PL/SQL Programming Language

The intended use of this help manual is as a quick reference guide as it is not fully inclusive of all elements of the PL/SQL Programming Language.   Please refer to the PL/SQL User's Guide and Reference for more information.

Syntax Notation

The following notation is used in code fragments and program examples:

< >	Angle brackets enclose the name of a syntactic element.
--	Two hyphens begin a single-line comment, which extends to the end of a line.
/*	A slash-asterisk begins a multiline comment, which can extend from one line to another.
*/	An asterisk-slash ends a multiline comment.
.	A dot separates an object name from a component name and so qualifies a reference. For example, dot notation is used to select fields in a record and to specify declarations within a package.
..	Two dots separate the lowest and highest values in a range.
...	An ellipsis shows that statements or clauses irrelevant to the discussion were left out.

PL/SQL syntax is described using a variant of Backus-Naur Form (BNF), which includes the following symbols:

[ ]	Square brackets enclose optional items.
{ 'D	Braces enclose items only one of which is required.
|	A vertical bar separates alternatives within brackets or braces.
...	An ellipsis shows that the preceding parameter can be repeated.

Character Set

You write a PL/SQL program as lines of text using a specific set of characters. The PL/SQL character set includes

```
    * the upper- and lower-case letters A .. Z, a .. z
    * the numerals 0 .. 9
    * tabs, spaces, and carriage returns
    * the symbols (  )  +  -  *  /  <  >  =  !  ~  ;  :  .  '
                  @  %  ,  "  #  $  ^  &  |  _  {  'D  ?  [  ]
```

PL/SQL is not case-sensitive, so lower-case letters are equivalent to corresponding upper-case letters except within string and character literals.

Lexical Units

A line of PL/SQL text contains groups of characters known as lexical units, which can be classified as follows:

```
    * delimiters (simple and compound symbols)
    * identifiers, which include reserved words
    * literals
    * comments
```

For example, the following line contains two identifiers, a compound symbol, two simple symbols, a numeric literal, and a comment:

```
    bonus := salary * 0.10;  -- compute bonus
```

To improve readability, you can separate lexical units by spaces. In fact, you must separate adjacent identifiers by a space or punctuation. However, you cannot embed spaces in lexical units except for string literals and comments.

See also: Comments, Delimiters, Identifiers, Literals,
Reserved Words

Delimiters

A delimiter is a simple or compound symbol that has a special meaning to PL/SQL. For example, you use delimiters to represent arithmetic operations such as addition and subtraction. Simple symbols consist of one character; compound symbols consist of two characters.

Simple Symbols		Compound Symbols	
+	addition operator	**	exponentiation operator
-	subtraction/negation operator	<>	relational operator
*	multiplication operator	!=	relational operator
/	division operator	~=	relational operator
=	relational operator	^=	relational operator
<	relational operator	<=	relational operator
>	relational operator	>=	relational operator
(	expression or list delimiter	:=	assignment operator
)	expression or list delimiter	=>	association operator
;	statement terminator	..	range operator
%	attribute indicator	||	concatenation operator
,	item separator	<<	label delimiter
.	component selector	>>	label delimiter
@	remote access indicator	--	single-line comment indicator
'	character string delimiter	/*	multiline comment delimiter
"	quoted identifier delimiter	*/	multiline comment delimiter
:	host variable indicator		

See also: Lexical Units

Identifiers

You use identifiers to name PL/SQL program objects and units, which include constants, variables, exceptions, cursors, subprograms, and packages.   Some examples of identifiers follow:

```
X
t2
phone#
credit_limit
LastName
oracle$number
```

An identifier consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs.   Other characters such as hyphens, slashes, and spaces are illegal, as the following examples show:

```
mine&yours    -- illegal ampersand
debit-amount  -- illegal hyphen
on/off        -- illegal slash
user id       -- illegal space
```

The next examples show that adjoining and trailing dollar signs, underscores, and number signs are legal:

```
money$$$tree  -- legal
SN##          -- legal
try_again_    -- legal
```

You can use upper, lower, or mixed case to write identifiers.   PL/SQL is not case-sensitive except within string and character literals.   So, if the only difference between identifiers is the case of corresponding letters, PL/SQL considers the identifiers to be the same, as the following example shows:

```
lastname
LastName  -- same as lastname
LASTNAME  -- same as lastname and LastName
```

The length of an identifier cannot exceed 30 characters.   But, every character, including dollar signs, underscores, and number signs, is significant.   For example, PL/SQL considers the following identifiers to be different:

```
lastname
last_name
```

Quoted Identifiers

For flexibility, PL/SQL lets you enclose identifiers within double quotes.   Quoted identifiers are seldom needed, but occasionally they can be useful.   They can contain any sequence of printable characters including spaces but excluding double quotes.   Hence, the following identifiers are legal:

```
    "X+Y"
    "last name"
    "on/off switch"
    "employee(s)"
```

The maximum length of a quoted identifier is 30 characters not counting the double quotes.

Using PL/SQL reserved words as quoted identifiers is allowed but not recommended.   It is poor programming practice to reuse reserved words. However, some PL/SQL reserved words are not reserved by SQL.   For example, the PL/SQL reserved word TYPE can be used in a CREATE TABLE statement to name a database column.   But, if a SQL statement in your PL/SQL program refers to that column, you get a compilation error, as the following example shows:

```
    SELECT acct, type, bal INTO ...  -- causes compilation error
```

To prevent the error, enclose the upper-case column name in double quotes as follows:

```
    SELECT acct, "TYPE", bal INTO ...
```

The column name cannot appear in lower or mixed case (unless it was defined that way in the CREATE TABLE statement).   For example, the following statement is invalid:

```
    SELECT acct, "type", bal INTO ...  -- causes compilation error
```

See also: Lexical Units, Variables and Constants,
          Reserved Words

Reserved Words (PL/SQL)

Some identifiers, called "reserved words," have a special syntactic meaning to PL/SQL and so cannot be redefined. For example, the words BEGIN and END, which bracket the executable part of a block or subprogram, are reserved. As the next example shows, if you try to redefine a reserved word, you get a compilation error:

```
DECLARE
    end  BOOLEAN;  -- illegal; causes compilation error
```

However, you can embed reserved words in an identifier, as the following example shows:

```
DECLARE
    end_of_game  BOOLEAN;  -- legal
```

Typically, reserved words are written in upper case to promote readability. However, like other PL/SQL identifiers, reserved words can be written in lower or mixed case.

