quantity ordered. The second procedure will then insert a new record into the ORDER_LINE table.

Creating procedure CREATE_NEW_ORDER_LINE

```
CREATE SEQUENCE order_id_sequence
START WITH 1100;


PROCEDURE create_new_order_line( current_inv_id NUMBER, current_quantity NUMBER)
IS
BEGIN
        -- Insert new order line record
        INSERT INTO order_line VALUES(order_id_sequence.currval, current_inv_id,
                                                current_quantity);

        COMMIT;
END;
```

Creating procedure CREATE_NEW_ORDER

```
PROCEDURE CREATE_NEW_ORDER(current_cust_id  NUMBER, current_meth_pmt
VARCHAR2, current_order_source_id  NUMBER, current_inv_id  NUMBER, current_quantity
NUMBER)
IS
BEGIN
        -- Insert the CUST_ORDER record
        INSERT INTO cust_order VALUES(order_id_sequence.nextval, SYSDATE,
                        current_meth_pmt, current_cust_id, current_order_source_id);
        COMMIT;

        -- Call the procedure to insert the ORDER_LINE record
        CREATE_NEW_ORDER_LINE(current_inv_id, current_quantity);
END;
```

To run the procedures:

```
PL/SQL> CREATE_NEW_ORDER(107, 'CC', 6, 11795, 1);
PL/SQL> SELECT *
        +>  FROM cust_order;

ORDER_ID   ORDER_DATE   METH_PMT  CUST_ID  ORDER_SOURCE_ID
---------- ------------ --------- -------- ---------------
      1057 29-MAY-03    CC            107               2
      1058 29-MAY-03    CC            232               6
      1059 31-MAY-03    CHECK         133               2
      1060 31-MAY-03    CC            154               3
      1061 01-JUN-03    CC            179               6
      1062 01-JUN-03    CC            179               3
      1100 13-DEC-00    CC            107               6

7 ROWS SELECTED.

PL/SQL> SELECT *
        +> FROM order_line
        +> WHERE order_id = 1100;

ORDER_ID   INV_ID     ORDER_QUANTITY
---------- ---------- --------------
      1100      11795              1

1 row selected.
```

Functions

The function Balance_check returns status of the 'Bought' and 'sold' transactions for a Person in the LEDGER table.

```
CREATE  function Balance_check(Person_Name  IN VARCHAR2)
  RETURN  NUMBER
IS
     balance    NUMBER(10,2);

BEGIN
    SELECT    SUM(DECODE(Action, 'Bought', Amount, 0))
                          -  SUM(DECODE(Action, 'sold', Amount, 0))
     INTO  balance
   FROM   ledger
   WHERE      person = Person_Name;
  RETURN(balance);
END;
/



FUNCTION Student_Age(Current_S_ID  NUMBER)
   RETURN  NUMBER
IS
    Current_Date       DATE;
     Student_DOB     DATE;
    Curent_Age         NUMBER;

BEGIN
     Current_Date := SYSDATE;
 -- retrieve SDOB for SID
     SELECT  s_dob
       INTO      Student_DOB
     FROM    student
       WHERE   s_id = Current_S_ID;
     Current_Age := TRUNC((Current_Date – Student_DOB)/365);
     RETURN  CurrentAge;
END;
/
```

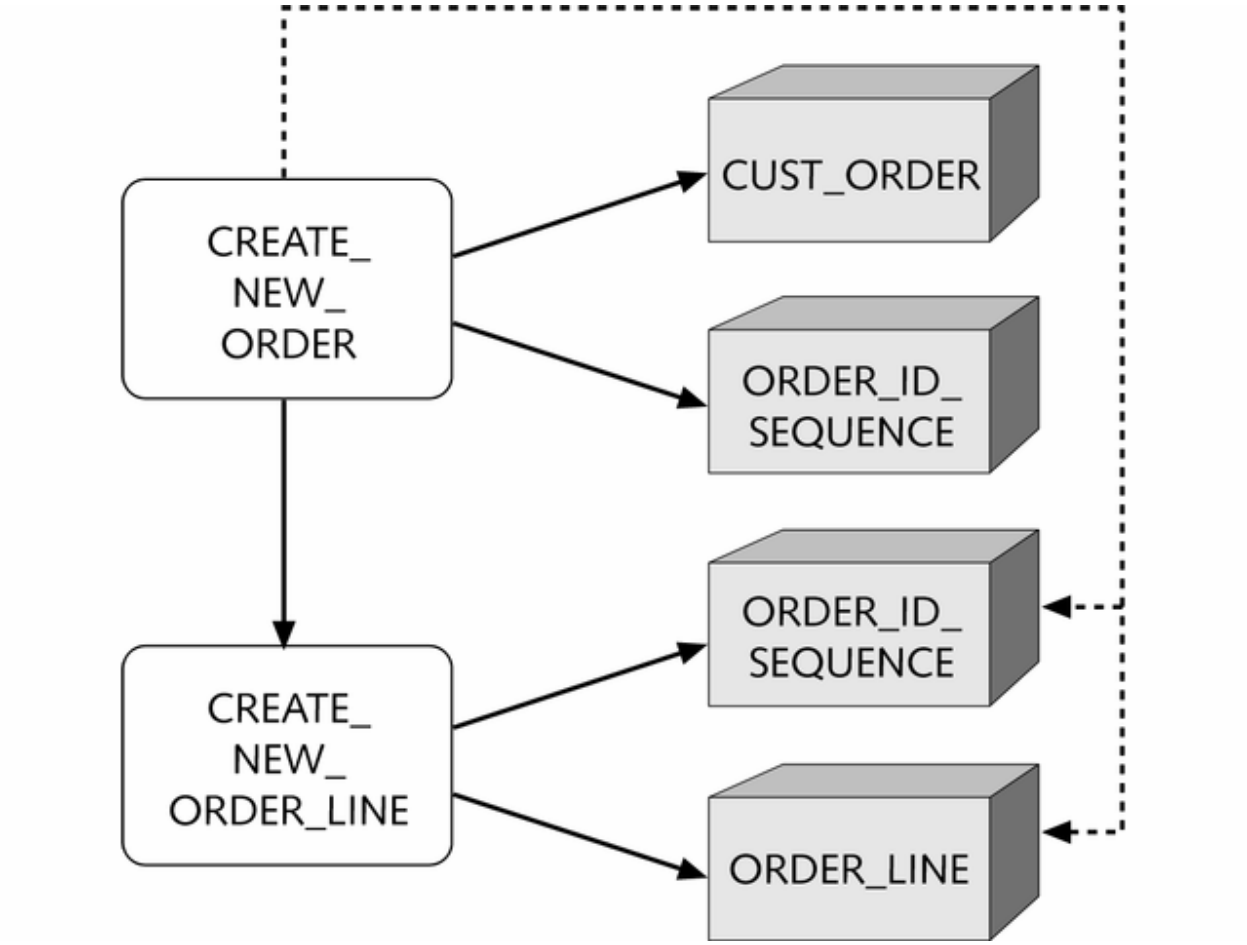
Calling a Function

```
PROCEDURE Update_Student_Data(Curr_S_ID  NUMBER)
IS
     Current_Student_Age      NUMBER(2);

BEGIN
    Current_Student_Age := Student_Age(Curr_S_ID);
END;
```

Program Unit Dependencies

When a procedure or function is compiled, the database objects that it references (such as tables, views, or sequences) are verified to make sure that they exist and that the user has sufficient object privileges to use the objects as specified in the program code.



