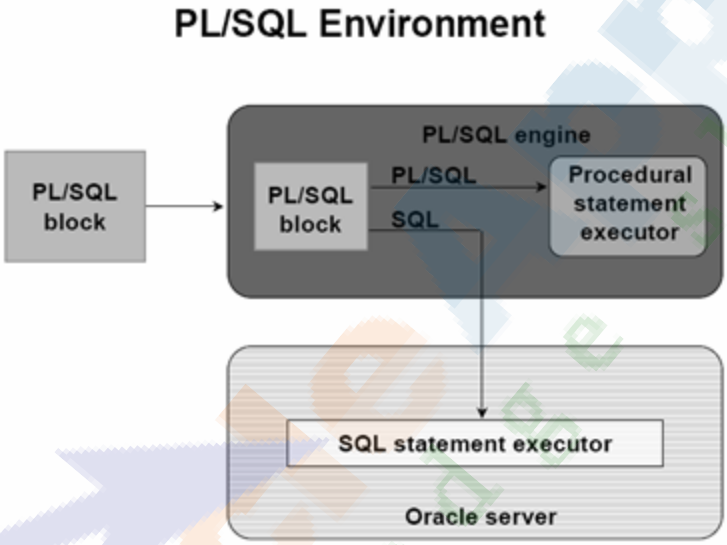
Basics of PL/SQL

PL/SQL stands for Procedural Language extension of SQL.
PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

PL/SQL Environment

PL/SQL is not an Oracle product in its own right; it is a technology used by the Oracle server and by certain Oracle tools. Blocks of PL/SQL are passed to and processed by a PL/SQL engine, which may reside within the tool or within the Oracle server. The engine that is used depends on where the PL/SQL block is being invoked from. When you submit PL/SQL blocks from a Oracle precompiler such as Pro*C or Pro*Cobol program, userexit, iSQL*Plus, or Server Manager, the PL/SQL engine in the Oracle Server processes them. It separates the SQL statements and sends them individually to the SQL statements executor.



A single transfer is required to send the block from the application to the Oracle Server, thus improving performance, especially in a client-server network. PL/SQL code can also be stored in the Oracle Server as subprograms that can be referenced by any number of applications connected to the database.

Many Oracle tools, including Oracle Developer, have their own PL/SQL engine, which is independent of the engine present in the Oracle Server. The engine filters out SQL statements and sends them individually to the SQL statement executor in the Oracle server. It processes the remaining procedural statements in the procedural statement executor, which is in the PL/SQL engine. The procedural statement executor processes data that is local to the application (that is, data already inside the client environment, rather than in the database). This reduces the work that is sent to the Oracle server and the number of memory cursors that are required.

Advantages of PL/SQL

These are the advantages of PL/SQL.

Block Structures: PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.

Procedural Language Capability: PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).

Better Performance: PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.

Error Handling: PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

Architecture

The PL/SQL language is a robust tool with many options. PL/SQL lets you write code once and deploy it in the database nearest the data. PL/SQL can simplify application development, optimize execution, and improve resource utilization in the database.
The language is a case-insensitive programming language, like SQL. This has led to numerous formatting best practice directions. Rather than repeat those arguments for one style or another, it seems best to recommend you find a style consistent with your organization's standards and consistently apply it. *The PL/SQL code in this book uses uppercase for command words and lowercase for variables, column names, and stored program calls*

PL/SQL also supports building SQL statements at run time. Run-time SQL statements are dynamic SQL. You can use two approaches for dynamic SQL: one is Native Dynamic SQL (NDS) and the other is the DBMS_SQL package. The Oracle 11g Database delivers new NDS features and improves execution speed. With this release, you only need to use the DBMS_SQL package when you don't know the number of columns that your dynamic SQL call requires. Chapter 11 demonstrates dynamic SQL and covers both NDS and the DBMS_SQL package.

PL/SQL Block

PL/SQL is a block-structured language, meaning that programs can be divided into logical blocks. Program units can be named or unnamed blocks. Unnamed blocks are known as anonymous blocks. The PL/SQL coding style differs from that of the C, C++, and Java programming languages. For example, curly braces do not delimit blocks in PL/SQL.
A PL/SQL block consists of up to three sections: **declarative (optional), executable (required),** and **exception handling (optional).**
Note: In PL/SQL, an error is called an exception.

PL/SQL Block Structure

DECLARE (Optional)

Variables, cursors, user-defined exceptions

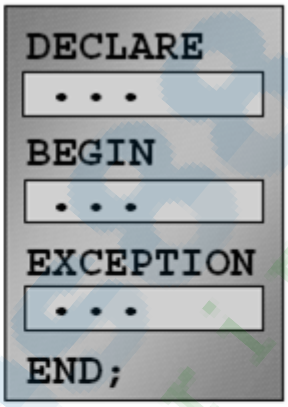
BEGIN (Mandatory)

- SQL statements
- PL/SQL statements

EXCEPTION (Optional)

Actions to perform when errors occur

END; (Mandatory)



Section	Description	Inclusion
Declarative	Contains all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and declarative sections	Optional
Executable	Contains SQL statements to manipulate data in the database and PL/SQL statements to manipulate data in the block	Mandatory
Exception handling	Specifies the actions to perform when errors and abnormal conditions arise in the executable section	Optional

Executing Statements

```
DECLARE v_variable VARCHAR2(5);
BEGIN
SELECT column_name INTO v_variable FROM table_name;
EXCEPTION
WHEN exception_name THEN
...
END;
```

- Place a semicolon (;) at the end of a SQL statement or PL/SQL control statement.
- Section keywords DECLARE, BEGIN, and EXCEPTION are not followed by semicolons. END and all other PL/SQL statements require a semicolon to terminate the statement.

Block Types

A PL/SQL program comprises one or more blocks. These blocks can be entirely separate or nested one within another. The basic units (procedures and functions, also known as subprograms, and anonymous blocks) that make up a PL/SQL program are logical blocks, which

can contain any number of nested subblocks. Therefore, one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Anonymous

```
[DECLARE]

BEGIN
   --statements

[EXCEPTION]

END;
```

Procedure

```
PROCEDURE name
IS

BEGIN
   --statements

[EXCEPTION]

END;
```

Function

```
FUNCTION name
RETURN datatype
IS
BEGIN
   --statements
   RETURN value;
[EXCEPTION]

END;
```

PL/SQL Placeholders

Placeholders are temporary storage area. Placeholders can be any of **Variables**, **Constants** and **Records**. Oracle defines placeholders to store data temporarily, which are used to manipulate data during the execution of a PL SQL block.

Depending on the kind of data you want to store, you can define placeholders with a name and a datatype. Few of the datatypes used to define placeholders are as given below. Number (n,m) , Char (n) , Varchar2 (n) , Date , Long , Long raw, Raw, Blob, Clob, Nclob, Bfile

Place holders are used for

• Temporary storage of data, • Manipulation of stored values, • Reusability, • Ease of maintenance

Declaring PL/SQL Variable

Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]
     [:= | DEFAULT expr];
```

Examples:

```
DECLARE
  v_hiredate      DATE;
  v_deptno        NUMBER(2) NOT NULL := 10;
  v_location      VARCHAR2(13) := 'Atlanta';
  c_comm          CONSTANT NUMBER := 1400;
```

Handling Variables in PL/SQL

- Declare and initialize variables in the declaration section.
- Assign new values to variables in the executable section.
- Pass values into PL/SQL blocks through parameters.
- View results through output variables.

Types of PL/SQL Variables

All PL/SQL variables have a data type, which specifies a storage format, constraints, and valid range of values. PL/SQL supports four data type categories—scalar, composite, reference, and LOB (large object)—that you can use for declaring variables, constants, and pointers.

1. Scalar data types hold a single value. The main data types are those that correspond to column types in Oracle server tables; PL/SQL also supports Boolean variables.
2. Composite data types, such as records, allow groups of fields to be defined and manipulated in PL/SQL blocks.
3. Reference data types hold values, called pointers, that designate other program items. Reference data types are not covered in this course.
4. LOB data types hold values, called locators, that specify the location of large objects (such as graphic images) that are stored out of line. LOB data types are discussed in detail later in this course.

Non-PL/SQL variables include host language variables declared in precompiler programs, screen fields in Forms applications, and iSQL*Plus host variables.

Quick Notes - Variable Declaration

1. The rules for identifiers are same as for SQL objects.
2. NOT NULL/CONSTANT may be optionally used
3. Only one identifier per line is allowed .

DECLARE

```
   firstname    lastname  CHAR(20)  ; - illegal
DECLARE
   firstname    CHAR(20)  ; -legal
   lastname     CHAR(20)  ; - legal
```

Variable Declarations Examples

```
NUMBER
Count                NUMBER;
revenue              NUMBER (9,2);
second_per_day       CONSTANT NUMBER := 60*24 ;
running_total        NUMBER (10,0) := 0;

VARCHAR2
mid_initial          VARCHAR2 := 'K';
last_name            VARCHAR2(10) NOT NULL;
company_name         CONSTANT VARCHAR2(12);
```

```
DATE
anniversary          DATE := '05-NOV-78';
project_complexion   DATE;
next_checkup         DATE NOT NULL := '28-JUN-90';

BOOLEAN
over_budget          BOOLEAN   NOT NULL  := FALSE;
available            BOOLEAN   := NULL ;
```

Attribute Declaration

PL/SQL objects (such as variables and constants) and database objects (such as columns and tables) are associated with certain attributes.

%TYPE attribute