Following is a list of the reserved words:

ABORT	AUTHORIZATION	CLUSTERS	DATE	DO
ACCEPT	AVG	COLAUTH	DBA	DROP
ACCESS	BASE_TABLE	COLUMNS	DEBUGOFF	ELSE
ADD	BEGIN	COMMIT	DEBUGON	ELSIF
ALL	BETWEEN	COMPRESS	DECLARE	END
ALTER	BINARY_INTEGER	CONNECT	DECIMAL	ENTRY
AND	BODY	CONSTANT	DEFAULT	EXCEPTION
ANY	BOOLEAN	COUNT	DEFINITION	EXCEPTION_INIT
ARRAY	BY	CRASH	DELAY	EXISTS
ARRAYLEN	CASE	CREATE	DELETE	EXIT
AS	CHAR	CURRENT	DELTA	FALSE
ASC	CHAR_BASE	CURRVAL	DESC	FETCH
ASSERT	CHECK	CURSOR	DIGITS	FLOAT
ASSIGN	CLOSE	DATABASE	DISPOSE	FOR
AT	CLUSTER	DATA_BASE	DISTINCT	FORM
FROM	MIN	PARTITION	ROWNUM	TASK
FUNCTION	MINUS	PCTFREE	ROWTYPE	TERMINATE
GENERIC	MLSLABEL	POSITIVE	RUN	THEN
GOTO	MOD	PRAGMA	SAVEPOINT	TO
GRANT	MODE	PRIOR	SCHEMA	TRUE
GROUP	NATURAL	PRIVATE	SELECT	TYPE
HAVING	NEW	PROCEDURE	SEPARATE	UNION
IDENTIFIED	NEXTVAL	PUBLIC	SET	UNIQUE
IF	NOCOMPRESS	RAISE	SIZE	UPDATE
IN	NOT	RANGE	SMALLINT	USE
INDEX	NULL	REAL	SPACE	VALUES
INDEXES	NUMBER	RECORD	SQL	VARCHAR
INDICATOR	NUMBER_BASE	RELEASE	SQLCODE	VARCHAR2

INSERT	OF	REM	SQLERRM	VARIANCE
INTEGER	ON	RENAME	START	VIEW
INTERSECT	OPEN	RESOURCE	STATEMENT	VIEWS
INTO	OPTION	RETURN	STDDEV	WHEN
IS	OR	REVERSE	SUBTYPE	WHERE
LEVEL	ORDER	REVOKE	SUM	WHILE
LIKE	OTHERS	ROLLBACK	TABAUTH	WITH
LIMITED	OUT	ROWID	TABLE	WORK
LOOP	PACKAGE	ROWLABEL	TABLES	XOR
MAX				

See also: Lexical Units, Identifiers

Literals (PL/SQL)

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. The numeric literal 147 and the Boolean literal FALSE are examples.

Numeric Literals

You can use two kinds of numeric literals in arithmetic expressions: integers and reals. An integer literal is an optionally signed whole number without a decimal point. Some examples follow:

```
    030   6   -14   0   +32767
```

A real literal is an optionally signed whole or fractional number with a decimal point. Several examples follow:

```
    6.6667   0.0   -12.0   3.14159   +8300.00   .5   25.
```

PL/SQL considers numbers such as -12.0 and 25. to be reals even though they have integral values.

Numeric literals cannot contain dollar signs or commas but can be written using scientific notation. Simply suffix the number with an E (or e) followed by an optionally signed integer. A few examples follow:

```
    2E5   1.0E-7   3.14159e0   -1E38   -9.5e-3
```

E stands for "times ten to the power of." As the next example shows, the number after E is the power of ten by which the number before E must be multiplied:

```
    5E3 = 5 x 10**3 = 5 x 1000 = 5000
```

The number after E also corresponds to the number of places the decimal point shifts. In the last example, the implicit decimal point shifted three places to the right.

Character Literals

A character literal is an individual character enclosed by single quotes (apostrophes). Several examples follow:

```
    'Z'   '%'   '7'   ' '   'z'   '('
```

Character literals include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols such as underscores and number signs. PL/SQL is case-sensitive within character literals. For example, PL/SQL considers the literals 'Z' and 'z' to be different.

Do not confuse the character literals '0' .. '9' with integer literals.

Character literals cannot be used in arithmetic expressions.

String Literals

A character value can be represented by an identifier or explicitly written as a string literal, which is a sequence of zero or more characters enclosed by single quotes.   Several examples follow:

```
    'Hello, world!'
    'XYZ Corporation'
    '10-NOV-91'
    'He said "Life is like licking honey from a thorn."'
    '$1,000,000'
```

PL/SQL is case-sensitive within string literals.   For example, PL/SQL considers the following literals to be different:

```
    'baker'
    'Baker'
```

Given that apostrophes (single quotes) delimit string literals, how do you represent an apostrophe within a string?   As the next example shows, you write two single quotes, which is not the same as writing a double quote:

```
    'Don''t leave without saving your work.'
```

All string literals except the null string ('') belong to type CHAR.

Boolean Literals

Boolean literals are the predefined values TRUE and FALSE and the non-value NULL, which stands for a missing, unknown, or inapplicable value.   Keep in mind, Boolean literals are not strings.   For example, TRUE is no less a value than the number 25.

See also: Lexical Units

Comments (PL/SQL)

The PL/SQL compiler ignores comments. However, adding comments to your program promotes readability and aids understanding. Generally, you use comments to describe the purpose and use of each code segment. PL/SQL supports two comment styles: single-line and multiline.