A program unit is **directly dependent** on an item if it references it directly within its procedure code.

A program unit is **indirectly dependent** on an item if the item is referenced by an item that the procedure references.

Data Dictionary views

DBA_DEPENDENCIES – dependency information for program units for all users

USER_DEPENDENCIES – dependency information for only the current user

ALL_DEPENDENCIES – dependency information for the current user, as well as dependency information for program units that the current user has privileges to execute.

Column Name	Description
OWNER	Username of the program unit owner
NAME	Program unit name in all uppercase characters
TYPE	Program unit type (procedure, function, library, package, or package body)
REFERENCED_OWNER	Username of the owner of the referenced object
REFERENCED_NAME	Name of the referenced object
REFERENCED_TYPE	Object type (table, sequence, view, procedure, package)
REFERENCED_LINK_NAME	The name of the database link (connect string) used to access the referenced object. This column has a value only when the referenced object is on a different Oracle database

To create a formatted report to query the USER_DEPENDENCIES view:

```
SET LINESIZE 100
SET PAGESIZE 25

COLUMN name HEADING 'Name' FORMAT A25
COLUMN type HEADING  'Type' FORMAT A10
COLUMN referenced_name  HEADING 'Referenced' FORMAT A25
COLUMN referenced_type HEADING 'ref. Type' FORMAT A25

SELECT name, type, referenced_name, referenced_type
FROM USER_DEPENDENCIES
WHERE name = 'CREATE_NEW_ORDER';
```

```
Name                            Type               Reference                          Ref. Type
------------------------------- ------------------ ---------------------------------- ----------------------
CREATE_NEW_ORDER    PROCEDURE   STANDARD                           PACKAGE
CREATE_NEW_ORDER    PROCEDURE   DBMS_STANDARD                      PACKAGE
CREATE_NEW_ORDER    PROCEDURE   CUST_ORDER                         TABLE
CREATE_NEW_ORDER    PROCEDURE   ORDER_ID_SEQUENCE                  NON-EXISTENT
CREATE_NEW_ORDER    PROCEDURE   CREATE_NEW_ORDER_LINE              PROCEDURE
CREATE_NEW_ORDER    PROCEDURE   CREATE_NEW_ORDER_LINE              NON-EXISTENT
CREATE_NEW_ORDER    PROCEDURE   ORDER_ID_SEQUENCE                  SEQUENCE
```

The Package Specification

PACKAGE <package name>
IS
        <variable declarations>;
        <cursor declarations>;
        <procedure and function declarations>;
END <package name>;

CURSOR <cursor name>
RETURN <cursor return row variable>
IS <cursor SELECT statement>;

PROCEDURE <procedure name>(param1 param1datatype, param2 param2datatype, …);

FUNCTION <function name>(param1 param1datatype, param2 param2datatype, …)
RETURN <return data type>;

PACKAGE Inventory_Package IS
 -- variable declaration
    Global_Inv_ID   NUMBER(6);
 -- Program unit declarations
 Procedure Update_Inv_Value;
 Procedure Update_Inv_Value_Record;
 Procedure Update_QOH(Current_Inv_ID NUMBER, New_QOH NUMBER);
END;

The Package Body

PACKAGE BODY <package name>
IS
<variable declarations>
<cursor specifications>
<module bodies>
END <package name>;

```
PACKAGE BODY Inventory Package IS    -- start of package body
 -- cursor declarations
     CURSOR  Update_Inv_Cursor IS
       SELECT * from Inventory;
 -- variable declarations
     Inventory_Row          Update_Inv_Cursor%ROWTYPE;
     Inventory_Value        NUMBER(11,2);
     Current_Price          NUMBER(5,2);
     Current_QOH            NUMBER;
     New_Inv_Value          NUMBER(9,2);

 -- modules

PROCEDURE Update_Inv_Value IS
BEGIN
  For   Inventory_Row  IN  Update_Inv_Cursor   LOOP
     Inventory_Value := Inventory_Row.price * Inventory_Row.qoh;
      UPDATE  Inventory
         SET inv_value = Inventory_Value
       WHERE   inv_id = Inventory_Row.inv_id;
      COMMIT;
   END LOOP;
END;


PROCEDURE Update_Inv_Value_Record
IS
BEGIN
-- retrieve the current values
    SELECT  price, qoh
        INTO     Current_Price, Current_QOH
    FROM   Inventory
        WHERE    inv_id = Inventory_Package.Global_Inv_ID;

-- calculate the new inv_value
     New_Inv_Value := Current_Price* Current_QOH;
-- update the record
     UPDATE  inventory
          SET    inv_value = New_Inv_Value
     WHERE  inv_id = Inventory_Package.Global_Inv_ID;
    COMMIT;
END;
```

```
PROCEDURE Update_QOH(Current_Inv_ID NUMBER, New_QOH  NUMBER)
IS
BEGIN
        -- assign global variable to value of input INV_ID
          Inventory_Package.Global_Inv_ID := Current_Inv_ID;
 -- update QOH
   UPDATE  inventory
   SET qoh = New_QOH
   WHERE inv_id = Current_Inv_ID;
   COMMIT;
 -- call procedure to update INV_VALUE
   Update_Inv_Value_Record;
END;
```

END; -- end of package body

To run the Update_QOH procedure in the package and verify that it updated the record:

```
Inventory_Package.update_QOH(11669,10);

select invi_d, inv_value
    from inventory
where inv_id = 11669;
```

To create package specification using SQL*PLUS:

```
SQL> CREATE OR REPLACE PACKAGE order_package IS
        -- Declare public variables
        global_inv_id          NUMBER(6);
        global_quantity        NUMBER(6);

        -- declare program units
        PROCEDURE create_new_order(current_cust_id  NUMBER, current_meth_pmt
                                   VARCHAR2, current_order_source_id  NUMBER);
        PROCEDURE create_new_order_line;
END;
/
```

