**CHAPTER SIX ROW TRIGGER MECHANIC**

**¿What’s a trigger?**

It is a compiled procedure stored in a database, you code and compile a trigger as procedures.

The trigger body is written in PL/SQL. Some statements only works inside a trigger body. These topics are covered in the following paragraphs.

A trigger body can refer to two states of a column value. The general syntax becomes as follows.

**:NEW.COLUMN\_NAME**

**:OLD.COLUMN\_NAME**

If the statement does not refer to a column, the next rules apply:

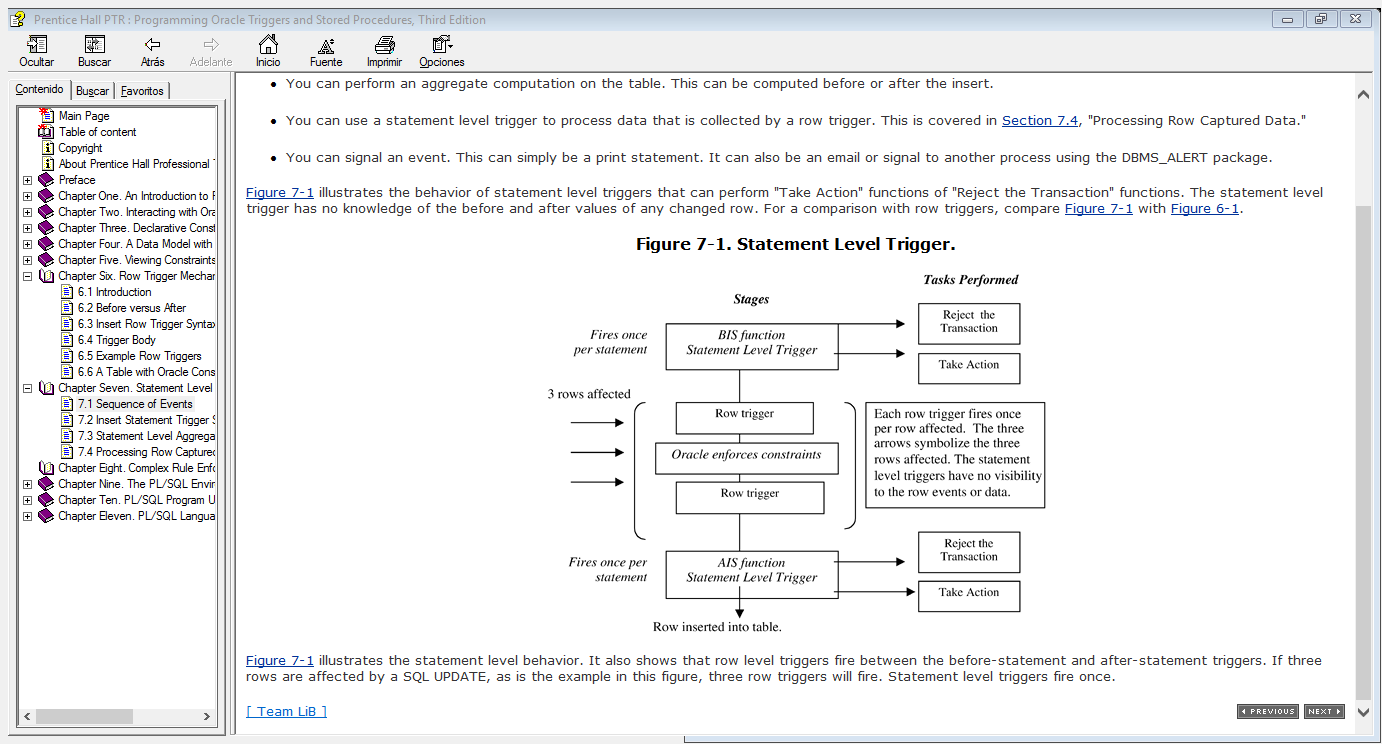
* For **INSERT** statements, :**NEW.COLUMN\_NAME** is either NULL or the value of the DEFAULT used in creating the table.
* For UPDATE statements, **:NEW.COLUMN\_NAME** is the value currently in the table.