Single-line Comments

Single-line comments begin with a double hyphen (--) anywhere on a line and extend to the end of the line. A few examples follow:

```
-- begin processing
SELECT sal INTO salary FROM emp  -- get current salary
    WHERE empno = emp_id;
bonus := salary * 0.15;  -- compute bonus amount
```

Multiline Comments

Multiline comments begin with a slash-asterisk (/*), end with an asterisk-slash (*/), and can span multiple lines. An example follows:

```
IF rating > 90 THEN          /* Compute a 15% bonus for
                                 top-rated employees. */
    bonus := salary * 0.15;
END IF;
```

Disabling Code

While testing or debugging a program, you might want to disable some code. The following example shows how you can "comment-out" a single line or a whole section:

```
-- DELETE FROM emp WHERE comm IS NULL;
...
/* OPEN c1;
   LOOP
       FETCH c1 INTO my_empno, my_ename, my_sal;
       EXIT WHEN c1%NOTFOUND;
       ...
   END LOOP;
   CLOSE c1; */
```

See also: Lexical Units

Datatypes (PL/SQL)

Every constant and variable has a datatype, which specifies a storage format, constraints, and valid range of values. PL/SQL provides a variety of predefined scalar and composite datatypes. A scalar type has no internal components. A composite type has internal components that can be manipulated individually.

The predefined types available for your use are shown below. An additional scalar type, MLSLABEL is available with Trusted Oracle, a specially secured version of Oracle. The scalar types fall into four families, which store number, character, date/time, or Boolean data, respectively.

```
         Scalar Types              Composite Types
-----------------------------      ---------------
BINARY_INTEGER      CHAR           RECORD    TABLE
DEC                 CHARACTER
DECIMAL             LONG
DOUBLE PRECISION    LONG RAW
FLOAT               RAW
INT                 ROWID
INTEGER             STRING
NATURAL             VARCHAR
NUMBER              VARCHAR2
NUMERIC
POSITIVE            DATE
REAL
SMALLINT            BOOLEAN
```

BINARY_INTEGER Type

You use the BINARY_INTEGER datatype to store signed integers. The magnitude range of a BINARY_INTEGER value is -2147483647 .. 2147483647. PL/SQL represents BINARY_INTEGER values as signed binary numbers, which, unlike NUMBER values, can be used in calculations without conversion.

For convenience, PL/SQL predefines the following BINARY_INTEGER subtypes:

```
NATURAL  (0 .. 2147483647)
POSITIVE  (1 .. 2147483647)
```

You can use the NATURAL or POSITIVE subtype when you want to restrict a variable to non-negative integer values.

NUMBER Type

You use the NUMBER datatype to store fixed or floating point numbers of virtually any size. You can specify precision, which is the total number of digits, and scale, which determines where rounding occurs. The syntax follows:

```
    NUMBER[(precision, scale)]
```

You cannot use constants or variables to specify precision and scale; you must use integer literals. The maximum precision of a NUMBER value is 38; the magnitude range is 1.0E-129 .. 9.99E125. If you do not specify the precision, it defaults to the maximum value supported by your system.

Scale can range from -84 to 127. For instance, a scale of 2 rounds to the nearest hundredth (3.456 becomes 3.46). Scale can be negative, which causes rounding to the left of the decimal point. For example, a scale of -3 rounds to the nearest thousand (3456 becomes 3000). A scale of zero rounds to the nearest whole number. If you do not specify the scale, it defaults to zero.

The NUMBER subtypes below have the same range of values as their base type. For example, FLOAT is just another name for NUMBER.

```
    DEC
    DECIMAL
    DOUBLE PRECISION
    FLOAT
    INTEGER
    INT
    NUMERIC
    REAL
    SMALLINT
```

You can use these subtypes for compatibility or when you want an identifier more descriptive than NUMBER.

CHAR Type

You use the CHAR datatype to store fixed-length character data. How the data is represented internally depends on the database character set, which might be 7-bit ASCII or EBCDIC Code Page 500 for example.

The CHAR datatype takes an optional parameter that lets you specify a maximum length up to 32767 bytes. The syntax follows:

```
    CHAR[(maximum_length)]
```

You cannot use a constant or variable to specify the maximum length; you must use an integer literal. If you do not specify the maximum length, it defaults to 1.

The CHAR subtypes below have the same range of values as their base type. For example, STRING is just another name for CHAR.

```
    CHARACTER
    STRING
```

You can use these subtypes for compatibility or when you want an identifier more descriptive than CHAR.

VARCHAR2 Type

You use the VARCHAR2 datatype to store variable-length character data. The VARCHAR2 datatype takes a required parameter that lets you specify a maximum length up to 32767 bytes. The syntax follows:

```
    VARCHAR2(maximum_length)
```

You cannot use a constant or variable to specify the maximum length; you must use an integer literal.

The VARCHAR subtype has the same range of values as its base type. That is, VARCHAR is just another name for VARCHAR2. You can use this subtype for compatibility. However, the VARCHAR datatype might change to accommodate emerging SQL standards. So, it is a good idea to use VARCHAR2 rather than VARCHAR.

LONG Type

You use the LONG datatype to store variable-length character strings. The LONG datatype is like the VARCHAR2 datatype, except that the maximum length of a LONG value is 32760 bytes.

LONG columns can store text, arrays of characters, or even short documents. You can reference LONG columns in UPDATE, INSERT, and (most) SELECT statements, but not in expressions, SQL function calls, or certain SQL clauses such as WHERE, GROUP BY, and CONNECT BY.

RAW Type

You use the RAW datatype to store binary data or byte strings. For example, a RAW variable might store a sequence of graphics characters or a digitized picture. Raw data is like character data, except that PL/SQL does not interpret raw data. Likewise, Oracle does no character set conversions (from 7-bit ASCII to EBCDIC Code Page 500 for example) when you transmit raw data from one system to another.