```
DECLARE
  books_printed      NUMBER (6);
  books_sold         books.sold%TYPE ;
  maiden_name        emp.ename%TYPE ;
```

%ROWTYPE attribute

```
DECLARE
  dept_row       dept%ROWTYPE ;
```

Assignments

Variables and constants are initialized every time a block or subprogram is entered.

By default, variables are initialized to NULL. So, unless you expressly initialize a variable, its value is undefined, as the following example shows:

```
DECLARE
count INTEGER;
...
BEGIN
count := count + 1; -- assigns a null to count
```

The expression on the right of the assignment operator yields NULL because count is null. To avoid unexpected results, never reference a variable before you assign it a value.

You can use assignment statements to assign values to a variable. For example, the following statement assigns a new value to the variable bonus, overwriting its old value:

```
bonus := salary * 0.15;
```

The expression following the assignment operator can be arbitrarily complex, but it must yield a datatype that is the same as or convertible to the datatype of the variable.

Boolean Values

Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable. For example, given the declaration

```
DECLARE
done BOOLEAN;
```

the following statements are legal:

```
BEGIN
done := FALSE;
WHILE NOT done LOOP
...
END LOOP;
```

When applied to an expression, the relational operators return a Boolean value. So, the following assignment is legal:

```
done := (count > 500);
```

Expressions and Comparisons

Database Values

You can use the SELECT statement to have Oracle assign values to a variable. For each item in the select list, there must be a corresponding, type-compatible variable in the INTO list. An example follows:

```
DECLARE
my_empno emp.empno%TYPE;
my_ename emp.ename%TYPE;
wages NUMBER(7,2);
BEGIN
...
SELECT ename, sal + comm
INTO last_name, wages FROM emp
WHERE empno = emp_id;
```

However, you cannot select column values into a Boolean variable.

Quick notes -Assignment

1. := (ASSIGNMENT) whereas = (VALUE EQUALITY)
2. The datatype of the left and right hand side of an assignment must be the same or implicitly convertible to each other.

For ex. , N:='7' is legal because number may be implicitly converted to char.

3. Column or table reference are not allowed on either side of an assignment operator(: =).

```
SCOTT.EMP.EMPNO := 1234;
location := dept.loc.;
```

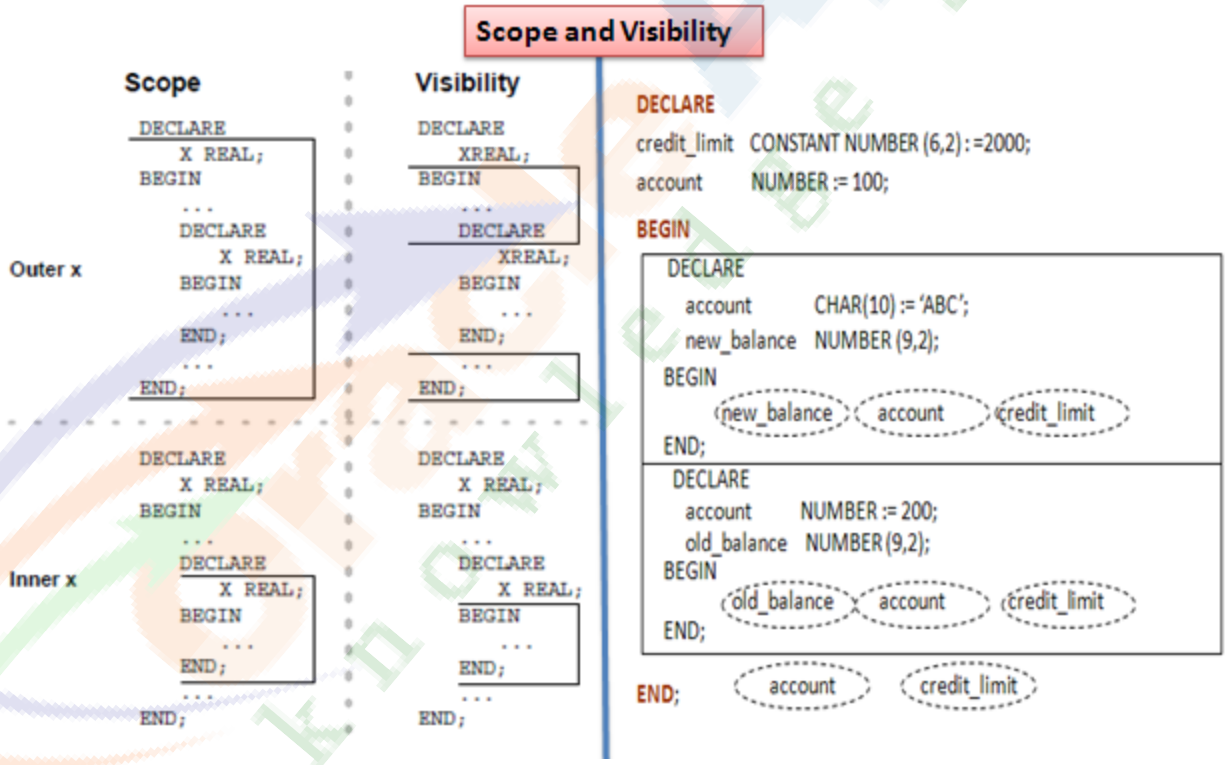
Above two are **incorrect**.

Scope and Visibility

References to an identifier are resolved according to its scope and visibility. The scope of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier. An identifier is visible only in
the regions from which you can reference the identifier using an unqualified name. Below Figure shows the scope and visibility of a variable named x, which is declared in an enclosing block, then redeclared in a sub-block.

Identifiers declared in a PL/SQL block are considered local to that block and global to all its sub-blocks. If a global identifier is redeclared in a sub-block, both identifiers remain in scope. Within the sub-block, however, only the local identifier is visible because you must use a qualified name to reference the global identifier.

Although you cannot declare an identifier twice in the same block, you can declare the same identifier in two different blocks. The two items represented by the identifier are distinct, and any change in one does not affect the other. However, a block cannot reference identifiers declared in other blocks at the same level because those identifiers are neither local nor global to the block.



Control Structures

This chapter shows you how to structure the flow of control through a PL/SQL program. You learn how statements are connected by simple but powerful control structures that have a single

entry and exit point. Collectively, these structures can handle any situation. Their proper use leads naturally to a well-structured program.

IF Statements

Often, it is necessary to take alternative actions depending on circumstances. The IF statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.

```
IF condition THEN
sequence_of_statements
END IF;
```

```
IF condition THEN
sequence_of_statements1
ELSE
sequence_of_statements2
END IF;
```

```
IF condition1 THEN
sequence_of_statements1
ELSIF condition2 THEN
sequence_of_statements2
ELSE
sequence_of_statements3
END IF;
```

Iterative Control: LOOP and EXIT Statements

LOOP statements let you execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use an EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

```
IF condition THEN
EXIT ;
END IF ;
```

```
EXIT WHEN condition ;
```

Examples:

1.
```
LOOP
...
IF credit_rating < 3 THEN
EXIT; -- exit loop immediately
END IF;
END LOOP;
-- control resumes here
```

```
2. LOOP
FETCH c1 INTO ...
EXIT WHEN c1%NOTFOUND; -- exit loop if condition is true
...
END LOOP;
CLOSE c1;
```

Loop Labels

Like PL/SQL blocks, loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement, as follows:

```
<<label_name>>
LOOP
sequence_of_statements
END LOOP;
```

Optionally, the label name can also appear at the end of the LOOP statement, as the following example shows:

```
<<my_loop>>
LOOP
...
END LOOP my_loop;
```

When you nest labeled loops, you can use ending label names to improve readability.
With either form of EXIT statement, you can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete. Then, use the label in an EXIT statement, as follows:

```
<<outer>>
LOOP
...
LOOP
...
EXIT outer WHEN ... -- exit both loops
END LOOP;
...
END LOOP outer;
```

Every enclosing loop up to and including the labeled loop is exited.

WHILE-LOOP

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition
LOOP
sequence_of_statements
END LOOP;
```

```
WHILE total <= 25000
LOOP
SELECT sal INTO salary FROM
emp WHERE ...
total := total + salary;
END LOOP;
```

Before each iteration of the loop, the condition is evaluated. If the condition is true, the sequence of statements is executed, then control resumes at the top of the loop. If the condition is false or null, the loop is bypassed and control passes to the next statement.
The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times. In the last example, if the initial value of total is larger than
25000, the condition is false and the loop is bypassed.

FOR-LOOP

Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. (Cursor FOR loops iterate over the result set of a cursor, are discussed in later section) The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP. A double dot (..) serves as the range operator. The syntax follows:

```
FOR i IN [REV.] LB..HB
LOOP
sequence_of_statements
END LOOP;
```

The range is evaluated when the FOR loop is first entered and is never re-evaluated.
As the next example shows, the sequence of statements is executed once for each integer in the range. After each iteration, the loop counter is incremented.

```
FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
sequence_of_statements -- executes three times
END LOOP;
```

The following example shows that if the lower bound equals the higher bound, the sequence of statements is executed once:

```
FOR i IN 3..3 LOOP -- assign the value 3 to i
sequence_of_statements -- executes one time
END LOOP;
```

By default, iteration proceeds upward from the lower bound to the higher bound. However, as the example below shows, if you use the keyword REVERSE, iteration proceeds downward from

the higher bound to the lower bound. After each
iteration, the loop counter is decremented.

```
FOR i IN REVERSE 1..3 LOOP -- assign the values 3,2,1 to i
sequence_of_statements -- executes three times
END LOOP;
```

Dynamic Ranges

PL/SQL lets you determine the loop range dynamically at run time, as the following example shows:

```
SELECT COUNT(empno) INTO emp_count FROM emp;
FOR i IN 1..emp_count LOOP
...
END LOOP;
```

The value of emp_count is unknown at compile time; the SELECT statement returns the value at run time.