To create the package body in SQL*PLUS:

```
SQL> CREATE OR REPLACE PACKAGE BODY order_package IS
        PROCEDURE create_new_order(current_cust_id  NUMBER, current_meth_pmt
           VARCHAR2, current_order_source_id  NUMBER)
        IS
        BEGIN
                -- Insert the CUST_ORDER record
                INSERT INTO cust_order VALUES(order_id_sequence.nextval, SYSDATE,
                        current_meth_pmt, current_cust_id, current_order_source_id);
                COMMIT;

        END;


        PROCEDURE create_new_order_line
        IS
        BEGIN
                -- Insert new order line record
                INSERT INTO order_line VALUES(order_id_sequence.currval, global_inv_id,
                                                global_quantity);

                COMMIT;
        END;
END;
/
```

Referencing Package Items

To grant other users the privilege to execute a package:

GRANT EXECUTE ON package_name TO username;

To create the program to reference the package items:

```
        BEGIN
                -- Initialize package variables
                ORDER_PACKAGE.GLOBAL_INV_ID := 11668;
                ORDER_PACKAGE.GLOBAL-QUANTITY := 2;

                -- Call the procedures
                ORDER_PACKAGE.CREATE_NEW_ORDER(107, 'CC', 2);
                ORDER_PACKAGE.CREATE_NEW_ORDER_LINE;
        END;
        /
```

```
SELECT *
FROM cust_order;
```

ORDER_ID	ORDER_DAT	METH_PMT	CUST_ID	ORDER_SOURCE_ID
1057	29-MAY-03	CC	107	2
1058	29-MAY-03	CC	232	6
1059	31-MAY-03	CHECK	133	2
1060	31-MAY-03	CC	154	3
1061	01-JUN-03	CC	179	6
1062	01-JUN-03	CC	179	3
1100	13-DEC-00	CC	107	6
1101	13-DEC-03	CC	107	2

8 rows selected

```
SELECT *
FROM order_line;
```

ORDER_ID	INV_ID	ORDER_QUANTITY
1057	11668	1
1057	11800	2
1058	11824	1
1059	11846	1
1059	11848	1
1060	11798	2
1061	11779	1
1061	11780	1
1062	11799	1
1062	11669	3
1100	11795	1
1101	11668	2

Overloading Program Units in Packages

In a package, program units can be overloaded, which means that multiple program units with the same name but different input parameters exist.

To create the package specification:

```
PACKAGE enrollment_package IS
    PROCEDURE add_enrollment(current_s_id  NUMBER, current_c_sec_id
        NUMBER);
    PROCEDURE add_enrollment(current_s_first  VARCHAR2, current_s_last
        VARCHAR2, current_c_sec_id  NUMBER);
END;
```

To create the package body:

```
PACKAGE BODY enrollment_package IS
    -- Procedure to insert enrollment record based on student ID
    PROCEDURE add_enrollment(current_s_id  NUMBER, current_c_sec_id
       NUMBER)
    IS
    BEGIN
            INSERT INTO enrollment VALUES(current_s_id, current_c-sec_id,
                                                    NULL);
            COMMIT;
    END;

    -- Procedure to insert enrollment record based on student first and last names
    PROCEDURE add_enrollment(current_s_first  VARCHAR2, current_s_last
             VARCHAR2, current_c_sec_id  NUMBER)
    IS
    Retrieved_s_id   NUMBER;
    BEGIN
            -- Retrieve student ID
            SELECT s_id
            INTO retrieved_s_id
            FROM student
            WHERE s_first = current_s_first
            AND s_last = current_s_last;

            INSERT INTO enrollment VALUES(retrieved-s_id, current_c_sec_id,
                                                    NULL);
            COMMIT;
    END;
END;
```

To test the overloaded procedures:

```
PL/SQL> ENROLLMENT_PACKAGE.ADD_ENROLLMENT(100, 1012);
PL/SQL> ENROLLMENT_PACKAGE.ADD_ENROLLMENT('Daniel', 'Black',
        +> 1001);
PL/SQL> SELECT *
        +> FROM enrollment
        +> WHERE grade IS NULL;

S_ID       C_SEC_ID    GRADE
------     ----------  ----------
  102          1011
  102          1012
  103          1010
  103          1011
  104          1012
  104          1010
  105          1010
  105          1011
  100          1012
  102          1001

10 rows selected
```

- You can overload two procedures that use the same number of parameters if the parameters differ in data type. The following overloaded procedure declarations are legal:

  PROCEDURE legal_overload(current_value1 NUMBER);
  PROCEDURE legal_overload(current_value1 VARCHAR2);

- You cannot overload two procedures if they have the same number of parameters and the parameters have the same data type and differ only in name. The following overloaded procedure declarations are invalid:

  PROCEDURE illegal_overload(current_value1 NUMBER);
  PROCEDURE illegal_overload(current_value2 NUMBER);

- You cannot overload two procedures that use the same number of parameters if the parameter data types come from the same data type family, such as numbers, or characters. The following overloaded procedure declarations are invalid:

  PROCEDURE illegal_overload(current_value1 CHAR);
  PROCEDURE illegal_overload(current_value1 VARCAHR2);

- You cannot overload two functions if their only differences are their return types. The following overloaded function declarations are invalid:

  FUNCTION illegal_overload RETURN NUMBER;
  FUNCTION illegal_overload RETURN DATE;

Database Triggers

Database triggers are program units that execute in response to the database events of inserting, updating, or deleting a record.

Triggers may be used to supplement declarative referential integrity, to enforce complex business rules, to audit changes to data, or to signal to other programs that changes were made to a table.

The code within a trigger body is made up of PL/SQL. The execution of triggers is transparent to the users.

Difference between triggers and other program units

1. Triggers cannot accept input parameters

2. Program units have to be explicitly executed by typing the program unit name at the command prompt or in a command in a calling procedure.

3. Trigger executes only when its triggering event occurs. When a trigger executes, it is said to have **fired**.

Required System Privileges

To create a trigger on a table, you must be able to alter that table. So you must either **own the table**, have **ALTER** privilege for the table, or have **ALTER ANY TABLE** system privilege.

Also you must have **CREATE TRIGGER** system privilege; to create triggers in another user's account (also called a schema), you must have **CREATE ANY TRIGGER** system privilege.

The **CREATE TRIGGER** system privilege is part of the **RESOURCE** role provided with ORACLE.

To **alter a trigger**, you must either **own the trigger** or have the **ALTER ANY TRIGGER** system privilege.

You may also **alter trigger** by **altering the tables** they are based on.

Required Table Privileges

Triggers may reference tables other than the one that initiated the triggering event.

For example, if you use triggers to audit changes to data in the LEDGER table, then you may insert a record into a different table (say, LEDGER_AUDIT) every time a record is changed in LEDGER.