The RAW datatype takes a required parameter that lets you specify a maximum length up to 32767 bytes. The syntax follows:

```
    RAW(maximum_length)
```

You cannot use a constant or variable to specify the maximum length; you must use an integer literal.

LONG RAW Type

You use the LONG RAW datatype to store binary data or byte strings.

LONG RAW data is like LONG data, except that LONG RAW data is not interpreted by PL/SQL. The maximum length of a LONG RAW value is 32760 bytes.

BOOLEAN Type

You use the BOOLEAN datatype to store the values TRUE and FALSE and the non-value NULL. NULL stands for a missing, unknown, or inapplicable value.

The BOOLEAN datatype takes no parameters. Only the values TRUE and FALSE and the non-value NULL can be assigned to a BOOLEAN variable. You cannot insert the values TRUE and FALSE into a database column. Furthermore, you cannot select or fetch column values into a BOOLEAN variable.

DATE Type

You use the DATE datatype to store fixed-length date values. The DATE datatype takes no parameters. Valid dates for DATE variables include January 1, 4712 BC to December 31, 4712 AD.

When stored in a database column, date values include the time of day in seconds since midnight. The date portion defaults to the first day of the current month; the time portion defaults to midnight.

ROWID Type

Internally, every table in an Oracle database has a ROWID pseudocolumn, which stores 6-byte binary values called "rowids." Rowids uniquely identify rows and provide the fastest way to access particular rows.
You use the ROWID datatype to store rowids in a readable format. When you select or fetch a rowid into a ROWID variable, you can use the function ROWIDTOCHAR, which converts the binary value to an 18-byte character string and returns it in the format

```
    BBBBBBB.RRRR.FFFF
```

where BBBBBBBB is the block in the database file, RRRR is the row in the block (the first row is 0), and FFFF is the database file. These numbers are hexadecimal. For example, the rowid

```
    0000000E.000A.0007
```

points to the 11th row in the 15th block in the 7th database file.

Typically, ROWID variables are compared to the ROWID pseudocolumn in the WHERE clause of an UPDATE or DELETE statement to identify the latest row fetched by a cursor.

MLSLABEL Type

With Trusted Oracle, you use the MLSLABEL datatype to store variable-length, binary operating system labels.   Trusted Oracle uses labels to control access to data.

You can use the MLSLABEL datatype to define a database column.   And, you can use the %TYPE and %ROWTYPE attributes to reference the column. However, with standard Oracle, such columns can store only nulls.

With Trusted Oracle, you can insert any valid operating system label into a column of type MLSLABEL.   If the label is in text format, Trusted Oracle converts it to a binary value automatically.   The text string can be up to 255 bytes long.   However, the internal length of an MLSLABEL value is between 2 and 5 bytes.   You can also select values from a MLSLABEL column into a character variable.   Trusted Oracle converts the internal binary value to a VARCHAR2 value automatically.

See also: Datatype Conversion, Datatype Comparison Rules

Datatype Conversion

Sometimes it is necessary to convert a value from one datatype to another. For example, if you want to examine a rowid, you must convert it to a character string. PL/SQL supports both explicit and implicit (automatic) datatype conversion.

Explicit Conversion

To specify conversions explicitly, you use built-in functions that convert values from one datatype to another. The table below shows which function to use in a given situation. For example, to convert a CHAR value to a NUMBER value, you use the function TO_NUMBER.

To / From	CHAR	DATE	NUMBER	RAW	ROWID
CHAR		TO_DATE	TO_NUMBER	HEXTORAW	CHARTOROWID
DATE	TO_CHAR				
NUMBER	TO_CHAR	TO_DATE			
RAW	RAWTOHEX				
ROWID	ROWIDTOCHAR				

Implicit Conversion

When it makes sense, PL/SQL can convert the datatype of a value implicitly. This allows you to use literals, variables, and parameters of one type where another type is expected. In the example below, the CHAR variables "start_time" and "finish_time" hold string values representing the number of seconds past midnight. The difference between those values must be assigned to the NUMBER variable "elapsed_time." So, PL/SQL converts the CHAR values to NUMBER values implicitly.

```
DECLARE
    start_time    CHAR(5);
    finish_time   CHAR(5);
    elapsed_time  NUMBER(5);
BEGIN
    SELECT TO_CHAR(SYSDATE,'SSSSS') INTO start_time FROM sys.dual;
    -- do something
    SELECT TO_CHAR(SYSDATE,'SSSSS') INTO finish_time FROM sys.dual;
    elapsed_time := finish_time - start_time;
    ...
END;
```

The table below shows the implicit conversions PL/SQL can do. If PL/SQL cannot determine which implicit conversion is needed, you get a compilation error. In such cases, use datatype conversion functions.

To / From	CHAR	DATE	LONG	NUMBER	RAW	ROWID	VARCHAR2

CHAR		yes	yes	yes	yes	yes	yes
DATE	yes		yes				yes
LONG	yes				yes		yes
NUMBER	yes		yes				yes
RAW	yes		yes				yes
ROWID	yes						yes
VARCHAR2	yes	yes	yes	yes	yes	yes	

It is your responsibility to ensure that values are convertible. For instance, PL/SQL can convert the CHAR value '02-JUN-92' to a DATE value, but PL/SQL cannot convert the CHAR value 'YESTERDAY' to a DATE value. Similarly, PL/SQL cannot convert a CHAR value containing alphabetic characters to a NUMBER value.

See also: Datatypes, Datatype Comparison Rules

Subtypes

Each PL/SQL datatype specifies a set of values and a set of operations applicable to objects of that type.   Subtypes specify the same set of operations as their base type but only a subset of its values.   Thus, a subtype does not introduce a new type; it merely places an optional constraint on its base type.

Kinds of Subtypes

PL/SQL predefines several subtypes in package STANDARD.   For example, the subtype NATURAL is defined as follows:

```
    SUBTYPE NATURAL IS BINARY_INTEGER RANGE 0..2147483647;
```

The RANGE constraint restricts objects of type NATURAL to the subset of BINARY_INTEGER values in the range 0..2147483647.   Thus, NATURAL is a "constrained" subtype.   Constrained subtypes can increase reliability by detecting nulls or out-of-range values.   For example, if you try to store a negative value in a NATURAL variable, PL/SQL raises the predefined exception VALUE_ERROR.