Using the EXIT Statement

The EXIT statement allows a FOR loop to complete prematurely. For example, the following loop normally executes ten times, but as soon as the FETCH statement fails to return a row, the loop completes no matter how many times it has executed:

```
FOR j IN 1..10 LOOP
FETCH c1 INTO emp_rec;
EXIT WHEN c1%NOTFOUND;
...
END LOOP;
```

Suppose you must exit from a nested FOR loop prematurely. You can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete. Then, use the label in an EXIT statement to specify which
FOR loop to exit, as follows:

```
<<outer>>
FOR i IN 1..5 LOOP
...
FOR j IN 1..10 LOOP
FETCH c1 INTO emp_rec;
EXIT outer WHEN c1%NOTFOUND; -- exit both FOR loops
...
END LOOP;
END LOOP outer;
-- control passes here
```

NULL Statement

The NULL statement explicitly specifies inaction; it does nothing other than pass control to the next statement. It can, however, improve readability. In a construct allowing alternative actions, the NULL statement serves as a placeholder. It tells
readers that the associated alternative has not been overlooked, but that indeed no action is necessary. In the following example, the NULL statement shows that no action is taken for unnamed exceptions:

```
EXCEPTION
WHEN ZERO_DIVIDE THEN
ROLLBACK;
WHEN VALUE_ERROR THEN
INSERT INTO errors VALUES ...
COMMIT;
WHEN OTHERS THEN
NULL;
END;
```

Each clause in an IF statement must contain at least one executable statement. The NULL statement is executable, so you can use it in clauses that correspond to circumstances in which no action is taken. In the following example, the NULL
statement emphasizes that only top-rated employees get bonuses:

```
IF rating > 90 THEN
compute_bonus(emp_id);
ELSE
NULL;
END IF;
```

Also, the NULL statement is a handy way to create stubs when designing applications from the top down. A stub is dummy subprogram that allows you to defer the definition of a procedure or function until you test and debug the main program. In the following example, the NULL statement meets the requirement that at least one statement must appear in the executable part of a subprogram:

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
BEGIN
NULL;
END debit_account;
```

Cursor

A cursor is the Private Memory area which is created by an Oracle server for manipulating the data.

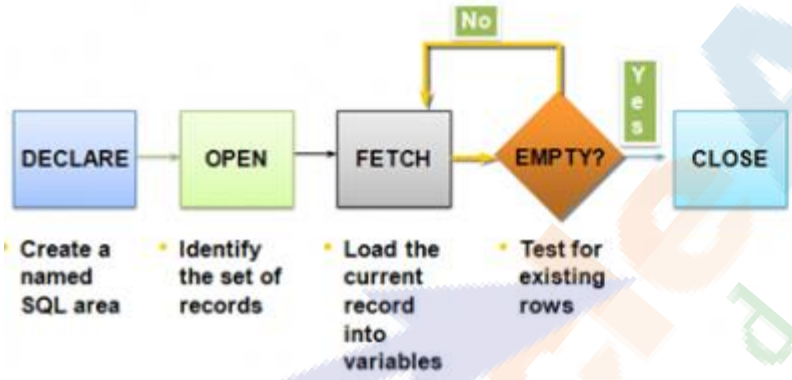
Two Types of CURSORS

1. EXPLICIT : Multiple row SELECT STATEMENTS
2. IMPLICIT

All INSERT statements
All UPDATE statements
All DELETE statements
Single row SELECT….INTO Statements

Using Explicit Cursors

The set of rows returned by a query can consist of zero, one, or multiple rows, depending on how many rows meet your search criteria. When a query returns multiple rows, you can explicitly declare a cursor to process the rows. Moreover,
you can declare a cursor in the declarative part of any PL/SQL block, subprogram, or package. You use three commands to control a cursor: **OPEN, FETCH,** and **CLOSE**. First, you initialize the cursor with the OPEN statement, which identifies the result set. Then, you use the FETCH statement to retrieve the first row. You can execute FETCH repeatedly until all rows have been retrieved. When the last row has been processed, you release the cursor with the CLOSE statement. You can process several queries in parallel by declaring and opening multiple cursors.




```
DECLARE
CURSOR Cur_dept IS
        SELECT   deptno, deptname
        FROM     dept
        WHERE    deptno = 10;

V_ID              Dept.deptno%TYPE;
V_Name Dept.deptname%TYPE;

BEGIN
OPEN Cur_dept ;
LOOP
    FETCH Cur_dept INTO V_ID, V_Name;
    EXIT WHEN Cur_dept %NOTFOUND;
END LOOP;
CLOSE Cur_dept ;
END;
```

Explicit Cursors Attributes

```
%NOTFOUND
LOOP
 FETCH my_cursor INTO my_ename , my_sal ;
 EXIT WHEN my_cursor%NOTFOUND ;
   -- process data here
END LOOP ;

%FOUND
WHILE my_cursor%FOUND
LOOP
-- process data here
FETCH my_cursor INTO my_ename ,my_sal ;
END LOOP ;
```

```
%ROWCOUNT
LOOP
FETCH my_cursor INTO my_ename , my_sal ;
EXIT WHEN (my_cursor%NOTFOUND)
   OR  (my_cursor%ROWCOUNT > 10) ;
--   process data here
END LOOP

%ISOPEN
IF my_cursor%ISOPEN THEN
FETCH my_cursor INTO my_ename , my_sal ;
ELSE
OPEN my_cursor ;
END IF ;
```

Using Cursor FOR Loops

In most situations that require an explicit cursor, you can simplify coding by using a cursor FOR loop instead of the OPEN, FETCH, and CLOSE statements. A cursor FOR loop implicitly declares its loop index as a %ROWTYPE record, opens a cursor,
repeatedly fetches rows of values from the result set into fields in the record, and closes the cursor when all rows have been processed.
Consider the PL/SQL block below, which computes results from an experiment, then stores the results in a temporary table. The FOR loop index c1_rec is implicitly declared as a record. Its fields store all the column values fetched from the cursor c1. Dot notation is used to reference individual fields.

```
DECLARE
result temp.col1%TYPE;
CURSOR c1 IS
SELECT n1, n2, n3 FROM data_table WHERE exper_num = 1;
BEGIN
FOR c1_rec IN c1
LOOP
/* calculate and store the results */
result := c1_rec.n2 / (c1_rec.n1 + c1_rec.n3);
INSERT INTO temp VALUES (result, NULL, NULL);
END LOOP;
COMMIT;
END;
```

Passing Parameters

You can pass parameters to the cursor in a cursor FOR loop. In the following example, you pass a department number. Then, you compute the total wages paid to employees in that department.

Also, you determine how many employees have
salaries higher than $2000 and/or commissions larger than their salaries.
-- available online in file 'examp8'

```
DECLARE
CURSOR emp_cursor(dnum NUMBER) IS
     SELECT sal, comm FROM emp WHERE deptno = dnum;
     total_wages NUMBER(11,2) := 0;
     high_paid NUMBER(4) := 0;
     higher_comm NUMBER(4) := 0;
BEGIN
/* The number of iterations will equal the number of rows returned by
emp_cursor. */
     FOR emp_record IN emp_cursor(20)
     LOOP
          emp_record.comm := NVL(emp_record.comm, 0);
          total_wages := total_wages + emp_record.sal +
          emp_record.comm;
          IF emp_record.sal > 2000.00
          THEN
               high_paid := high_paid + 1;
          END IF;
          IF emp_record.comm > emp_record.sal
          THEN
               higher_comm := higher_comm + 1;
          END IF;
     END LOOP;
     INSERT INTO temp
     VALUES (high_paid, higher_comm, 'Total Wages: ' ||
TO_CHAR(total_wages));
     COMMIT;
END;
```

Implicit Cursors - FOR Loops

An Implicit Cursor is automatically associated with any SQL DML statement that does not have an explicit cursor associated with it.
This includes :

1. ALL INSERT statements
2. ALL UPDATE statements
3. ALL DELETE statements
4. ALL SELECT…INTO statements

QuickNotes - Implicit Cursors

1. Implicit cursor is called the "SQL" cursor --it stores information concerning the processing of the last SQL statement not associated with an explicit cursor.
2.OPEN, FETCH, AND CLOSE don't apply.
3. All cursor attributes apply.

Implicit Cursors Attributes

```
%NOTFOUND
UPDATE emp SET sal = sal * 10.0
WHERE ename ="WARD" ;
IF SQL %NOTFOUND THEN
  -- WARD wasn't found
  INSERT INTO emp (empno, ename ,sal)
  VALUES ( 1234,'WARD' 99999 );
END IF ;
```

```
%ROWCOUNT
DELETE FROM baseball_team
WHERE batting _avg. < .100;

IF SQL%ROWCOUNT > 5 THEN
  INSERT INTO temp(message)
  VALUES('Your team needs helps.');
END IF;
```

FOR UPDATE Clause

- Use explicit locking to deny access for the duration of a transaction
- Locks the rows before update or delete .

Syntax: Select …..
from
FOR UPDATE [OF column reference] [NOWAIT];

e.g.