Types of Triggers

Type	Values	Description
Statement	INSERT, DELETE, UPDATE	Defines statements that causes trigger to fire
Timing	BEFORE, AFTER	Defines whether trigger fires before or after statement is executed
Level	ROW, STATEMENT	Defines whether trigger fires once for each triggering statement, or once for each row affected by the triggering statement

Row-Level Triggers

Row-level triggers execute once for each row in a transaction.

They are created using **the for each row** clause in the **create trigger** command.

```
Create [or replace] trigger [user.]trigger
   {before | after | instead of}
    {delete | insert | update [of column [, column] …] }
   [ or {delete | insert | update [ of column [, column] …] } ] …
on [user.] {Table | View}
[ [ referencing {old [as] old
                    | new [as] new} … ]
for each  { row |statement} [when (condition) ]] PL/SQL block;
```

Statement-Level Triggers

Statement-level triggers execute once for each transaction, either before or after the SQL triggering statement executes.

For example, you could use a statement-level trigger to record an audit trail showing each time the ENROLLMENT table is changed, regardless of how many rows are affected.

Before and After Triggers

Since the events that execute triggers are database transactions, triggers can be executed immediately before or after **inserts**, **updates**, and **deletes**.

INSTEAD OF Triggers

You can use **INSTEAD OF** triggers to tell ORACLE what to do instead of performing the actions that executed the trigger.

For example, you could use an **INSTEAD OF** trigger to redirect table **inserts** into a different table or to **update** multiple tables that are part of a view.

Valid Trigger Types
14 possible configurations:

BEFORE INSERT row
BEFORE INSERT statement
AFTER INSERT row
AFTER INSERT statement
BEFORE UPDATE row
BEFORE UPDATE statement
AFTER UPDATE row
AFTER UPDATE statement
BEFORE DELETE row
BEFORE DELETE statement
AFTER DELETE row
AFTER DELETE statement
INSTEAD OF row
INSTEAD OF statement

Trigger Syntax

```
Create [or replace] trigger [user.] trigger
   {before | after | instead of }
    {delete
     | insert
     | update [of column [, column [, column] … ] }
   [or {delete
         | insert
         | update [of column [, column] …} } ] …
  on [user.] {TABLE | VIEW}
 [ [ referencing  {old [as] old
                     | new [as] new} … ]
for each {row | statement} [when (condition) ]] PL/SQL BLOCK;
```

```
Create trigger ledger_bef_upd_row
before update on LEDGER
for each row
when (new.Amount/old.Amount > 1.1)
BEGIN
   insert into LEDGER_AUDIT
        values( :old.Actiondate, :old.Action, :old.Item, :old.quantity, :old.QuantutyType,
                 :old.Rate, :old.Amount, :old.Person);
END;
```

Combining Trigger Types

```
Create trigger ledger_bef_upd_ins_row
before insert or update of Amount on LEDGER
for each row
BEGIN
         if INSERTING then
            insert into LEDGER_AUDIT
                values(:new.Action_Date, :new.Action, :new.Item, :new.Quantity,
                          :new.QuantityType, :new.Rate, :new.Amount, :new.Person);
         else – if not inserting, then we are updating Amount
            insert into LEDGER_AUDIT
               values(:old.Action_date, :old.Action, :old.Item, :old.Amount, :old.Person);
         end if;
END;
```

Setting Inserted Values

You may use triggers to set column values during **insert**s and **update**s.

For example, you may have partially denormalized your LEDGER table to include derived data, such as UPPER(Person).

```
create trigger ledger_bef_upd_ins_row
before insert or update of Person on LEDGER
for each row
BEGIN
        :new.UperPerson := UPPER(:new.Person);
END;
```

Customizing Error Conditions

Within a single trigger, you may establish different error conditions. For each of the error conditions you define, you may select an error message that appears when the error occurs.

The error numbers and messages that are displayed to the user are set via the RAISE_APPLICATION_ERROR procedure.

The following example shows a statement-level BEFORE DELETE trigger on the LEDGER table.

When a user attempts to delete a record from the LEDGER table, this trigger is executed and checks two system conditions:

1. The day of the week is neither Saturday nor Sunday, and that the
2. ORACLE username of the account performing the delete begins with the letters ‘FIN’.

```
create trigger ledger_bef_del
before delete on LEDGER
declare
   weekend_error      EXCEPTION;
   not_finance_user   EXCEPTION;
BEGIN
        if TO_CHAR(SYSDATE, 'DY') = 'SAT' or
           TO_CHAR(SYSDATE, 'DY') = 'SUN' THEN
            RAISE  weekend_error;
        end if;
        IF SUBSTR(USER, 1,3) <> 'FIN' THEN
                RAISE  not_finance_user;
EXCEPTION
        WHEN weekend_error  THEN
                RAISE_APPLICATION_ERROR(-20001, 'Deletions not allowed on
                                                              weekends');
        WHEN not_finance_error THEN
                RAISE_APPLICATION_ERROR(-20001, 'Deletions only aloowed by
                                                                Finance Users');
END;
```

There are no **when** clauses in this trigger, so the trigger body is executed for all **deletes**.

Enabling and Disabling Triggers

To enable a trigger, use the **alter trigger** command with the **enable** keyword.

In order to use this command, you must either own the table or have **ALTER ANY TRIGGER** system privilege.

```
alter trigger ledger_bef_upd_row enable;
```

A second method of enabling triggers uses the **alter table** command, with the **enable all triggers** clause.

```
alter table LEDGER enable all triggers;
```

To use the **alter table** command, you must either own the table or have **ALTER ANY TABLE** system privilege.

You can disable triggers using the same basic commands with modifications to their clauses.

```
alter trigger ledger_bef_upd_row disable;
```

For the **alter table** command, use the **disable all triggers** clause as shown:

```
alter table LEDGER disable all triggers;
```

Dropping Triggers

In order to drop a trigger, you must either own the trigger or have the DROP ANY TRIGGER system privilege.

```
drop trigger ledger_bef-upd_row;
```

To create the ORDER_LINE table trigger:

1. Click + beside ORDER_LINE under your list of database tables, then click **Triggers**. Click the **Create** button to create a new trigger for the ORDER_LINE table. The Database Triggers dialog box opens.

2. Click **New** on the bottom-left of the dialog box. Delete the default trigger name, and type **Order_line_Trigger** for the trigger name.

3. Make sure that the **Before** option button is selected, and check **UPDATE**, **INSERT**, and **DELETE** to specify the trigger statements.

4. Click **ORDER_QUANTITY** in the Of Columns: list box to specify the update field that will fire the trigger. Select the **Row** option button to specify that this is a row-level trigger. Type **OLD** in the Referencing OLD As: text box, and type New in the **NEW** in the New As: text box.