Every set is a subset of itself.   So, any type can be a subtype of itself.   For instance, PL/SQL predefines the subtype CHARACTER in package STANDARD as follows:

```
    SUBTYPE CHARACTER IS CHAR;
```

The subtype CHARACTER specifies the same set of values as its base type CHAR.   Thus, CHARACTER is an "unconstrained" subtype.   Unconstrained subtypes can provide compatibility with ANSI/ISO and IBM types. Also, they can improve readability by indicating the intended use of constants and variables.

The current release of PL/SQL lets you define your own unconstrained subtypes.   Future releases of PL/SQL will let you define constrained subtypes as well.

Declaring Subtypes

You can declare unconstrained subtypes in the declarative part of any PL/SQL block, subprogram, or package using the syntax

```
    SUBTYPE subtype_name IS base_type;
```

where "subtype_name" is a type specifier used in subsequent declarations and "base_type" stands for the following syntax:

```
    {type_name | subtype_name | variable%TYPE |
       table.column%TYPE | table%ROWTYPE'D
```

The base type can be a predefined type, a user-defined RECORD or TABLE type, a predefined subtype, or a user-defined subtype.   For example, all of the following declarations are legal:

```
DECLARE
   SUBTYPE EmpDate IS DATE;       -- based on predefined type
   TYPE NameTab IS TABLE OF CHAR(10)
      INDEX BY BINARY_INTEGER;
   SUBTYPE EnameTab IS NameTab;   -- based on user-defined type
   SUBTYPE Counter IS NATURAL;    -- based on predefined subtype
   SUBTYPE Totalizer IS Counter;  -- based on user-defined subtype
   ...
```

The base type cannot be constrained. For example, the following declarations are illegal:

```
DECLARE
   SUBTYPE Accumulator IS NUMBER(9,2); -- illegal; must be NUMBER
   SUBTYPE Delimiter IS CHAR(1);       -- illegal; must be CHAR
   SUBTYPE Text IS VARCHAR2(n);        -- illegal
   ...
```

Using Subtypes

Once you define a subtype, you can declare objects of that type. In the next example, you declare two variables of type "Counter":

```
DECLARE
   SUBTYPE Counter IS NATURAL;
   rows       Counter;
   employees  Counter;
   ...
```

Notice how the subtype name indicates the intended use of the variables.

Type Compatibility

Objects of a subtype are always compatible with objects of its base type. For example, given the following declarations, the value of "salary" can be assigned to "total" without conversion:

```
DECLARE
   SUBTYPE Accumulator IS NUMBER;
   salary  NUMBER(7,2);
   total   Accumulator;
   ...
```

Objects of different subtypes are compatible if the subtypes have the same base type. For instance, given the following declarations, the value of "finished" can be assigned to "debugging":

```
DECLARE
   SUBTYPE Sentinel IS BOOLEAN;
   SUBTYPE Switch IS BOOLEAN;
```

```
      finished   Sentinel;
      debugging  Switch;
      ...
```

Objects of different subtypes are also compatible if their base types belong to the same datatype family. For example, given the following declarations, the value of "comma" can be assigned to "line":

```
   DECLARE
      SUBTYPE Delimiter IS CHAR;
      SUBTYPE Text IS LONG;
      comma  Delimiter;
      line   Text;
      ...
```

Overloading

You cannot overload two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same family. For example, you cannot overload the following procedures because the base types CHAR and LONG are in the same family:

```
   DECLARE
      SUBTYPE Delimiter IS CHAR;
      SUBTYPE Text IS LONG;
      ...
      PROCEDURE scan (x Delimiter) IS
         BEGIN ... END;
      PROCEDURE scan (x Text) IS
         BEGIN ... END;
    BEGIN
```

See also: Datatypes, Datatype Comparison Rules, Datatype Conversion

PL/SQL Blocks

PL/SQL is a block-structured language.   That is, the basic units that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks.   Typically, each logical block corresponds to a problem or subproblem to be solved.

A block (or sub-block) lets you group logically related declarations and statements.   That way, you can place declarations close to where they are used.   The declarations are local to the block and cease to exist when the block completes.

As the following syntax shows, a PL/SQL block has three parts: an optional declarative part, an executable part, and an optional exception-handling part:

```
[DECLARE
    -- declarations]
BEGIN
    -- statements
[EXCEPTION
    -- handlers]
END;
```

The order of the parts is logical.   First comes the declarative part in which objects can be declared.   Once declared, objects can be manipulated in the executable part.   Exceptions raised during execution can be dealt with in the exception-handling part.

You can nest sub-blocks in the executable and exception-handling parts of a PL/SQL block or subprogram but not in the declarative part.

See also: Variables and Constants, Exceptions, Subprograms, Block Label

Block Label

A block label is an undeclared identifier that labels a PL/SQL block. It must be enclosed by double angle brackets and must appear at the beginning of the block. (However, in the SQL*Plus environment, the first line you input cannot start with a block label.) Optionally, a block label can also appear at the end of the block.

A global identifier declared in an enclosing block can be redeclared in a sub-block, in which case the local declaration prevails and the sub-block cannot reference the global identifier. To reference the global identifier, you must use a block label to qualify the reference, as follows:

```
block_label.global_identifier
```

In the following example, you compare two INTEGER variables declared with the same name, one in an enclosing block, the other in a sub-block:

```
<<outer>>
DECLARE
    x  INTEGER;
BEGIN
    ...
    DECLARE
        x  INTEGER;
    BEGIN
        ...
        IF x = outer.x THEN
            ...
        END IF;
    END;
END outer;
```

See also: PL/SQL Blocks, Loop Label

Variables and Constants

Your program stores values in variables and constants. As the program executes, the values of variables can change, but the values of constants cannot.