```
Declare
  Cursor EmpCursor is
  select emp_id, last_name, dept_name
  from employees , department
  where employees.dept_id=department.dept_id
  and employees.dept_id=80
  FOR UPDATE OF salary NOWAIT;
```

Exception Handling

In PL/SQL, a warning or error condition is called an exception. Exceptions can be internally defined (by the run-time system) or user defined. Examples of internally defined exceptions include division by zero and out of memory. Some common internal exceptions have predefined names, such as ZERO_DIVIDE and STORAGE_ERROR. The other internal exceptions can be given names.

- In PL/SQL error are called exceptions
- When an exception is raised, processing jumps to the exception handlers
- Two Types of Exceptions 1. PREDEFINED INTERNAL EXCEPTIONS 2. USER-DEFINED EXCEPTIONS
- An exception handler is a sequence of statements to be processed when a certain exception occurs
- When an exception handler is complete processing of the block terminates

Examples of Predefined internal exceptions

TOO_MANY_ROWS ORA-(01427)
- a single row SELECT returned more than one row

NO_DATA_FOUND ORA-(01403)
- a single row SELECT returned no data

INVALID_CURSOR ORA-(01001)
- invalid cursor was specified

VALUE_ERROR ORA-(06502)
- arithmetic ,numeric, string , conversion, or constraint error occurred.

ZERO_DIVIDE ORA-(01476)
- attempted to divide by zero

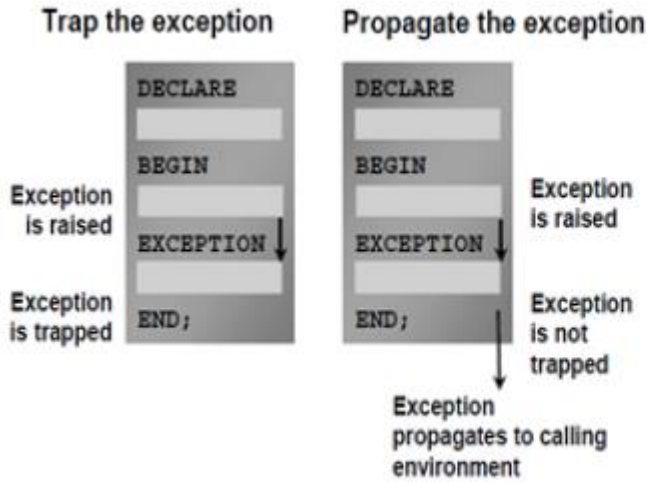
DUP_VAL_ON_INDEX ORA-(00001)
-attempted to insert a duplicate value into a column that has a unique index specified.

Exception Types

There are three types of exception

Exception	Description	Directions for Handling
Predefined Oracle Server error	One of approximately 20 errors that occur most often in PL/SQL code	Do not declare and allow the Oracle server to raise them implicitly
Nonpredefined Oracle Server error	Any other standard Oracle Server error	Declare within the declarative section and allow the Oracle Server to raise them implicitly
User-defined error	A condition that the developer determines is abnormal	Declare within the declarative section, *and* raise explicitly

Exception Handlers




```
DECLARE
   employee_num   emp.empno%TYPE;
BEGIN
   SELECT empno INTO employee_num FROM emp;
   WHERE ename = 'BLAKE';
   INSERT INTO temp VALUES(NULL, empno,
                           Blake's employee_num');
   DELETE FROM emp  WHERE ename ='BLAKE';
EXCEPTION
  WHEN TOO_MANY_ROWS OR NO_DATA_FOUND THEN
     ROLLBACK;
     INSERT INTO temp VALUES
     (NULL,'Blake not found, or more than one Blake');
     COMMIT;
    WHEN OTHERS THEN
     ROLLBACK;
END;
```

Trapping an Exception:

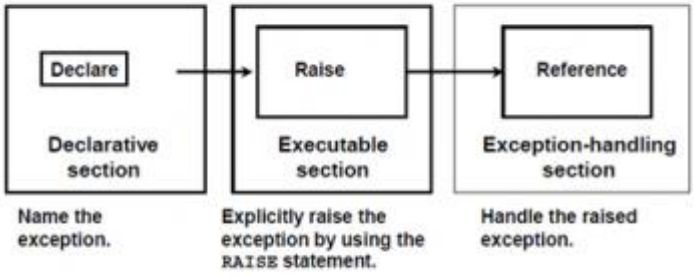
If the exception is raised in the executable section of the block, processing branches to the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, then the exception does not propagate to the enclosing block or environment. The PL/SQL block terminates successfully.

Propagating an Exception:

If the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with failure and the exception is propagated to the calling environment.

User-Defined Exceptions

PL/SQL lets you define exceptions of your own. Unlike predefined exceptions, user-defined exceptions must be **declared** and must be raised explicitly by **RAISE** statements.




```
DECLARE
   my_ename    emp.ename%TYPE :='BLAKE';
   assigned_projects    NUMBER;
   too_few_projects  EXCEPTION ;
BEGIN  ---- get no of projects assigned to BLAKE
   IF assigned_project < 3 THEN
     RAISE  too_few_projects;
   END IF;
EXCEPTION    --begin the exception handlers
WHEN too_few_projects THEN
       INSERT INTO temp
       VALUES(my_ename, assigned_projects ,'
            LESS THAN 3 PROJECTS!')
       COMMIT;
END;
```

Predefined Exceptions

An internal exception is raised implicitly whenever your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. So, PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception NO_DATA_FOUND if a SELECT INTO statement returns no rows.

To handle other Oracle errors, you can use the OTHERS handler. The functions SQLCODE and SQLERRM are especially useful in the OTHERS handler because they return the Oracle error code and message text. Alternatively, you can use the pragma EXCEPTION_INIT to associate exception names with Oracle error codes.

PL/SQL declares predefined exceptions globally in package STANDARD, which defines the PL/SQL environment. So, you need not declare them yourself. You can write handlers for predefined exceptions using the names shown in the list below. Also shown are the corresponding Oracle error codes and SQLCODE return values.

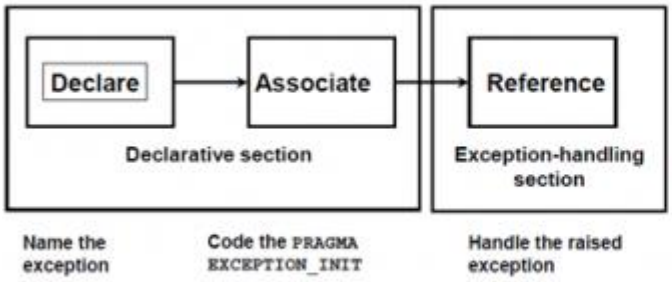
NonPredefined Error

To handle unnamed internal exceptions, you must use the OTHERS handler or the pragma EXCEPTION_INIT. A pragma is a compiler directive, which can be thought of as a parenthetical remark to the compiler. Pragmas (also called pseudoinstructions)
are processed at compile time, not at run time.

In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That allows you to refer to any internal exception by name and to write a specific handler for it. You code the pragma EXCEPTION_INIT in the declarative part of a PL/SQL block, subprogram, or package using the syntax

PRAGMA EXCEPTION_INIT(exception_name, Oracle_error_number);

where exception_name is the name of a previously declared exception. The pragma must appear somewhere after the exception declaration in the same declarative section, as shown in the following example:




```
DEFINE p_deptno = 10
DECLARE
  e_emps_remaining EXCEPTION;                    1
  PRAGMA EXCEPTION_INIT
    (e_emps_remaining, -2292);                   2
BEGIN
  DELETE FROM departments
  WHERE  department_id = &p_deptno;
  COMMIT;
EXCEPTION
  WHEN e_emps_remaining  THEN                    3
   DBMS_OUTPUT.PUT_LINE ('Cannot remove dept ' ||
   TO_CHAR(&p_deptno) || '.  Employees exist. ');
END;
```

Error Reporting Functions

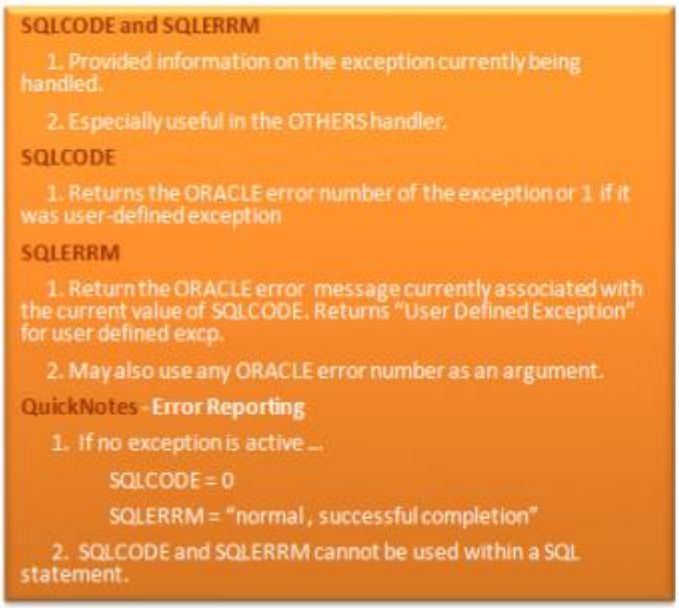
In an exception handler, you can use the built-in functions **SQLCODE** and **SQLERRM** to find out which error occurred and to get the associated error message.

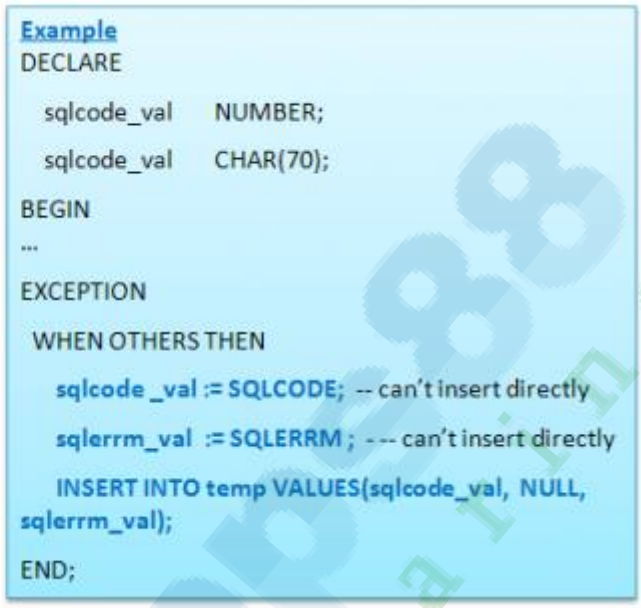
1. For internal exceptions, SQLCODE returns the number of the Oracle error. The number that SQLCODE returns is negative unless the Oracle error is no data found, in which case SQLCODE returns +100. SQLERRM returns the corresponding error message. The message begins with the Oracle error code.

2. For user-defined exceptions, SQLCODE returns +1 and SQLERRM returns the message User-Defined Exception unless you used the pragma EXCEPTION_INIT to associate the exception name with an Oracle error number, in which case SQLCODE returns that error number and SQLERRM returns the corresponding error message.

The maximum length of an Oracle error message is 512 characters including the error code, nested messages, and message inserts such as table and column names.

If no exception has been raised, SQLCODE returns zero and SQLERRM returns the message ORA-0000: normal, successful completion.





Passing an error number

You can pass an error number to SQLERRM, in which case SQLERRM returns the message associated with that error number. Make sure you pass negative error numbers to SQLERRM. In the following example, you pass positive numbers and so get unwanted results:

```
DECLARE
...
err_msg VARCHAR2(100);
BEGIN
/* Get all Oracle error messages. */
FOR err_num IN 1..9999 LOOP
err_msg := SQLERRM(err_num); -- wrong; should be -err_num
INSERT INTO errors VALUES (err_msg);
END LOOP;
END;
```

Passing a positive number to SQLERRM always returns the message user-defined exception unless you pass +100, in which case SQLERRM returns the message no data found. Passing a zero to SQLERRM always returns the message normal, successful completion.

SQLCODE or SQLERRM in SQL statements

You cannot use SQLCODE or SQLERRM directly in a SQL statement. Instead, you must assign their values to local variables, then use the variables in the SQL statement, as shown in the following example:

```
DECLARE
err_num NUMBER;
err_msg VARCHAR2(100);
BEGIN
```

```
...
EXCEPTION
...
WHEN OTHERS THEN
err_num := SQLCODE;
err_msg := SUBSTR(SQLERRM, 1, 100);
INSERT INTO errors VALUES (err_num, err_msg);
END;
```

The string function SUBSTR ensures that a VALUE_ERROR exception (for truncation) is not raised when you assign the value of SQLERRM to err_msg. The functions SQLCODE and SQLERRM are especially useful in the OTHERS exception handler because they tell you which internal exception was raised.

Exceptions Propagation

Step# 1 The current block is searched for a handler. If not found, go to step 2.

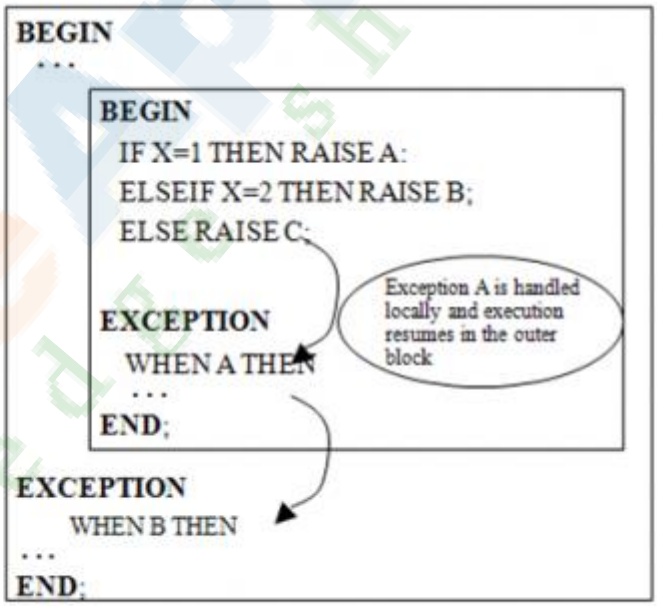
Step# 2 If an enclosing block is found, it is searched for it handler.

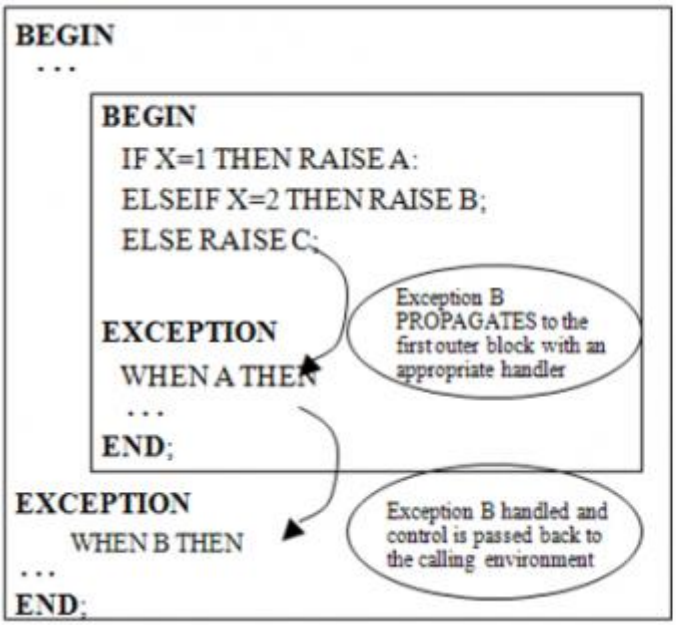
Step# 3 Step #1 and#2 are repeated until either there are no more enclosing blocks, or a handler is found .

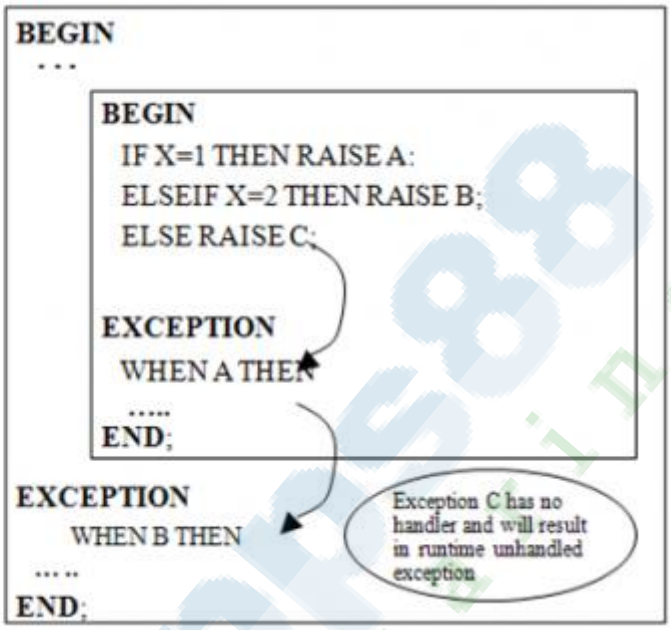
- If there are no more enclosing blocks, the exception is passed back to the calling environment (SQL *Plus,SQL *Forms, a precompiled program,etc.)
- If the handler is found ,it is executed .when done the block in which the handler was found is terminated, and control is passed to thee enclosing block (if one exists), or to environment (if there is no enclosing block)

Quick notes

1. Only one handler per block may be active at a time
2. If an exception is raised in a handler, the search for a handler for the new exception begins in the enclosing block of the current block



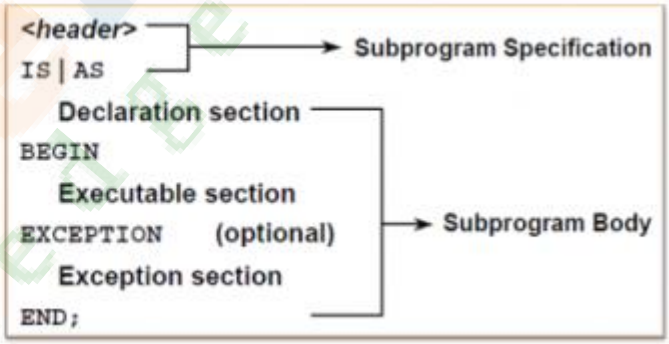




Subprograms

Subprograms are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two types of subprograms called procedures and functions. Generally, you use a procedure to perform an action and a function to compute a value.

- Collections of SQL and PL/SQL statements
- Stored in complied from in the database
- Can call other procedures
- Can be called from all client environments
- Procedures and function (including remote) are the same, except a function returns a values and a procedure does not.