To create the trigger body:

1. Type the trigger body code shown in Fig. 4.94. Click **Save** to create the trigger, then click Close to close the Database Trigger dialog window. Now you will test the trigger. First, you will test to confirm that the INSERT part of the trigger works correctly.

```
BEGIN
    IF INSERTING THEN
       update inventory
         set qoh = qoh - :NEW.order_quantity;
    END IF;
    IF UPDATING THEN
        UPDATE inventory set qoh = qoh + :OLD.order_quantity - :NEW.order_quantity;
    END IF;
    IF DELETING THEN
         UPDATE inventory set qoh = qoh + :OLD.order_quantity;
    END IF;
END;
```

2. Open the PL/SQL Interpreter window, and type the SELECT command shown in Fig. 4.94 at the PL/SQL> prompt to determine the current QOH of inventory item 11848. The current QOH should be 12.

```
SELECT qoh
   from inventory
where inv_id = 11848;
```

3. Type the INSERT command shown in Fig. 4.95 to insert a new record into ORDER_LINE.

```
INSERT INTO order_line
     VALUES(1062, 11848, 1);
```

4. Type the SELECT command again, as shown, to confirm that the QOH of INV_ID 11848 was decreased by one from 12 to 11 as a result of adding the record to ORDER_LINE. Now you will test to make sure the UPDATE and DELETE trigger statements work correctly.

```
select qoh
   from inventory
where inv_id = 11848;
```

5. Type the UPDATE command shown in Fig. 4.96 to update the quantity ordered from one item to two items. Type the SELECT command to confirm that the QOH was decreased by one to reflect the change.

```
update order_line
  set order_quantity = 2
where order_id = 1062
    and inv_id = 11848;

select qoh
  from inventory
where inv_id = 11848;
```

6. Type the DELETE command shown in Fig. 4.96 to delete the order from the database.

```
delete from order_line
    where order_id = 1062
     and     inv_id = 11848;
```

7. Type the second SELECT command to confirm that the QOH was increased by two (from 10 to 12) to reflect the change caused by deleting the order.

```
select qoh
  from inventory
where inv_id = 11848;
```

Viewing Information About Triggers

```
SELECT trigger_name, trigger_type, triggering_event
FROM user_triggers;

TRIGGER_NAME            TRIGGER_TYPE     TRIGGERING_EVENT
------------------------ ------------------------- ------------------------------
enrollment_audit        after statement  insert or update or delete
enrollment_row_audit    before each row  update
```

Built-In Packages

The packages fall into five categories:

1. transaction processing
2. application development
3. database administration
4. application administration
5. internal support

Transaction Processing Packages

Procedures related to **locking** and **releasing** database objects to support transaction processing, supporting the **COMMIT** and **ROLLBACK** commands, and providing a way to specify query properties in PL/SQL programs at runtime.

Package Name	Description
DBMS_ALERT	Dynamically sends messages to other database sessions
DBMS_LOCK	Creates user-defined locks on tables and records
DBMS_SQL	Implements Dynamic SQL
DBMS_TRANSACTION	Provides procedures for transaction management, such as creating read-only transactions, generating a rollback, and creating a savepoint

Application Development Packages

Aid developers in creating and debugging PL/SQL applications.

Package Name	Description
DBMS_DESCRIBE	Returns information about the parameters of any stored program unit
DBMS_JOB	Schedules PL/SQL programs to run at specific times
DBMS_OUTPUT	Provides text output in PL/SQL programs
DBMS_PIPE	Sends messages to other database sessions asynchronously
DBMS_SESSION	Dynamically changes properties of a database session
UTL_FILE	Enables PL/SQL output to be written to a binary file

Using the DBMS_JOB Package

PL/SQL procedure that creates and prints a report containing information about incoming shipments at Clearwater Traders on the first day of every month.

The package creates a **job queue**,

- which is a list of program units to be run and the details for when and how often they are to be run.

The packages **SUBMIT** procedure is used to submit a job to the job queue, using the following syntax:

DBMS_JOB.SUBMIT(job_number, call_to_stored_program_unit, next_run_date, interval_to_run_job_again, no_parse_value);

This command has the following parameters:

- *job_number* is a unique number that is automatically assigned to the job by the procedure.
- *Call_to_stored_program_unit* is a string representing the code used to call the program unit, including parameter values.
- *Next_run_date* is a date that specifies the next date and time when the job is scheduled to run.
- *Interval_to_run_job_again* is a text string that specifies a time interval when the job will be run again. For example, if the job is to run for the first time on the current system date and the next time one day aft the current system date, the interval parameter would be specified as 'SYSDATE+1'.
- *No_parse_value* is a Boolean parameter that specifies whether or not the program unit is to be parsed and validated the next time it executes.

The DBMS_JOB package's **RUN** procedure can be used to run a previously submitted job immediately.

This procedure has the syntax:

DBMS_JOB.RUN(job_number);

The *job_number* parameter corresponds to the job number that is assigned to the job when it is submitted to the job queue using the SUBMIT procedure.

The DBMS_JOB package's **REMOVE** procedure can be used to remove a job from the job queue. The **REMOVE** procedure has the syntax

DBMS_JOB.REMOVE(job_number);

Using the DBMS_PIPE Package

A **pipe** is a program that directs information from one destination (such as screen display) to a different destination (such as a file or database).

The DBMS_PIPE package implements **database pipes**, which are pipes that are implemented entirely in Oracle database and are independent of the operating system, of the database server, or client workstation.

A specific pipe has a single **writer**, which is the session that creates the message, and multiple **readers**, which are sessions that receive the message.

Pipes are **asynchronous**, which means that they operate independently of transactions.

To send a message using a pipe, use the PACK_MESSAGE and SEND_MESSAGE procedures.

PACK_MESSAGE procedure places the outgoing message in a buffer using the syntax:

DBMS_PIPE.PACK_MESSAGE(message);

The message can be of any data type, and multiple items of different data types can be sent in a single message.

The SEND_MESSAGE function sends the contents of the pipe using the following syntax:

Return_value := DBMS_PIPE.SEND_MESSAGE('pipe_name', timeout_interval, maximum_pipe_size);

- *return_value*, which is an integer variable that is assigned the following values depending on the pipe status after the function is called:

Value	Meaning
0	Pipe message successfully sent
1	The call timed out, possibly because the pipe was too full to be sent or a lock could not be obtained
3	The call was interrupted due to an internal error

- *pipe_name*, which identifies the pipe and can be any name that conforms to the Oracle naming standard

- *timeout_interval*, which is an optional parameter that specifies the time interval, in seconds, that the procedure should try to send the pipe before quitting and returning an error message. Because the default value is 1000 days, it is a good idea to specify a smaller value when developing new pipes. Otherwise, the system will hang, and SQL*PLUS will have to be shut down and restart it if there is a mistake in the code.