You can declare variables and constants in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its datatype, and name the storage location so that the value can be referenced. They can also assign an initial value and specify the NOT NULL constraint. A few examples follow:

```
    birthdate  DATE;
    emp_count  SMALLINT := 0;
    acct_id    VARCHAR2(5) NOT NULL := 'AP001';
```

The first declaration names a variable of type DATE. The second declaration names a variable of type SMALLINT and uses the assignment operator (:=) to assign an initial value of zero to the variable. The third declaration names a variable of type VARCHAR2, specifies the NOT NULL constraint, and assigns an initial value of 'AP001' to the variable.

You cannot assign nulls to a variable or constant defined as NOT NULL. If you try, the predefined exception VALUE_ERROR is raised. The NOT NULL constraint must be followed by an initialization clause; otherwise, you get a compilation error. For example, the following declaration is illegal:

```
    acct_id  VARCHAR2(5) NOT NULL;  -- illegal; not initialized
```

The next examples show that the expression following the assignment operator can be arbitrarily complex and can refer to previously initialized variables and constants:

```
    pi      CONSTANT REAL := 3.14159;
    radius  REAL := 1;
    area    REAL := pi * radius**2;
```

In constant declarations, the reserved word CONSTANT must precede the type specifier, as the following example shows:

```
    credit_limit  CONSTANT REAL := 5000.00;
```

This declaration names a constant of type REAL and assigns an initial (also final) value of 5000 to the constant. A constant must be initialized in its declaration; otherwise, you get a compilation error.

Within the same scope, all declared identifiers must be unique. So, even if their datatypes differ, variables and parameters cannot share the same name. For example, two of the following declarations are illegal:

```
    DECLARE
```

```
        valid_id  BOOLEAN;
        valid_id  VARCHAR2(5);  -- illegal duplicate identifier
        valid_id  INTEGER;      -- illegal triplicate identifier
        ...
```

PL/SQL does not allow forward references. You must declare a variable or constant before referencing it in other statements, including other declarative statements. For example, the following declaration of "maxi" is illegal:

```
    maxi  INTEGER := 2 * mini;
    mini  INTEGER := 15;
```

Some languages allow you to declare a list of variables belonging to the same datatype. PL/SQL does not allow this. For example, the following declaration is illegal:

```
    i, j, k  SMALLINT;  -- illegal
```

The legal version follows:

```
    i  SMALLINT;
    j  SMALLINT;
    k  SMALLINT;
```

DEFAULT Keyword

If you prefer, you can use the reserved word DEFAULT instead of the assignment operator to initialize variables and constants. For example, the declarations

```
    tax_year  SMALLINT := 92;
    valid     BOOLEAN := FALSE;
```

can be rewritten as follows:

```
    tax_year  SMALLINT DEFAULT 92;
    valid     BOOLEAN DEFAULT FALSE;
```

You can also use DEFAULT to initialize subprogram parameters, cursor parameters, and fields in a user-defined record.

See also: Datatypes, %ROWTYPE Attribute, %TYPE Attribute

%TYPE Attribute

The %TYPE attribute provides the datatype of a variable, constant, or database column. In the following example, %TYPE provides the datatype of a variable:

```
    credit  REAL(7,2);
    debit   credit%TYPE;
```

Variables and constants declared using %TYPE are treated like those declared using a datatype name. For example, given the previous declarations, PL/SQL treats "debit" like a REAL(7,2) variable.
The next example shows that a %TYPE declaration can include an initialization clause:

```
    balance          NUMBER(7,2);
    minimum_balance  balance%TYPE := 10.00;
```

The %TYPE attribute is particularly useful when declaring variables that refer to database columns. You can reference a table and column, or you can reference a schema, table, and column, as the following example shows:

```
    my_dname  scott.dept.dname%TYPE;
```

Using %TYPE to declare "my_dname" has two advantages. First, you need not know the exact datatype of "dname." Second, if the database definition of "dname" changes, the datatype of "my_dname" changes accordingly at run time.

Note, however, that a NOT NULL column constraint does not apply to variables declared using %TYPE. In the next example, even though the database column "empno" is defined as NOT NULL, you can assign a null to the variable "my_empno":

```
    DECLARE
        my_empno  emp.empno%TYPE;
        ...
    BEGIN
        my_empno := NULL;  -- this works
        ...
    END;
```

See also: Datatypes, Variables and Constants,
          %ROWTYPE Attribute

%ROWTYPE Attribute

The %ROWTYPE attribute provides a record type that represents a row in a table (or view). The record can store an entire row of data selected from the table or fetched by a cursor. In the example below, you declare two records. The first record stores a row selected from the "emp" table. The second record stores a row fetched by the "c1" cursor.

```
DECLARE
    emp_rec   emp%ROWTYPE;
    CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
    dept_rec  c1%ROWTYPE;
    ...
```

Columns in a row and corresponding fields in a record have the same names and datatypes. In the following example, you select column values into a record named "emp_rec":

```
DECLARE
    emp_rec  emp%ROWTYPE;
    ...
BEGIN
    SELECT * INTO emp_rec FROM emp WHERE ...
    ...
END;
```

The column values returned by the SELECT statement are stored in fields. To reference a field, you use dot notation. For example, you might reference the "deptno" field as follows:

```
IF emp_rec.deptno = 20 THEN ...
```

In addition, you can assign the value of an expression to a specific field, as the following examples show:

```
emp_rec.ename := 'JOHNSON';
emp_rec.sal := emp_rec.sal * 1.05;
```

Aggregate Assignment

A %ROWTYPE declaration cannot include an initialization clause. However, there are two ways to assign values to all fields in a record at once. First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor. For example, the following assignments are legal:

```
DECLARE
    dept_rec1  dept%ROWTYPE;
    dept_rec2  dept%ROWTYPE;
    CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
    dept_rec3  c1%ROWTYPE;
    dept_rec4  c1%ROWTYPE;
BEGIN
    ...
    dept_rec1 := dept_rec2;
    dept_rec4 := dept_rec3;
    ...
```