On the other hand, we explain what OLD.COLUMN\_NAME is.

This sentence is valid for **UPDATE** and **DELETE** row triggers only. This evaluates to NULL for INSERT triggers. The **:OLD.COLUMN\_NAME** evaluates to the value currently in the table. The values of **:OLD.COLUMN\_NAME** and :NEW.COLUMN\_NAME expressions are identical for **BEFORE** and **AFTER** row triggers. The choice of a BEFORE or AFTER row trigger is a choice to execute the trigger before or after Oracle constraint enforcement. Visibility to **:OLD** and NEW values is identical with each. One caveat is that you can change the value of **:NEW.COLUMN\_NAME** in **BEFORE** row triggers, not in **AFTER** row triggers.

**CHAPTER SEVEN ROW TRIGGER MECHANIC**

|  |
| --- |
| **Sequence of Events**  You can do the following with an INSERT STATEMENT trigger:   * You can perform an aggregate computation on the table. This can be computed before or after the insert. * You can use a statement level trigger to process data that is collected by a row trigger. * You can signal an event. This can simply be a print statement. It can also be an email or signal to another process using the DBMS\_ALERT package. |

For a better understanding we can let us guide by the next diagram.

As recently talked about the INSERT statement. We can lead us with the next general syntax.

CREATE OR REPLACE TRIGGER trigger\_name

[AFTER | BEFORE] INSERT ON table\_name

DECLARE

Local declarations

BEGIN

Body written PL/SQL

END;

The key difference in the syntax between the statement and row trigger is the FOR EACH ROW clause—this clause specifically identifies the trigger as row level and is not in the statement level

The following are valid clauses for statement level triggers, as well as row triggers.

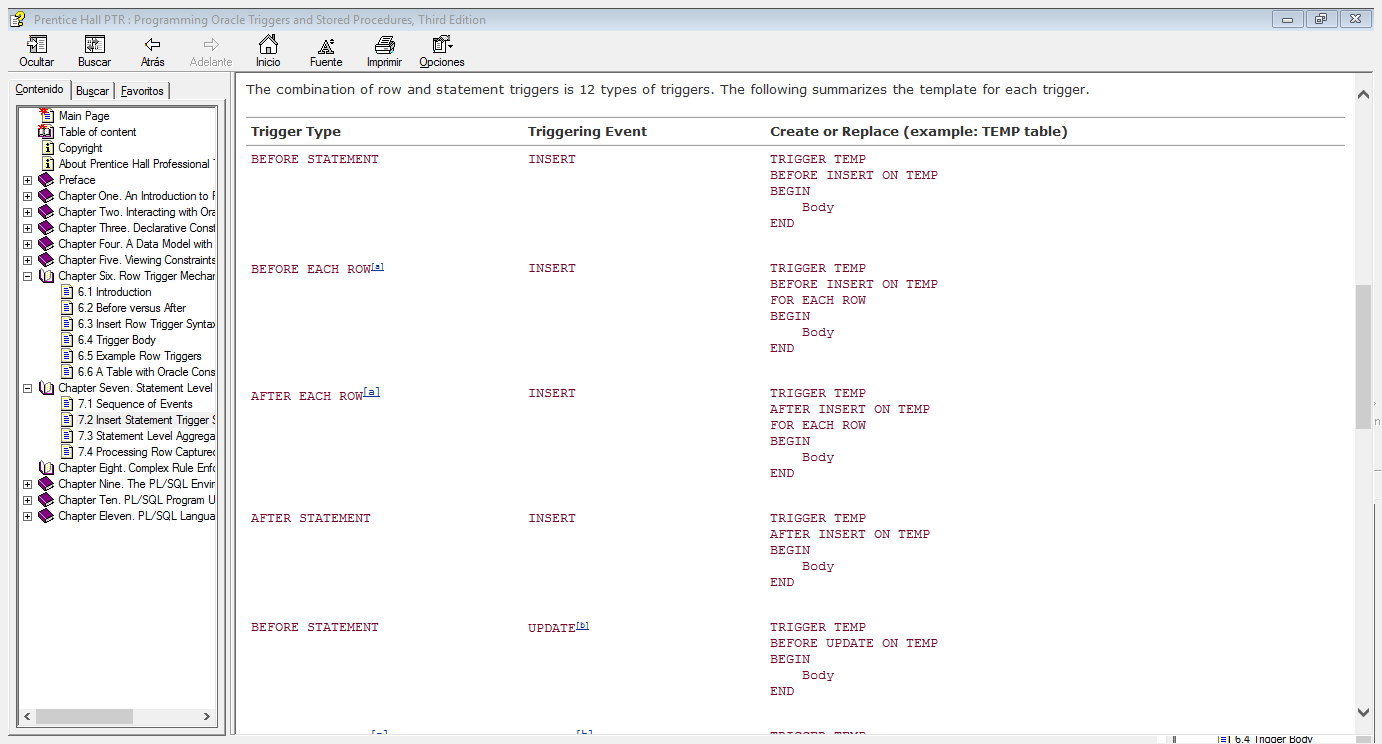
BEFORE INSERT OR UPDATE OR DELETE ON table\_name