- *Maximum_pipe_size*, specifies the amount of buffer space, in bytes, that is allocated to the pipe, with a default value of 8,192 bytes. Most messages need a larger value, because some of the pipe buffer space is occupied with values specifying the pipe properties. If the pipe buffer size is too small, the pipe will not be successfully received and the receiver's SQL*PLUS session will lock up and have to be restarted. Once a pipe is sent, it remains in the pipe buffer until the database is shut down and then restarted.

```
DECLARE
        -- Pipe message contents
        out_message_text      VARCHAR(30);
        sending_username      VARCHAR2(30);
        sending_date          DATE;
        -- return value for sending pipe
        pipe_status           BINARY_INTEGER;
BEGIN
        -- Retrieve current date and username
        SELECT user, sysdate
        INTO sending_username, sending_date
        FROM dual;

        -- Initialize text message
        output_message_text := 'Test Message';
        -- pack the message items
        DBMS_PIPE.PACK_MESSAGE(output_message_text);
        DBMS_PIPE.PACK_MESSAGE(sending_username);
        DBMS_PIPE.PACK_MESSAGE(sending_date);
        -- Send the pipe
        pipe_status := DBMS_PIPE.SEND_MESSAGE('test_pipe', 0, 30000);
        DBMS_OUTPUT.PUT_LINE('Outgoing pipe status: ' || pipe_status);
END;
/
Outgoing pipe status:   0
```

System Privileges

To use the DBMS_PIPE package, you must have the EXECUTE_ANY_PROCEDURE system privilege.

Receiving a pipe

Return_value := DBMS_PIPE.RECEIVE_MESSAGE('pipe_name', timeout_interval);

Return_value is an integer variable assigned the following values based on the outcome of receiving message:

Value	Meaning
0	Pipe message successfully received
1	No message received and the function timed out
2	The message in the pipe was too large for the buffer
3	an internal error occurred

pipe_name – name of the pipe specified in the SEND_MESSAGE function

timeout_interval – optional parameter that specifies the time interval, in seconds, that the procedure should continue to try to receive the pipe.

Default value: 1000 days

Use shorter value when developing new programs to receive pipes, or else the program will appear to lock up in SQL*PLUS if there is an error in the code

Unpacking message

DBMS_PIPE.UNPACK_MESSAGE(output_variable_name);

output_variable_name – name of variable that is used to reference each item in the pipe.

The receiver must unpack the pipe message for each item that was sent. The message items must be unpacked in the same order they were packed, using output variables of the correct data type.

To receive the pipe:

```
SET SERVER OUTPUT ON SIZE 4000

DECLARE
        -- Return value for receiving pipe
        pipe_status     BINARY_INTEGER;
        -- variables to unpack message items
        incoming_message_text       VARCHAR2(30);
        incoming_username           VARCHAR2(30);
        incoming_date               DATE;
BEGIN
        Pipe_status := DBMS_PIPE.RECEIVE_MESSAGE('test_pipe', 0);
        DBMS_OUTPUT.PUT_LINE('Incoming pipe status: ' || pipe_status);
        DBMS_PIPE.UNPACK_MESSAGE(incoming_message_text);
        DBMS_OUTPUT.PUT_LINE('Message: ' || incoming_message_text);
        DBMS_PIPE.UNPACK_MESSAGE(incoming_username);
        DBMS_OUTPUT.PUT_LINE('Sender: ' || incoming_username);
        DBMS_PIPE.UNPACK_MESSAGE(incoming_date);
        DBMS_OUTPUT.PUT_LINE('Date sent: ' || incoming_date);
END;
/
Incoming pipe status:  0
Message:  Test Message
Sender:  LHOWARD
Date sent:  19-DEC-00
```

Database and Application Administration Packages

Supports database administration tasks, such as

- Managing memory on the database server
- Managing how disk space is allocated and used
- Recompiling and managing stored program units and packages

Package Name	Description
DBMS_APPLICATION_INFO	registers information about programs being run by individual user sessions
DBMS_DDL	Provides procedures for compiling program units and analyzing database objects
DBMS_SHARED_POOL	Used to manage the shared pool, which is a server memory area that contains values that can be accessed by all users
DBMS_SPACE	Provides information for managing how data values are physically stored on the database server disks
DBMS_UTILITY	Provides procedures for compiling all program units and analyzing all objects in a specific database schema

The DBMS-DDL Package

ALTER_COMPILE procedure – used to recompile a program unit that is invalidated due to changes in tables or procedures on which it has dependencies.

The procedure makes it easy to quickly recompile several program units

Calling the procedure:

DBMS_DDL.ALTER_COMPILE(program_unit_type, owner_name, program_unit_name);

- *Program_unit_type*, which specifies the type of program unit, such as 'PROCEDURE', 'FUNCTION', 'PACKAGE', 'PACKAGE BODY'. The value must be enclosed in single quotation marks and must be in all capital letters.
- *Owner_name*, which specifies the username of the user who owns the program unit. The username is enclosed in single quotation marks and must be in all capital letters.
- *Program_unit_name*, which specifies the name of the program unit, enclosed in single quotation marks and in all capital letters.

To recompile the program units:

```
BEGIN
        DBMS_DDL.ALTER_COMPILE('PROCEDURE', 'LHOWARD',
                                                'CREATE_NEW_ORDER');
        DBMS_DDL.ALTER_COMPILE('FUNCTION', 'LHOWARD',
                                                'AGE');
        DBMS_DDL.ALTER_COMPILE('PACKAGE', 'LHOWARD',
                                                'ENROLLMENT_PACKAGE');
END;
/
```

PL/SQL procedure successfully completed

Oracle Internal Support Packages

Provides underlying functionality of the Oracle database.

This code is placed in packages to enable users to view the package specifications, which are useful for understanding how to use the items, while preventing users from modifying the underlying package body code.

Package Name	Description
STANDARD	Defines all built-in functions and procedures, database data types, and PL/SQL data type extensions
DBMS_SNAPSHOT	Used to manage snapshots, which capture the database state at a specific point in time
DBMS_REFRESH	Used to create groups of snapshots, which can be refreshed simultaneously
DBMS_STANDARD	Contains common processing functions of the PL/SQL language

To call many procedures in commonly used packages like STANDARD and DBMS_STANDARD, you do not need to preface the procedure name with the package name.

For example, the COMMIT command is actually a procedure in the DBMS_STANDARD package.

DYNAMIC SQL

Static SQL commands in PL/SQL programs: structure of the commands have been established and the database objects are validated when the program containing the SQL command is compiled.

Dynamic SQL commands in PL/SQL programs: commands are created as text strings and then compiled and validated at runtime.

Advantage: can dynamically structure SQL queries based on user inputs, and you can include DDL commands like CREATE, ALTER, and DROP in PL/SQL programs.