```
PROCEDURE PROC_SCHEDULE_CRUISE
 ( p_start_date IN DATE    DEFAULT SYSDATE
 , p_total_days IN NUMBER
 , p_ship_id    IN NUMBER
 , p_cruise_name IN VARCHAR2 DEFAULT 'Island Getaway')
IS
v_cruise_type_id CRUISE_TYPES.CRUISE_TYPE_ID%TYPE;

BEGIN
SELECT CRUISE_TYPE_ID
 INTO  v_cruise_type_id
 FROM  CRUISE_TYPES
 WHERE LENGTH_DAYS = p_total_days;
 -- Schedule cruise
 INSERT INTO CRUISES
  ( CRUISE_ID
, END_DATE)
 VALUES
 ( SEQ_CRUISE_ID.NEXTVAL
 , (p_start_date + p_total_days));
 COMMIT;

EXCEPTION
 WHEN OTHERS THEN
 ROLLBACK;
 PROC_RECORD_ERROR('PROC_SCHEDULE_CRUISE');

END;
```

1 2 3

```
FUNCTION FUNC_GET_CLASSIFICATION
(p_duration NUMBER)
RETURN VARCHAR2
IS

BEGIN
IF (p_duration <= 4) THEN
RETURN 'Weekend Getaway';
ELSIF (p_duration > 4) AND (p_duration <= 7)
THEN
RETURN 'Weeklong Adventure';
ELSE
END IF;

EXCEPTION
 WHEN OTHERS THEN
 ROLLBACK;
 Statements;

END;
```

Uses of Procedures/Functions

Procedures are excellent for defining a PL/SQL code block that you know you will need to call more than once, and whose work may produce results largely seen in the database or perhaps some module, like an Oracle Form, or a client-side form, as opposed to work whose result is some single answer; that would probably be more appropriate for a function.

In addition, an anonymous PL/SQL block is parsed each time it is submitted for execution. But if that same anonymous block is assigned a name and created as a procedure, then Oracle will parse the procedure once, at the time it is created. Each subsequent call to that procedure will not require reparsing; it will simply execute, saving time over an anonymous block.

A PL/SQL procedure can be invoked from a single executable statement in another PL/SQL statement. These other PL/SQL statements could be in an anonymous PL/SQL block or in a named program unit, such as another procedure. A PL/SQL procedure can also be invoked from a single command-line executable statement in a SQL*Plus session.

A **function**'s main purpose is to return a single value of some sort, as opposed to a procedure, whose main purpose is to perform some particular business process. Like a procedure, a function is a PL/SQL block that's been assigned a name; but unlike a procedure, the function will always return one—and only one—value of some sort. This returned value is embodied in the function call in such a way that the function becomes, in essence, a variable.
When you create a function, you must consider how you intend to use the function.

There are two major categories of functions you can create:

- Functions that are called from expressions in other PL/SQL program units. Any function can be used this way.
- Functions that are called from within SQL statements, whether the SQL statement is part of a PL/SQL program unit or not. Some functions you create in PL/SQL can be used in this way.

It's possible to create functions that can be invoked in both manners. However, if you intend to make a function that can be called from a valid SQL statement, there are some restrictions you have to consider. For example, a function that returns a BOOLEAN datatype, which is perfectly acceptable in PL/SQL, cannot be invoked from a SQL statement, where BOOLEAN datatypes are not recognized.

Functions versus Procedures

Functions can be used in places where procedures cannot be used. Whereas a procedure call is a statement unto itself, a call to a function is not; a function call is part of an expression. This means that functions can be used as a part, or all, of the right side of the assignment statement. Functions can be used as part, or perhaps all, of the Boolean expression in an IF statement. In short, wherever you might use a variable, you can use a function.

Functions always return a single value, embodied in the function call itself. In other words, contrary to the optional OUT parameter feature in procedures, which you may or may not use to return multiple values from a procedure, a function must always return one—and only one—value through the very call to the function itself. This value is not returned in the form of an OUT parameter, but instead it is returned in the body of the function itself, so that the function call behaves like a variable. Technically, functions can use IN, OUT, and IN OUT parameters. In practice, functions are generally only given IN parameters.

Where Can You Store Procedures?

Procedures can be stored in the database, alongside tables and other database objects. Once a procedure is stored in the database, it can be invoked from any process with database access. If a process, such as a SQL*Plus window, Java program, Oracle Form, or another PL/SQL procedure, has access to the database, it can execute the procedure, provided that the proper privileges have been granted on the procedure (more on this later) to the schema under which the process is running.

Create, Alter and Drop

Creating Procedures

The following is a code sample that will create a stored procedure named PROC_RESET_ERROR_LOG:

```
CREATE PROCEDURE PROC_RESET_ERROR_LOG IS
BEGIN
  -- Clean out the ERRORS table
```

```
  DELETE FROM ERRORS;
  COMMIT;
END;
```

The syntax to create a function is similar to the syntax used to create a procedure, with one addition: the RETURN declaration. The following is a sample CREATE FUNCTION statement.

```
CREATE OR REPLACE FUNCTION FUNC_COUNT_GUESTS
  (p_cruise_id NUMBER)
RETURN NUMBER
IS
  v_count NUMBER(10)
BEGIN
  SELECT COUNT(G.GUEST_ID)
  INTO   v_count
FROM     GUESTS G,
        GUEST_BOOKINGS GB
WHERE    G.GUEST_ID = GB.GUEST_BOOKING_ID
  AND    GB.CRUISE_ID = p_cruise_id;
RETURN   v_count;
END;
```

This function will take a single parameter, p_cruise_id. This parameter could include the parameter type declaration, such as IN, OUT, or IN OUT, but this example leaves it out, so this parameter is assumed to be the default IN parameter type, just as it would be assumed in a procedure. This function will use the p_cruise_id parameter to query the database and count the total number of guests for a single cruise. The result of the query is then returned to the calling block, using the RETURN statement at the end of the function.

If you think of the entire function as a variable, then think of the **RETURN** datatype as the function's **datatype**.

Altering Procedures

Once a procedure has been created, you can use two methods to "alter" the procedure. If you are replacing the original source code with a new set of source code, use the **OR REPLACE** option discussed in the previous section. This is true for any code modification at all. If, however, you are recompiling the procedure without changing the code, then use the **ALTER PROCEDURE** command.

The ALTER PROCEDURE command is required when your stored procedure has not been changed in and of itself, but another database object referenced from within your procedure, such as a table, has been changed. This automatically causes your procedure to be flagged as INVALID.

```
CREATE OR REPLACE PROCEDURE PROC_RESET_ERROR_LOG IS
BEGIN
  -- Clean out the ERRORS table
```

```
  DELETE FROM ERRORS;
  COMMIT;
END;
```

ALTER PROCEDURE PROC_RESET_ERROR_LOG COMPILE;

As with a procedure, a function may reference database objects from within its code. As with a procedure, if those database objects are changed, then the function must be recompiled. To perform this recompilation, use the **ALTER FUNCTION ...COMPILE** command.

```
ALTER FUNCTION FUNC_COUNT_GUESTS COMPILE;
CREATE OR REPLACE FUNCTION FUNC_COUNT_GUESTS
  (p_cruise_id NUMBER)
RETURN NUMBER
IS
  v_count NUMBER(10)
BEGIN
  Statements;
END;
```

Dropping Procedures

An example of a command that drops a procedure is shown in the following code listing:

```
DROP PROCEDURE PROC_RESET_ERROR_LOG;
```

Once this command is successfully executed, the database response "Procedure dropped" will be displayed.

To drop a function, use the DROP … FUNCTION statement. The following is a sample command that will drop our sample function:

```
DROP FUNCTION FUNC_COUNT_GUESTS;
```

Invoking Procedures/Functions

Once a procedure has been created and stored in the database, it can be invoked from

- An executable statement of a PL/SQL block
- A command entered in the SQL*Plus command-line interface

Executing a Procedure from a PL/SQL Block

To invoke a procedure from another PL/SQL block, use a single statement that names the procedure. For example, suppose you've created a procedure called PROC_UPDATE_CRUISE_STATUS. The following PL/SQL block will execute the procedure:

```
BEGIN
  PROC_UPDATE_CRUISE_STATUS;
END;
```

Executing a Procedure from the SQL*Plus Command Line

You can execute a PL/SQL procedure from within the SQL*Plus command line without having

to write another PL/SQL block to do it. The SQL command EXECUTE, or EXEC for short, must be used.
For example, if you have already stored a procedure called PROC_RUN_BATCH with no parameters, then the following statement, entered in the SQL*Plus window at the SQL prompt, will invoke the procedure:

```
EXECUTE PROC_RUN_BATCH;
```

Invoking Functions

Functions are never called in a stand-alone statement as procedures are. Instead, a function call is always part of some other expression. Valid PL/SQL expressions can incorporate functions anywhere that a variable would be accepted. Valid SQL expressions may also invoke functions, but with a few limitations—only certain types of functions can be invoked from SQL.
The following is a sample of a block that might call our sample FUNC_COUNT_GUESTS function:

```
PROCEDURE PROC_ORDER_FOOD (p_cruise_number NUMBER)
IS
  v_guest_count NUMBER(10);
BEGIN
  -- Get the total number of guests
-- for the given cruise
v_guest_count := FUNC_COUNT_GUESTS(p_cruise_number);
-- Issue a purchase order
  INSERT INTO PURCHASE_ORDERS
    (PURCHASE_ORDER_ID, SUPPLIER_ID, PRODUCT_ID, QUANTITY)
    VALUES
    (SEQ_PURCHASE_ORDER_ID.NEXTVAL, 524, 1, v_guest_count)
  COMMIT;
END;
```

Functions Called from PL/SQL Expressions

Any PL/SQL function can be called from a PL/SQL expression of another program unit. Remember that expressions can be found in many places within PL/SQL:

- The right side of an assignment statement
- The Boolean expression of an IF … THEN … END IF statement
- The Boolean expression of a WHILE loop
- The calculation of a variable's default value

In short, anywhere you might use a PL/SQL variable, you can issue a function call.
Examples:

```
1. DECLARE
  v_official_statement VARCHAR2(1000);
BEGIN
  v_official_statement := 'The leading customer is ' ||
```

```
                leading_customer;
END;
```

2. Functions can even be used as parameter values to other functions. For example,

```
BEGIN
  IF (leading_customer(get_largest_department) = 'Iglesias')
  THEN
    DBMS_OUTPUT.PUT_LINE('Found the leading customer');
  END IF;
END;
```

3.

```
SELECT     COUNT(SHIP_ID) NUMBER_OF_SHIPS,
          leading_customer
FROM       SHIPS;
```

Parameters

Submitted by Anonymous on Sat, 04/11/2009 - 16:00

A parameter is a variable whose value can be defined at execution time and can be exchanged between the procedure and the calling PL/SQL block. Parameter values can be passed in to the procedure from the calling PL/SQL block and can optionally have their values passed back out of the procedure to the calling PL/SQL block upon the completion of the procedure's execution Parameters are declared at the top of the procedure within a set of parentheses. Each parameter declaration includes the following:

- A name, defined by the developer, and adhering to the rules of object names (discussed earlier).
- The type of parameter, which will either be IN, OUT, or IN OUT. The default is IN.
- The datatype. Note that no specification or precision is allowed in parameter datatype declarations. To declare something as an alphanumeric string, you can use VARCHAR2, but you cannot use, for example, VARCHAR2(30).
- Optionally, a parameter may be provided with a default value. This can be done by using the reserved word DEFAULT, followed by a value or expression that is consistent with the declared datatype for the parameter. The DEFAULT value identifies the value the parameter will have if the calling PL/SQL block doesn't assign a value.

After each parameter declaration, you may place a comma and follow it with another parameter declaration.

The following is an example of a procedure header that uses parameters:

```
PROCEDURE PROC_SCHEDULE_CRUISE
  ( p_start_date IN DATE DEFAULT SYSDATE
  , p_total_days IN NUMBER
  , p_ship_id IN NUMBER
  , p_cruise_name IN VARCHAR2 DEFAULT 'Island Getaway')
IS
... code follows ...
```

This procedure declares four parameters. Each parameter is an IN parameter. Each parameter is

assigned a datatype. The parameter p_cruise_name is given a datatype of VARCHAR2; the length cannot be specified in a parameter datatype declaration.
Two of the parameters are assigned default values. The first, p_start_date, uses the Oracle pseudocolumn SYSDATE, and the second, p_cruise_name, is assigned the string, 'Island Getaway'.

Functions parameters

Functions take parameters, just like procedures do, and just like procedures, a parameter for a function can be an IN, OUT, or an IN OUT parameter. The default parameter type is an IN parameter.
However, unlike a procedure, a function always returns a value through its unique RETURN statement, and this value replaces the original call to the function in the expression that calls the function. Given this, functions are not generally used to pass OUT or IN OUT parameters. Furthermore, the OUT and IN OUT parameter will not work with function calls that are made from SQL statements. For example, consider the following function:

```
FUNCTION FUNC_COMPUTE_TAX
(p_order_amount IN OUT NUMBER)
RETURN NUMBER
IS
BEGIN
  p_order_amount := p_order_amount * 1.05;
  RETURN p_order_amount * .05;
END;
```

This function has an IN OUT parameter. The parameter comes IN as some dollar amount representing an order; it goes OUT with tax added. The function RETURNS the amount of the tax itself, as a NUMBER datatype

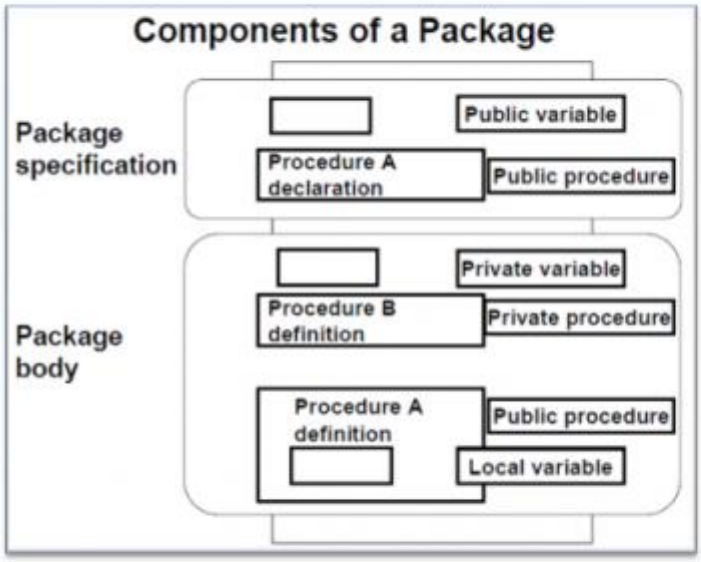
Return
Submitted by Anonymous on Sat, 04/11/2009 - 15:57
The use of the RETURN statement is unique to functions. The RETURN statement is used to return some value. In fact, the primary reason for storing a PL/SQL block as a function is to return this value—this is the purpose of the function. For example, if a function is meant to compute the total payments received so far from guest reservations booked on a cruise, then the function will do whatever it needs to do to arrive at this final value and use the RETURN statement at the end to send the result back to the function call.

If you attempt to compile a function that has no RETURN statement, you will succeed, and the function will be stored in the data dictionary with a status of VALID. However, when you attempt to execute the function, you will receive a message like this:
ORA-06503: PL/SQL: Function returned without value
ORA-06512: at "[schema.function_name]", line 6
ORA-06512: at line 1

Therefore, it is the developer's responsibility to remember the RETURN statement. The compilation process won't remind you that it's required.
The function processes its statements until the RETURN statement is reached. Once the RETURN statement is processed, the execution of the function will stop. Any statements that follow will be ignored, and control is returned to the calling source. Therefore, it is considered good design to make the RETURN statement the last executable statement. However, the parser does not require this. Your function will compile without any RETURN statement or with a RETURN statement that precedes other valid PL/SQL statements.

Packages

A PL/SQL package is a single program unit that contains one or more procedures and/or functions, as well as various other PL/SQL constructs such as cursors, variables, constants, and exceptions. Packages bring these various constructs together in a single program unit.
Most applications include several Procedural Language/Structured Query Language (PL/SQL) procedures and functions that are logically related together. These various procedures and functions could be left as stand-alone individual procedures and functions, stored in the database. However, they can also be collected in a package, where they can be more easily organized and where you will find certain performance improvements as well as access control and various other benefits.
A package is a collection of PL/SQL program constructs, including variables, constants, cursors, user-defined exceptions, and PL/SQL procedures and functions, as well as PL/SQL-declared types. A package groups all of these constructs under one name. More than that, the package owns these constructs, and in so doing, affords them powers and performance benefits that would not otherwise exist if these constructs were not packaged together.