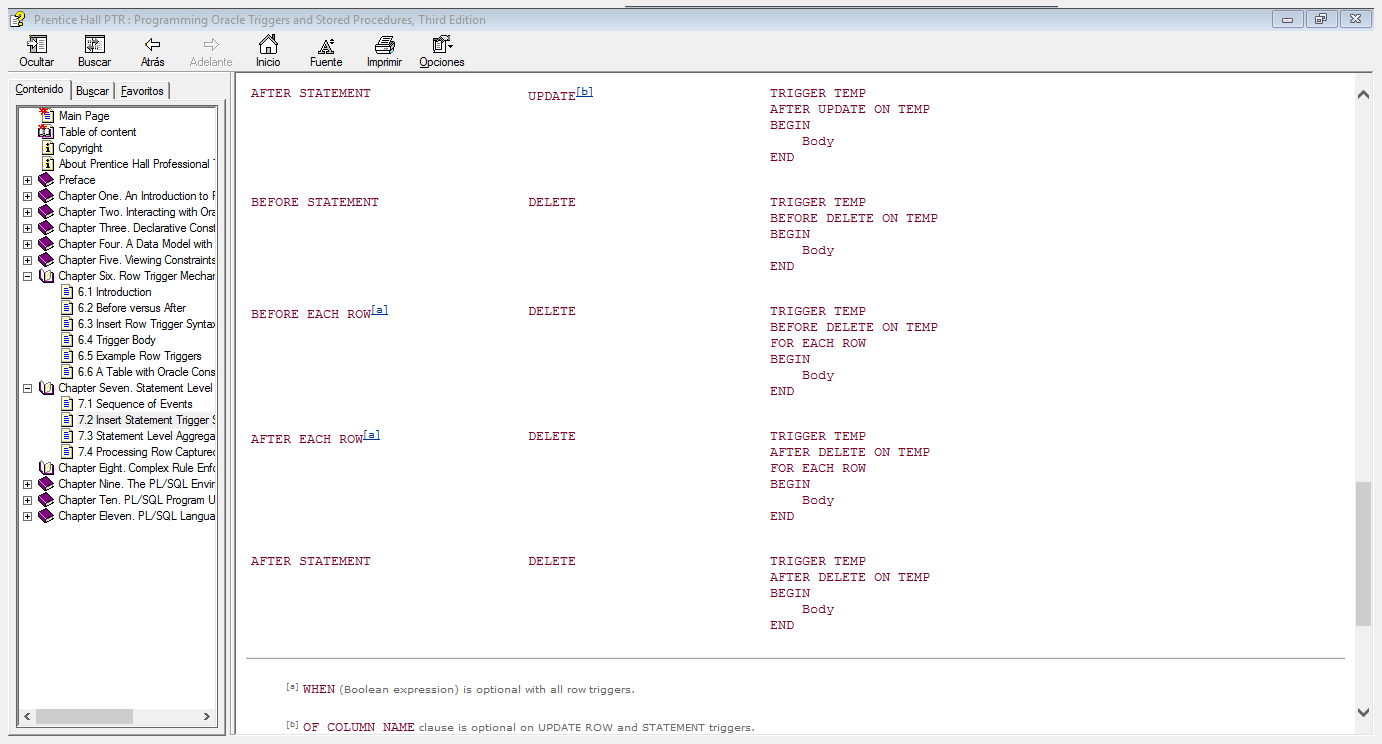
AFTER INSERT OR UPDATE OF column\_name OR DELETE ON table\_name