- Useful when you want to create programs that contain SQL queries that are based on dynamic conditions, such as the current system date or time.
- Allows to create programs that create or alter the structure of database tables. For example, a program may create a temporary database table, use the table to generate a report, and then drop the table.

All Dynamic SQL processing is performed using a cursor that defines the server memory area where the processing takes place.

DBMS_SQL package – contains program units for creating and manipulating the cursor.

Program Unit Name	Description	Type	Input variables
OPEN_CURSOR	Opens the cursor that defines the processing area	Function, returns cursor_ID	None
PARSE	Sends statement to the server, where syntax is verified	Procedure	Cursor_ID, SQL_statements, language_flag, which is set to DBMS_SQL.V7’
BIND_VARIABLE	Associates program variables with input or output values	Procedure	Cursor_ID, variable_name, variable_value, maximum_character_column_size
DEFINE_COLUMN	Specifies the type and length of output variables	Procedure	Cursor_ID, column_position, column_name, maximum_character_column_size
EXECUTE	Executes the Dynamic SQL statement	Function, returns the number of rows fetched	Cursor_ID
FETCH_ROWS	Fetches rows for a SELECT operation	Function, returns the number of rows fetched	Cursor_ID
VARIABLE_VALUE	Retrieves values of output variables	Procedure	Cursor_ID, variable_name, variable_value
COLUMN_VALUE	Associates fetched values with program variables	Procedure	Cursor_ID, column_position, output_variable
CLOSE_CURSOR	Closes the cursor when processing is complete	Procedure	Cursor_ID

Dynamic SQL Programs That Contain DML Commands

Processing dynamic SQL programs that involve DML commands involves the following steps:

1. Open the cursor using the OPEN_CURSOR function.

   `cursor_ID` := **DBMS_SQL.OPEN_CURSOR;**

   The *cursor_id* is a variable that has been declared using the NUMBER data type.

2. Define the SQL command as a text string, using placeholders for dynamic values.

   SQL_command_string_variable := 'SQL_command_text';

   A **placeholder** is a variable that is prefaced with a colon and is not formally declared in the procedure

   SELECT :field_placeholder_name FROM :table_placeholder_name;

3. Parse the SQL command using the PARSE procedure.

   DBMS_SQL.PARSE(cursor_ID, SQL_command_string_variable, language_flag);.

   Thee *language_flag* parameter specifies the version of the DBMS_SQL package being used. Possible values are DBMS_SQL.V6, which is used with Oracle Version 6 databases, and DBMS_SQL.V7 which is used with Oracle Version 7 and higher databases.

4. Bind input variables to placeholders using the BIND_VARIABLE procedure.

   DBMS_SQL.BIND_VARIABLE(cursor_ID, ':placeholder', placeholder_value, maximum_character_column_size);.

   For example, to bind the value STUDENT' to the placeholder *:table_pholder*, you would use

   DBMS_SQL.BIND_VARIABLE(cursor_ID, ':table_pholder','STUDENT',30);.

5. Execute the SQL command using the EXECUTE function.

   number_of_rows_processed := DBMS_SQL.EXECUTE(cursor_ID);.

   The *number_of_rows_processed* return value is a variable that is declared using the NUMBER or INTEGER data type.

6. Close the cursor using the CLOSE_CURSOR procedure.
   DBMAS_SQL.CLOSE_CURSOR(cursor_ID);

To create and execute the Dynamic SQL procedure that uses a DML command

```
CREATE OR REPLACE PROCEDURE update_prices(pct_change IN NUMBER,
                                          curr_item_id IN NUMBER)
AS
    Cursor_id          NUMBER;
    Update_stmt        VARCHAR2(1000);
    Rows_updated       NUMBER;
BEGIN
    --open the cursor
    cursor_id  := DBMS_SQL.OPEN_CURSOR;

    --specify the SQL string using placeholders
    update_stmt := 'UPDATE inventory
                    SET price = price * (1 + :pct_ch)
                    WHERE  item_id = :c_item_id';

    --parse the statement
    DBMS_SQL.PARSE(cursor_id, update_stmt, DBMS_SQL.V7);

    --bind the placeholders to the input parameter variables
    DBMS_SQL.BIND_VARIABLE(cursor_id, ':pct_ch', pct_change);
    DBMS_SQL.BIND_VARIABLE(cursor_id, ':c_item_id', curr_item_id);

    --execute the statement
    rows_updated := DBMS_SQL.EXECUTE(cursor_id);
    --close the cursor
    DBMS_SQL.CLOSE_CURSOR(cursor_id);
END;
/

Procedure created.
```

EXECUTE UPDATE_PRICES(.1, 786);.

SELECT price FROM inventory WHERE item_id = 786;.

Processing Dynamic SQL Programs that contain DDL Commands

Dynamic SQL programs allow CREATE, ALTER, or DROP tables within PL/SQL programs.

You cannot use placeholders in a DDL command, so you cannot dynamically bind parameter values at runtime.

In addition, DDL statements are executed in the PARSE procedure, so the call to the EXECUTE procedure is not needed.

The steps for processing a Dynamic SQL program that uses DDL commands are:

1. Open the cursor
2. Define the SQL command as a text string
3. Parse the SQL command
4. Close the cursor

To create a Dynamic SQL procedure that creates a table:

```
CREATE OR REPLACE PROCEDURE create_temp_table(table_name VARCHAR2)
AS
    Cursor_id       NUMBER;
    Ddl_stmt        VARCHAR2(500);
BEGIN
    --open the cursor
    cursor_id := DBMS_SQL.OPEN_CURSOR;
    --specify the SQL string to create the table
    ddl_stmt := 'CREATE TABLE ' || table_name || '(table_id NUMBER(6))';
    --parse and execute the statement
    DBMS_SQL.PARSE(cursor_id, ddl_stmt, DBMS_SQL.V7);
    --close the cursor
    DBMS_SQL.CLOSE_CURSOR(cursor_id);
END;
/

Procedure created.
```

EXECUTE privilege required on the DBMS_SYS_SQL package in the SYS database schema.

USERNAME: INTERNAL
PASSWORD: ORACLE,

GRANT execute on DBMS_SYS_SQL to PUBLIC;

EXECUTE CREATE_TEMP_TABLE('my_table');

If you are using Personal Oracle and you receive an error stating you have insufficient privileges to create the table, log onto SQL*Plus as username INTERNAL, password ORACLE, and use this account for all Dynamic SQL procedures that require creating a new table.

DESCRIBE my_table;.