```
CREATE OR REPLACE PACKAGE emp_actions AS -- spec
TYPE EmpRecTyp IS RECORD (emp_id INTEGER, salary REAL);
CURSOR desc_salary RETURN EmpRecTyp;
PROCEDURE hire_employee (
ename VARCHAR2,
job VARCHAR2,
mgr NUMBER,
sal NUMBER,
comm NUMBER,
deptno NUMBER);
PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

CREATE OR REPLACE PACKAGE BODY emp_actions AS -- body
CURSOR desc_salary RETURN EmpRecTyp IS
SELECT empno, sal FROM emp ORDER BY sal DESC;
PROCEDURE hire_employee (
ename VARCHAR2,
job VARCHAR2,
mgr NUMBER,
sal NUMBER,
comm NUMBER,
deptno NUMBER)
IS
BEGIN
INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job,
mgr, SYSDATE, sal, comm, deptno);
END hire_employee;
PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
DELETE FROM emp WHERE empno = emp_id;
END fire_employee;
END emp_actions;
```

Create, Alter and Drop - Packages

The statements used to create, alter, and drop packages are rather straightforward. However, this process is a little more involved than merely creating a procedure or function. The first point to understand is that a package consists of two parts: the package specification and the package body. The two parts are created separately. Any package must have a specification.

A package may optionally include a package body but is not necessarily required to. The requirement for a package to have a body will be determined by what you declare in a package specification; you may simply declare a package specification and no body. However, most packages often include both the package specification and the package body.

You can declare a package specification without the package body and successfully store it in the database as a valid object. Furthermore, with only a package specification, it's possible to create other PL/SQL programs that call on the constructs of your package, even procedures or functions, whose code isn't written yet—the actual code can only be defined in the package body. However, the existence of the package specification enables other outside PL/SQL program units to reference the constructs of this package.

It's recommended that you create the package specification first, before the package body. The package specification, as we have seen, will successfully store, compile, and support the

successful compilation of outside program units. A package body, on the other hand, cannot be compiled successfully without a corresponding package specification. However, the package body can be submitted and stored in the data dictionary without a package specification. The package body will simply be flagged with a status of INVALID in the USER_OBJECTS data dictionary view. After the package specification is successfully created, you need to either issue the ALTER PACKAGE … COMPILE statement, or simply reference a packaged construct and let the database automatically compile the package at that time.

Creating a Package Specification

The following is an example of a statement that creates a stored PL/SQL package specification:

```
CREATE OR REPLACE PACKAGE pack_booking AS
  c_tax_rate NUMBER(3,2) := 0.05;
  CURSOR cur_cruises IS
    SELECT CRUISE_ID, CRUISE_NAME
    FROM   CRUISES;
  rec_cruises cur_cruises%ROWTYPE;
  FUNCTION func_get_start_date
    (p_cruise_id IN CRUISES.CRUISE_ID%TYPE)
  RETURN DATE;
END pack_booking;
```

The syntax of the statement is the reserved word CREATE, followed by the optional OR REPLACE words, the reserved word PACKAGE, the name you choose for the package, and the reserved word AS. (The word IS is also accepted here.) Next is a series of declared constructs. Eventually, the closing END is included, followed by the package name, optionally repeated for clarity, and then the semicolon.

This package specification declares a constant c_tax_rate, a cursor cur_cruises, a record variable rec_cruises, and the function func_get_start_date. Notice that the function's actual code isn't included here, only the function header.

The package specification is the part of a package that declares the constructs of the package. These declared constructs may include any of the following:

- Variables and/or constants
- Compound datatypes, such as PL/SQL tables and TYPEs
- Cursors
- Exceptions
- Procedure headers and function headers
- Comments

The package specification contains no other code. In other words, the actual source code of procedures and functions is never included in the package specification. The specification merely includes enough information to enable anyone who wants to use these constructs to understand their names, parameters, and their datatypes, and in the case of functions, their return datatypes, so that developers who want to write calls to these constructs may do so. In other words, if any

developer wants to create new programs that invoke your package constructs, all the developer needs to see is the package specification. That's enough information to create programs that employ the procedures and functions of your package. The developer does not need to have full access to the source code itself, provided that he or she understands the intent of the program unit.

Once the package specification has been successfully stored in the database, it will be given a status of **VALID**, even if the associated package body, containing the actual source code of any and all functions and/or procedures, has yet to be stored in the database.

Creating a Package Body

The following is a sample of a package body that correlates to the package specification we saw earlier:

```
CREATE OR REPLACE PACKAGE BODY pack_booking AS
  FUNCTION func_get_start_date
    (p_cruise_id IN CRUISES.CRUISE_ID%TYPE)
  RETURN DATE
  IS
     v_start_date CRUISES.START_DATE%TYPE;
  BEGIN
     SELECT START_DATE
     INTO   v_start_date
     FROM   CRUISES
     WHERE  CRUISE_ID = p_cruise_id;
     RETURN v_start_date;
  END func_get_start_date;
END pack_booking;
```

This package body defines the source code of the function func_get_start_date. Notice that the function header is completely represented here, even though it was already declared completely in the package specification. Also, notice that there are no provisions for the other declared constructs in the package specification. Only the functions and/or procedures that were declared in the package specification need to be defined in the package body. The package specification handles the full declaration of the other public constructs, such as variables, constants, cursors, types, and exceptions.

The package body is only required if the package specification declares any procedures and/or functions, or in some cases of declared cursors, depending on the cursor syntax that is used. The package body contains the complete source code of those declared procedures and/or functions, including the headers that were declared in the specification.

The package body can also include privately defined procedures and/or functions. These are program units that are recognized and callable only from within the package itself and are not

callable from outside the package. They are not declared in the package specification, but are defined in the package body.

Altering a Package

Packages, like procedures and functions, should be recompiled with the ALTER command if their referenced constructs are changed for any reason. This includes any referenced database objects, such as tables, views, snapshots, synonyms, and other PL/SQL packages, procedures, and functions.
The syntax to recompile a package with the ALTER statement is

```
ALTER PACKAGE package_name COMPILE;
```

This statement will attempt to recompile the package specification and the package body.
The syntax to recompile just the package body is

```
ALTER PACKAGE package_name COMPILE BODY;
```

Note that the package is listed in the data dictionary with two records: one record for the PACKAGE and another for the PACKAGE BODY. Both have their individual status assignments. The PACKAGE, meaning the package specification, can be VALID, while the PACKAGE BODY is INVALID. If this is the case, then an ALTER PACKAGE package_name COMPILE statement will attempt to restore the entire package, including the body, to a status of VALID.

If a change is made to the package body and it is recompiled, then the package specification does not demand that the package be recompiled. Even if the recreated package body results in a change that is inconsistent with the package specification, the package specification will still show a status of VALID in the data dictionary (assuming it was VALID to begin with), and the package body will be flagged with a status of INVALID

Dropping a Package

You have two options when dropping a package. The following statement will remove the package body reservations from the database:

```
DROP PACKAGE BODY RESERVATIONS;
```

This statement will remove the package body, but will leave the package specification in the database. Furthermore, the package specification for reservations will still be VALID.
The following statement will remove the entire package:

```
DROP PACKAGE RESERVATIONS;
```

The result of issuing this statement to the database will be the complete removal of both the package specification and the package body from the database.
Dropping a package will cause any other program units that reference the package to be flagged with a status of INVALI

Public versus Private Constructs

Constructs that are declared in the package specification are considered public constructs. However, a package body can also include constructs that aren't declared in the package specification. These are considered private constructs, which can be referenced from anywhere within its own package body but cannot be called from anywhere outside the particular package body. Furthermore, any developers with privileges to use the constructs of the package do not necessarily have to see the package body, which means that they do not necessarily know of the existence of the private constructs contained within the package body.

Global Constructs

Package constructs, such as variables, constants, cursors, types, and user-defined exceptions, are global to the user session that references them. Note that this dynamic is irrelevant for packaged procedures and packaged functions, but applies to all other packaged constructs. This is true for both public and private constructs in the package. In other words, the values for these constructs will be retained across multiple invocations within the user session.

For example, if an anonymous PL/SQL block references a public packaged variable and changes its value, the changed value can be identified by another PL/SQL block that executes afterwards. Neither block declares the variable because it's declared as part of the package.

The user cannot directly access any private constructs, such as a variable, but imagine that a user invokes a packaged procedure, for example, that references its own private variable value and changes that value. If the user re-invokes that packaged procedure again within the same user session, the changed value will be recognized by the packaged procedure.

The value will be retained as long as the user session is still active. As soon as the user session terminates, the modified states of the packaged constructs are released, and the next time the user session starts up, the packaged constructs will be restored to their original state, until the user sessions modifies them again.

Invoking Packaged Constructs

Packages themselves are never directly invoked or executed. Instead, the constructs contained within the package are invoked. For example, procedures and functions within the package are executed. Other constructs in the package, such as constants and other declared constructs, can be referenced from within other PL/SQL programs, in the same manner that those program units could refer to their own locally declared PL/SQL program constructs.

Referencing Packaged Constructs

To call upon any construct of a package, simply use the package name as a prefix to the construct name. For example, to reference a procedure with no parameters called increase_wages that is stored in a package called hr, use this reference:

```
hr.increase_wages;
```

All packaged constructs are invoked in the same fashion: by including the package name as a prefix, with a period separating the package name and the construct name. This notation is often referred to as dot notation. It's the same format used to refer to database objects that are owned by a schema.

For example, if you have a package called assumptions that defines a public constant called tax_rate, you could use it this way in an expression:
v_full_price := v_pre_tax * (1 + assumptions.tax_rate);

Dot notation is required for references to any packaged constructs from outside of the package. However, although references to constructs from within the same package will accept dot notation, they do not require it.

Using Packaged Constructs

For the purpose of reviewing the rules of using packaged constructs, let's consider all constructs to be in either of two categories: packaged program units, meaning procedures and functions, and global constructs, meaning packaged variables, constants, cursors, exceptions, and types.

Packaged Program Units

The same rules apply to the use of packaged procedures and functions that apply to stand-alone procedures and functions. In other words, packaged procedures must be called from a single statement, whereas packaged functions must be called from within an expression, just like stand-alone PL/SQL stored functions. Parameters may be passed by positional notation or by named notation.
For example, the following PL/SQL block executes a packaged procedure:

```
BEGIN
  pack_finance.proc_reconcile_invoice(p_status => 'ACTIVE');
END;
```

Packaged Global Constructs

Packaged global constructs generally behave the same as their locally defined counterparts, with one dramatic difference: Their declaration is global to the user sessions.
Consider the following code sample:

```
PACKAGE rates AS
  bonus NUMBER(3,2);
END rates;
```