The following are two key points with regard to trigger options:

|  | |
| --- | --- |
| * WHEN (Boolean expression) | ALL ROW triggers only. |
| * OF column\_name clause | Valid for UPDATE ROW and STATEMENT triggers only. |

The syntax for statement level triggers is simpler than row triggers. The following features do not exist with statement level triggers:

* There is no WHEN (Boolean expression) that voids any discussion of OLD.COLUMN\_NAME and NEW.COLUMN\_NAME expressions.
* References to :NEW.COLUMN\_NAME and :OLD.COLUMN\_NAME are invalid. The combination of row and statement triggers is 12 types of triggers.





**CHAPTER EIGHT ROW COMPLEX RULE ENFORCEMENT**

This chapter talks about an specific case to expand your knowledge and show a typical case in real. The following scenario is based on the data in the STUDENT\_VEHICLES table and the PARKING\_TICKETS table.

A business rule can sometimes have a recursive nature. A general scenario is when an UPDATE statement executes and an update trigger modifies other rows in that same table. Deletes can also be recursive. Certainly, the foreign key delete cascade is one method of deleting dependent data. Other times, the business rule is complex and a delete trigger must procedurally determine if additional deletes are required.

The following scenario demonstrates a recursive delete. The delete row trigger may delete additional rows. It may not. It depends on the data. The business rule is:

When a student pays a parking ticket, all other tickets for the same car and for the same amount are deleted as well, provided the sum of the tickets to be deleted does not exceed $10.00.

The rationale is that tickets are sometimes duplicated. The assumption is that multiple tickets for the same amount and the same car are accidental duplicates. If the sum of the tickets is excessive, then this is probably not the case. Hence the limit of $10.00.

A student can have three tickets, each $3.00. All are deleted when one ticket is paid. The vehicle with New York tag number MH 8709 has a duplicate parking violation. Each ticket is $5.00. According to this rule, one paid ticket deletes both parking tickets from the table.

The task is to develop a trigger that takes the specific action to delete additional records within the same table. On example code below we are able to appreciate steps to achieve this.

The following is the specification and body of the trigger interface package. This is called the PARKING\_TICKETS\_CONS package. This code accepts row data from the row trigger. For statement level processing it simply prints the data read from the temporary table. This will be revised shortly to include the deletes.

CREATE OR REPLACE PACKAGE parking\_tickets\_cons IS

PROCEDURE load\_temp\_table

(v\_state parking\_tickets.state%TYPE,

v\_tag\_no parking\_tickets.tag\_no%TYPE,

v\_amount parking\_tickets.amount%TYPE);

PROCEDURE remove\_duplicates;

END parking\_tickets\_cons;

The package body is:

CREATE OR REPLACE PACKAGE BODY parking\_tickets\_cons IS

PROCEDURE load\_temp\_table

(v\_state parking\_tickets.state%TYPE,

v\_tag\_no parking\_tickets.tag\_no%TYPE,

v\_amount parking\_tickets.amount%TYPE)

IS

BEGIN

INSERT INTO parking\_tickets\_g VALUES

(v\_state, v\_tag\_no, v\_amount);

END load\_temp\_table;

PROCEDURE remove\_duplicates IS

BEGIN

FOR rec in(SELECT \* FROM parking\_tickets\_g) LOOP

dbms\_output.put\_line

('REC:'||rec.state

||' '||rec.tag\_no||' '||rec.amount);

END LOOP;

END remove\_duplicates;

END parking\_tickets\_cons;

The row trigger will call the aforementioned procedure LOAD\_TEMP\_TABLE. The statement level trigger will call REMOVE\_DUPLICATES, which for now prints the rows selected from the temporary table.