Processing Dynamic SQL Programs that contain SELECT Commands

The steps for creating a Dynamic SQL procedure that contains a SELECT command are:

1. Open the cursor
2. Define the SQL command as a text string
3. Parse the SQL command
4. Bind input variables to placeholders
5. Define output variables using the DEFINE_COLUMN procedure. This procedure specifies the variables within the procedure that will be used to reference the retrieved columns. The procedure syntax is

   DBMS_SQL.DEFINE_COLUMN(cursor_id, column_position, variable_name, maximum_character_column_size);

6. Execute the query.
7. Fetch the rows using the FETCH_RECORD function. This function fetches each record into a buffer that temporarily stores the values and has the syntax

   Number_of_rows_left_to_fetch := DBMS_SQL.FETCH_RECORD(cursor_id);

   The FETCH_RECORD function must be processed using a loop, because the query might retrieve multiple records.

   If the FETCH_RECORD function returns the value 0, then the SELECT query has no more rows to return. Therefore, the comparison

   IF DBMS_SQL.FETCH_RECORD(cursor_id) = 0

   is used to test for the loop exit condition.

8. Associate the fetched rows with the output columns using the COLUMN_VALUE procedure.

   DBMS_SQL.COLUMN_VALUE(cursor_id, column_position, variable_name);

9. Close the cursor.

To create a Dynamic SQL procedure that contains a SELECT command:

```
CREATE OR REPLACE PROCEDURE retrieve_consultant_details(curr_c_id IN
NUMBER)
AS
    Cursor_id           NUMBER;
    Select_stmt         VARCHAR2(500);
    Cursor_return_value INTEGER;
    Curr_proj_name      VARCHAR2(30);
    Curr_hours          NUMBER(6);
BEGIN
    --open the cursor
    cursor_id := DBMS_SQL.OPEN_CURSOR;
    --specify the SQL SELECT command
    select_stmt := 'SELECT project_name, total_hours FROM project, project_consultant
                    WHERE project.p_id = project_consultant.p_id AND c_id = :in_c_id';
    --parse the statement
    DBMS_SQL.PARSE(cursor_id, select_stmt, DBMS_SQL.V7);
    --bind the placeholder to the input parameter variable
    DBMS_SQL.BIND_VARIABLE(cursor_id, ':in_c_id', curr_c_id);
    --define the output columns
    DBMS_SQL.DEFINE_COLUMN(cursor_id, 1, curr_proj_name, 30);
    DBMS_SQL.DEFINE_COLUMN(cursor_id, 2, curr_hours);
    --execute the statement
    cursor_return_value := DBMS_SQL.EXECUTE(cursor_id);
    --fetch rows and associate fetched values with output columns
    LOOP
        IF DBMS_SQL.FETCH_ROWS(cursor_id) = 0 THEN
            EXIT;
        END IF;
        DBMS_SQL.COLUMN_VALUE(cursor_id, 1, curr_proj_name);
        DBMS_SQL.COLUMN_VALUE(cursor_id, 2, curr_hours);
        DBMS_SQL.PUT_LINE(curr_proj_name || ' ' || curr_hours);
    END LOOP;
    DBMS_SQL.CLOSE_CURSOR(cursor_id);
END;
/
```

Procedure created.

EXECUTE retrieve_consultant_details(102);

Teller Support System 30
Exploration Database 125

PL/SQL procedure successfully completed.

Using Dynamic SQL to Create an Anonymous PL/SQL Block

This provides a way to use Dynamic SQL in PL/SQL programs when you do not want to create a named program unit or when you do not have the necessary privileges to create a named program units.

This also provides an easier way to process SELECT commands in Dynamic SQL programs when only one record is retrieved.

1. Open the cursor using the OPEN_CURSOR function.
2. Define the PL/SQL block, including the declarations, body, and exception section, is assigned to a text string variable.

   Output variables are defined in the INTO clause of an implicit cursor in the SQL query using placeholders.

   SELECT s_first, s_last INTO :sf_place, :sl_place
   FROM student
   WHERE s_id = :s_id_place;

3. Parse the SQL command using the PARSE procedure.
4. Bind input and output variables to placeholders using the BIND_VARIABLE procedure.
5. Execute the SQL command using the EXECUTE function.
6. Retrieve the values of the output variables using the VARIABLE_VALUE procedure.

   DBMS_SQL.VARIABLE_VALUE(cursor_id, 'placeholder_name', output_variable_name);.

   The *output_variable_name* parameter is a declared variable that has the same data type as the associated output placeholder variable.

7. Close the cursor using the CLOSE_CURSOR procedure.

To create a Dynamic SQL procedure that processes an anonymous PL/SQL block:

```
    CREATE OR REPLACE PROCEDURE retrieve_consultant_hours(curr_c_id IN NUMBER)
    AS
        Cursor_id       NUMBER;
        Block_stmt      VARCHAR2(1000);
        Rows_processed      INTEGER;
        Curr_cons_first     VARCHAR2(30);
        Curr_cons_last      VARCHAR2(30);
        Total_cons_hours    NUMBER(6);
    BEGIN
        Cursor_id := DBMS_SQL.OPEN_CURSOR;
        /* specify the PL/SQL block */
        block_stmt := 'BEGIN
                SELECT c_first, c_last, SUM(total_hours)
                INTO :cons_first, :cons.last, :cons_hours
                FROM consultant, project, project_consultant
                WHERE consultant.c_id = project_consultant.c_id
                AND project.p_id = project_consultant.p_id
                AND consultant.c_id = :in_c_id
                GROUP BY  c_first, c_last;
            END;';
        DBMS_SQL.PARSE(cursor_id, block_stmt, DBMS_SQL.V7);
        /* bind the placeholders to the procedure variables */
        DBMS_SQL.BIND_VARIABLE(cursor_id, ':in_c_id', curr_c_id);
        DBMS_SQL.BIND_VARIABLE(cursor_id, ':cons_first', curr_cons_first, 30);
        DBMS_SQL.BIND_VARIABLE(cursor_id, ':cons_last', curr_cons_last, 30);
        DBMS_SQL.BIND_VARIABLE(cursor_id, ':cons_hours', total_cons_hours);
        Rows_processed := DBMS_SQL.EXECUTE(cursor_id);
        /* retrieve the output variable values */
        DBMS_SQL.VARIABLE_VALUE(cursor_id, ':cons_first', curr_cons_first);
        DBMS_SQL.VARIABLE_VALUE(cursor_id, ':cons_last', curr_cons_last, 30);
        DBMS_SQL.VARIABLE_VALUE(cursor_id, ':cons_hours', total_cons_hours);
        /* output the consultant hours */
        DBMS_OUTPUT.PUT_LINE('Total hours for ' || curr_cons_first || ' ' || curr_cons_last ||
                                                                    ': ' || total_cons_hours);
        DBMS_SQL.CLOSE_CURSOR(cursor_id);
END;
/

    EXECUTE retrieve_consultant_hours(102);
```

Total hours for Brian Zhang: 155