The row and statement level triggers are:

CREATE OR REPLACE TRIGGER parking\_tickets\_adr

AFTER DELETE ON parking\_tickets

FOR EACH ROW

BEGIN

parking\_tickets\_cons.load\_temp\_table

(:old.state, :old.tag\_no, :old.amount);

END;

CREATE OR REPLACE TRIGGER parking\_tickets\_ads

AFTER DELETE ON parking\_tickets

BEGIN

parking\_tickets\_cons.remove\_duplicates;

END;

If we execute the following SQL statement, the vehicle information is printed.

SQL> DELETE FROM parking\_tickets

2 WHERE ticket\_no = ='P\_02';

The final step is to revise the logic in REMOVE\_DUPLICATES. This procedure must query the PARKING\_STUDENTS table to determine if there are other rows for the same vehicle and same ticket amount.

One row is already deleted. To determine if the sum of tickets is within the $10.00 range, the ticket amount (of the deleted row) must be added to the SUM(amount) left in the table. Now the duplicates are removed. The finally table is as follows.}

TICKET\_NO AMOUNT ST TAG\_NO

---------- ---------- -- ----------

P\_01 5 CA CD 2348

P\_02 5 NY MH 8709

P\_03 5 NY MH 8709

P\_04 5 NY JR 9837

SQL> DELETE FROM parking\_tickets WHERE ticket\_no='P\_02';

SQL> SELECT \* FROM parking\_tickets;

TICKET\_NO AMOUNT ST TAG\_NO

---------- ---------- -- ----------

P\_01 5 CA CD 2348

P\_04 5 NY JR 9837

**CHAPTER NINE. THE PL/SQL ENVIRONMENT**

Use PL/SQL blocks as test drivers for stored procedures. There is nothing different about the code in a PL/SQL block and the code in stored procedure—both use PL/SQL. Enhance test driver code with exception handling code to display an exception error number and error message. The PL/SQL block below includes a WHEN OTHERS exception handler that prints the exception to a duplicate insert. First, create a table with a primary key constraint.

CREATE TABLE TEMP(N NUMBER CONSTRAINT PK\_TEMP PRIMARY KEY);

TEST\_TEMP.SQL is the name of the PL/SQL block and includes an exception handler. In contains 11 lines, including an initial comment and a forward slash as the last line, which must be in Column 1.

-- Filename: TEST\_TEMP.SQL

BEGIN

INSERT INTO temp VALUES (1);

INSERT INTO temp VALUES (1);

EXCEPTION

WHEN OTHERS THEN

dbms\_output.put\_line('Error code:'||SQLCODE||'\*\*\*');

dbms\_output.put\_line

('Error message:'||SQLERRM||'\*\*\*');

END;

/

In this PL/SQL block, the second insert fails with a primary key constraint violation. The code in the exception handler uses DBMS\_OUTPUT to print the error number and message.

The database administrator compiles many PL/SQL packages into the Oracle SYS account. This is part of creating the database. The database creation process also creates public synonyms and public grants for these packages. These packages provide a robust API.

As you write PL/SQL you display output using the DBMS\_OUTPUT package. This is one component of the API, and for most programmers, the most frequently used package during development..

The following illustrates the SYS statements executed, during database creation, that make DBMS\_OUTPUT available for general use.

1.GRANT EXECUTE ON DBMS\_OUTPUT TO PUBLIC;

2.CREATE PUBLIC SYNONYM DBMS\_OUTPUT FOR SYS.DBMS\_OUTPUT;

The first statement means that SCOTT and all future Oracle accounts can write PL/SQL that use the DBMS\_OUTPUT package. The keyword PUBLIC gives the grant to all users. The second statement means that SCOTT can write PL/SQL with statements like:

DBMS\_OUTPUT.PUT\_LINE('Hello');

If we want to know all packages in the API that begin with DBMS or UTL, query the data dictionary view ALL\_OBJECTS. We just write this

set pagesize 0

set term off

set feedback off

spool all\_objects

SELECT object\_name

FROM all\_objects

WHERE owner='SYS' AND

object\_type='PACKAGE'

AND (object\_name like 'DBMS%' OR object\_name like 'UTL%');

set feedback on

set term on

spool off

**USER PROJECTS**

The USER\_OBJECTS view provides status information on objects you create. This includes tables, sequences, views, stored procedures, database links, and others. The following is a partial description of columns from this view. Use this view to determine if a stored procedure is valid, if you need to recompile it, or to determine its last compile timestamp.

* jhUSER\_OBJECTS provides information only on those objects you have created in your account.
* ALL\_OBJECTS provides information on objects you have created plus objects to which you have privileges.
* DBA\_OBJECTS provides information on all objects in the database. You need the Oracle role DBA or SELECT\_CATALOG\_ROLE to access DBA views.

Because the scope of DBA views is everything in the database, you must have either the Oracle DBA role or the Oracle SELECT\_CATALOG\_ROLE role. The DBA role has high privileges. SELECT\_CATALOG\_ROLE is intended for users who need to query data dictionary views. Application developers should be given this role.

A procedure you create will have an entry in USER\_OBJECTS. If BLAKE creates a procedure HELLO\_BLAKE and grants execute on that procedure to you, then you can see this object when you query OWNER, OBJECT\_NAME, and OBJECT\_TYPE from ALL\_OBJECTS.

However it’s mandatory we talk about when something get wrong. For example sometimes processes are executed even with errors, but we do not know which are.

SHOW ERRORS is a SQL\*Plus command. It returns information from the USER\_ERRORS view, which has the following description:

SQL> desc user\_errors

Name Null? Type

------------------------------ -------- ----------------

NAME NOT NULL VARCHAR2(30)

TYPE VARCHAR2(12)

SEQUENCE NOT NULL NUMBER

LINE NOT NULL NUMBER

POSITION NOT NULL NUMBER

TEXT NOT NULL VARCHAR2(4000)

You can select compiler error results when not running SQL\*Plus by querying this view. The USER\_ERRORS view contains errors for objects compiled in your schema. ALL\_ERRORS and DBA\_ERRORS are other views with wider scope. Refer to [Chapter 5](mk:@MSITStore:E:\DataBaseESCOM\(prentice%20hall%202003)%20-%20programming%20oracle%20triggers%20and%20stored%20procedures,%203rd%20ed.chm::/0130850330_ch05.html#ch05) for a complete description of the differences between the USER, ALL, and DBA data dictionary views. The following is a description of the columns.

| NAME | This is the name in the CREATE OR REPLACE clause. This is not the host file. Running the script @MY\_HELLO.SQL with a CREATE OR REPLACE PROCEDURE HELLO statement creates the object name HELLO. The data dictionary stores all attributes in upper case. |
| --- | --- |
| TYPE | This is FUNCTION, PROCEDURE, PACKAGE, or PACKAGE BODY. There is never an underscore in PACKAGE BODY. |
| SEQUENCE | This corresponds to the error number relative to the number of errors in the compile. |
| LINE | This is the list-file line number, which contains the error. This corresponds to the LINE column in USER\_SOURCE. |
| POSITION | This is the column position of the error. |
| TEXT | This contains the text of the error. For example:  PLS-00302: component 'PUTLINE' must be declared |

**CHAPTER TEN. PL/SQL PROGRAM UNITS**

Is not necessary to go deep into this concepts ‘cause we have recently studied in previous books. But we should remember this concepts to get sure, we learned about components of SQL

PROCEDURE

A PL/SQL procedure is a stand-alone program that you compile into an Oracle database schema. Procedures can accept arguments. When you compile a procedure, the procedure identifier of the CREATE PROCEDURE statement becomes the object name in the data dictionary. For adding we talk about EXCEPTION HANDLER The handler is similar to the try-catch model in other languages. You can code an exception handler for a specific type of error or write a general-purpose exception handler.

Also we should know the subprogram encapsulation. This are called packages, an they Packages provide a mechanism to logically group smaller program units together. The grouping of procedures into a package modularizes the code. Physically, the use of packages means fewer files and fewer modules to manage.

Programmers who begin working on the system, built with packages, quickly grasp the overall architecture because they can learn the system

The basic package specification syntax is:

CREATE PACKAGE package\_name IS

Type definitions for records, index-by tables,

varrays, nested tables

Constants

Exceptions

Global variable declarations

PROCEDURE procedure\_name\_1 (parameters & types);

PROCEDURE procedure\_name\_2 (parameter & types);

FUNCTION function\_name\_1 (parameters & types) RETURN type;

END package\_name;

Looking for another concept to clarify and exploit SQL we find the application partitioning.

The design of a system begins with partitioning functionality into subcomponents, or subsystems. The process continues with partitioning the subsystem into packages. A single package within a subsystem satisfies a set of related requirements. A subsystem will likely include Oracle built-in packages (e.g., UTL\_FILE for File IO). A package can logically reside in multiple subsystems, also called applications.

We can implement the concept of a subsystem using the existing database framework, specifically the database schema. To make a subsystem, we can encapsulate packages as objects within a schema and use database grants to make some packages visible and some not. The packages, for which we gave grants, become the API of the subsystem. A programmer might develop a useful and general purpose dates package that would exist in numerous subsystems.

Finally we want to show you what overloading is, and how important is this.

You can overload the procedure or function in a package. The best example of overloading is the Oracle built-in DBMS\_OUTPUT package. It provides a print procedure to display a string:

PUT\_LINE(parameter VARCHAR2)

Other types can be printed including a DATE and NUMBER type. This leads to overloaded procedures:

PUT\_LINE(parameter VARCHAR2)

PUT\_LINE(parameter DATE)

PUT\_LINE(parameter NUMBER)

You overload procedures when identical functionality is performed on different types. You can technically overload procedures that perform unrelated functions. This practice makes code that uses your package vague, hard to read, and hard to understand.

Subprograms within a package can be overloaded only when the parameter profile is different. Oracle must be able to determine which procedure you call. The following package contains overloaded procedures. This example is not valid because the base type of each parameter is the same.

CREATE OR REPLACE PACKAGE sample\_pkg IS

PROCEDURE process(v\_value IN NUMBER);

PROCEDURE process(v\_value IN INTEGER);

END sample\_pkg;

You can successfully compile this package but Oracle will not know which procedure to invoke if you make the following call.

sample\_pkg.process(2);

This statement will fail with the Oracle message:

PLS-00307: too many declarations of 'PROCESS' match this call.

A modification to the SAMPLE\_PKG package corrects this problem. The following code defines two procedures, each with a unique profile. This is a valid example of overloading.

CREATE OR REPLACE PACKAGE sample\_pkg IS

PROCEDURE process(v\_value NUMBER);

PROCEDURE process(v\_value INTEGER, v\_date DATE);

END sample\_pkg;

Differences in MODE do not constitute differences in profile. The following is invalid.

CREATE OR REPLACE PACKAGE sample\_pkg IS

PROCEDURE process(v\_value IN NUMBER);

PROCEDURE process(v\_value OUT NUMBER);

END sample\_pkg;

Differences in formal parameter names do not make the procedures unique. The following will compile but the caller must use named notation. Logically, this conflicts with the concept of overloading, which is to be able to perform similar operations of different datatypes.

CREATE OR REPLACE PACKAGE sample\_pkg IS

PROCEDURE process(v\_new\_salary IN NUMBER);

PROCEDURE process(v\_old\_salary IN NUMBER);

END sample\_pkg;

For this package, the first call works, but the second does not. Oracle will raise a run-time error because it cannot determine which procedure you want to call.

sample\_pkg.process(v\_new\_salary=>9000);

sample\_pkg.process(9000);

Overloading works when the parameter profiles are different, which includes the evaluation of the base types in conjunction with default parameters declared in the specification.