But, because "dept_rec2" is based on a table and "dept_rec3" is based
on a cursor, the following assignment is illegal:

```
    dept_rec2 := dept_rec3;  -- illegal
```

Second, you can assign a list of column values to a record by using the
SELECT or FETCH statement, as the example below shows.   The column names
must appear in the order in which they were defined by the CREATE TABLE
or CREATE VIEW statement.

```
    DECLARE
        dept_rec  dept%ROWTYPE;
        ...
    BEGIN
        SELECT deptno, dname, loc INTO dept_rec FROM dept
            WHERE deptno = 30;
        ...
    END;
```

However, you cannot assign a list of column values to a record by using
an assignment statement.   So, the following syntax is illegal:

```
    record_name := (value1, value2, value3, ...);  -- illegal
```

Although you can retrieve entire records, you cannot insert them.   For
example, the following statement is illegal:

```
    INSERT INTO dept VALUES (dept_rec1);  -- illegal
```

Using Aliases

Select-items fetched by a cursor associated with %ROWTYPE must have
simple names or, if they are expressions, must have aliases.   In the
following example, you use an alias called "wages":

```
    DECLARE
        CURSOR my_cursor IS SELECT sal + NVL(comm, 0) wages, ename
            FROM emp;
        my_rec  my_cursor%ROWTYPE;
    BEGIN
        OPEN my_cursor;
        LOOP
            FETCH my_cursor INTO my_rec;
            EXIT WHEN my_cursor%NOTFOUND;
            IF my_rec.wages > 2000 THEN
                INSERT INTO temp VALUES (NULL, my_rec.wages,
                    my_rec.ename);
            END IF;
        END LOOP;
        CLOSE my_cursor;
    END;
```

See also: Datatypes, %TYPE Attribute, Records, Assignments

Naming Conventions

The same naming conventions apply to all PL/SQL program objects and units including constants, variables, cursors, exceptions, procedures, functions, and packages.   Names can be simple, qualified, remote, or both qualified and remote.   For example, you might use the procedure name "raise_salary" in any of the following ways:

```
    raise_salary(...);                        -- simple
    emp_actions.raise_salary(...);            -- qualified
    raise_salary@newyork(...);                -- remote
    emp_actions.raise_salary@newyork(...);  -- qualified and remote
```

In the first case, you simply use the procedure name.   In the second case, you must qualify the name using dot notation because the procedure is stored in a package called "emp_actions."   In the third case, you reference the database link "newyork" because the (standalone) procedure is stored in a remote database.   In the fourth case, you qualify the procedure name and reference a database link.

You can create synonyms to provide location transparency for remote database objects such as tables, sequences, views, standalone subprograms, and packages.   However, you cannot create synonyms for objects declared within subprograms or packages.   That includes constants, variables, cursors, exceptions, and packaged procedures.

See also: Global Names, Location Transparency, Remote Access

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.

For example, 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 objects 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 nested at the same level because those identifiers are neither local nor global to the block. The following example illustrates the scope rules:

```
    DECLARE
        A  CHAR;
        B  REAL;
    BEGIN
        -- identifiers available here: A (CHAR), B

        DECLARE
            A  INTEGER;
            C  REAL;
        BEGIN
            -- identifiers available here: A (INTEGER), B, C
        END;

        DECLARE
            D  REAL;
        BEGIN
            -- identifiers available here: A (CHAR), B, D
        END;

        -- identifiers available here: A (CHAR), B
    END;
```

Global identifiers can be redeclared in a sub-block, in which case the local declaration prevails and the sub-block cannot reference the global identifier unless you use a qualified name. The qualifier can be the label of an enclosing block as the following example shows:

```
    <<outer>>
    DECLARE
```

```
        birthdate  DATE;
    BEGIN
        ...
        DECLARE
            birthdate  DATE;
        BEGIN
            ...
            IF birthdate = outer.birthdate THEN
                ...
            END IF;
        END;
    END outer;
```

Or, as the next example shows, the qualifier can be the name of an enclosing subprogram:

```
    PROCEDURE check_credit (...) IS
        rating  NUMBER;
        ...
        FUNCTION valid (...) RETURN BOOLEAN IS
            rating  NUMBER;
        BEGIN
            ...
            IF check_credit.rating < 3 THEN
                ...
        END valid;
    BEGIN
        ...
    END check_credit;
```

See also: Variables and Constants, PL/SQL Blocks, Overloading

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
       ...
```

Therefore, 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.

Alternatively, you can use the SELECT or FETCH statement to have Oracle assign values to a variable. An example follows:

```
SELECT ename, sal + comm INTO last_name, wages FROM emp
    WHERE empno = emp_id;
```

For each item in the SELECT list, there must be a corresponding variable in the INTO list. Also, each item must return a value that is implicitly convertible to the datatype of its corresponding variable.

Boolean Variables

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

```
DECLARE
    done  BOOLEAN;
    ...
```

the following assignment statements are legal:

```
BEGIN
    done := FALSE;
    WHILE NOT done LOOP
        ...
        done := (total > 500);
    END LOOP;
END;
```

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

See also: Variables and Constants, FETCH Statement, SELECT

Expressions

Expressions are constructed using operands and operators. An operand is a variable, constant, literal, or function call that contributes a value to an expression. An example of a simple arithmetic expression follows:

```
    -X / 2
```

Unary operators such as the negation operator (-) operate on one operand; binary operators such as the division operator (/) operate on two operands. PL/SQL has no ternary operators.

The simplest expressions consist of a single variable, which yields a value directly. PL/SQL evaluates (finds the current value of) an expression by combining the values of the operands in ways specified by the operators. This always yields a single value and datatype. PL/SQL determines the datatype by examining the expression and the context in which it appears.

See also: Boolean Expressions, Comparison Operators,
Concatenation Operator, Logical Operators,
Operator Precedence

Operator Precedence

The operations within an expression are done in a particular order depending on their precedence (priority). The following table shows the default order of operations from first to last (top to bottom).

Operator	Operation
**, NOT	exponentiation, logical negation
+, -	identity, negation
*, /	multiplication, division
+, - , ||	addition, subtraction, concatenation
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	comparison
AND	conjunction
OR	inclusion

Operators with higher precedence are applied first. For example, both of the following expressions evaluate to 8 because division has a higher precedence than addition:

```
5 + 12 / 4
12 / 4 + 5
```

Operators with the same precedence are applied in no particular order. You can use parentheses to control the order of evaluation. For example, the following expression evaluates to 7, not 11, because parentheses override the default operator precedence:

```
(8 + 6) / 2
```

In the next example, the subtraction is done before the division because the most deeply nested subexpression is always evaluated first:

```
100 + (20 / 5 + (7 - 3))
```

See also: Expressions, Comparison Operators, Literals
Logical Operators

Logical Operators (PL/SQL)

The logical operators AND, OR, and NOT operate according to the tri-state logic illustrated by the truth tables shown below. AND and OR are binary operators; NOT is a unary operator.

```
NOT   | true   false  null
------|-------------------
      | false  true   null

AND   | true   false  null      OR    | true   false  null
------|--------------------     ------|-------------------
true  | true   false  null      true  | true   true   true
false | false  false  false     false | true   false  null
null  | null   false  null      null  | true   null   null
```

As the truth tables show, AND returns the value TRUE only if both its operands are true.   On the other hand, OR returns the value TRUE if either of its operands is true.   NOT returns the opposite value (logical negation) of its operand.   For example, NOT TRUE returns FALSE. NOT NULL returns NULL because nulls are indeterminate.

When you do not use parentheses to specify the order of evaluation, operator precedence determines the order.

See also: Expressions, Comparison Operators, Handling Nulls

Comparison Operators (PL/SQL)

Comparison operators compare one expression to another. The result is always TRUE, FALSE, or NULL. Typically, you use comparison operators in the WHERE clause of SQL data manipulation statements and in conditional control statements.

Relational Operators

The relational operators allow you to compare arbitrarily complex expressions. The following table gives the meaning of each operator:

```
Operator  Meaning
-------------------------------------
=         is equal to
!=        is not equal to
<         is less than
>         is greater than
<=        is less than or equal to
>=        is greater than or equal to
```

IS NULL Operator

The IS NULL operator returns the Boolean value TRUE if its operand is null, or FALSE if it is not null. Comparisons involving nulls always yield NULL. So, to test whether a value is NULL, do not use the statement

```
IF variable = NULL THEN ...
```

Instead, use the following statement:

```
IF variable IS NULL THEN ...
```

LIKE Operator

You use the LIKE operator to compare a character value to a pattern. Case is significant. LIKE returns the Boolean value TRUE if the character patterns match or FALSE if they do not match.

The patterns matched by LIKE can include two special-purpose characters called "wildcards." An underscore (_) matches exactly one character; a percent sign (%) matches zero or more characters. For example, if the value of "ename" is 'JOHNSON', the following expression evaluates to TRUE:

```
ename LIKE 'J%SON'
```

BETWEEN Operator

The BETWEEN operator tests whether a value lies in a specified range. It means "greater than or equal to <low value> and less than or equal to <high value>."   For example, the following expression evaluates to FALSE:

```
    45 BETWEEN 38 AND 44
```

IN Operator

The IN operator tests set membership.   It means "equal to any member of."   The set can contain nulls, but they are ignored.   For example, the following statement does not delete rows in which the "ename" column is null:

```
    DELETE FROM emp WHERE ename IN (NULL, 'KING', 'FORD');
```

Furthermore, expressions of the form

```
    value NOT IN set
```

evaluate to FALSE if the set contains a null.   For example, instead of deleting rows in which the "ename" column is not null and not 'KING', the following statement deletes no rows:

```
    DELETE FROM emp WHERE ename NOT IN (NULL, 'KING');
```

See also: Expressions, Logical Operators, Handling Nulls

Concatenation Operator

The concatenation operator (||) appends one string to another.   For example, the following expression returns the value 'suitcase':

```
    'suit' || 'case'
```

If both operands belong to type CHAR, the concatenation operator returns a CHAR value.   Otherwise, it returns a VARCHAR2 value.

The concatenation operator ignores null operands.   For example, the following expression returns the value 'applesauce':

```
    'apple' || NULL || NULL || 'sauce'
```

See also: Expressions, CONCAT

Boolean Expressions

PL/SQL lets you compare variables and constants in both SQL and procedural statements. These comparisons, called "Boolean expressions," consist of simple or complex expressions separated by relational operators. Often, Boolean expressions are connected by the logical operators AND, OR, and NOT. A Boolean expression always evaluates to TRUE, FALSE, or NULL.

In a SQL statement, Boolean expressions let you specify the rows in a table that are affected by the statement. In a procedural statement, Boolean expressions are the basis for conditional control. There are three kinds of Boolean expressions: arithmetic, character, and date.

Arithmetic Expressions

You can use the relational operators to compare numbers for equality or inequality. Comparisons are quantitative; that is, one number is greater than another if it represents a larger quantity. For example, given the assignments

```
number1 := 75;
number2 := 70;
```

the following expression evaluates to TRUE:

```
number1 > number2
```

Character Expressions

You can also compare character values for equality or inequality. Comparisons are based on the collating sequence used for the database character set. A "collating sequence" is an internal ordering of the database character set in which a range of numeric codes represent the individual characters. One character value is greater than another if its internal numeric value is larger. For example, given the assignments

```
string1 := 'Kathy';
string2 := 'Kathleen';
```

the following expression evaluates to TRUE:

```
string1 > string2
```

Date Expressions

You can also compare dates. Comparisons are chronological; that is, one date is greater than another if it is more recent. For example, given the assignments

```
date1 := '01-JAN-91';
```

```
    date2 := '31-DEC-90';
```

the following expression evaluates to TRUE:

```
    date1 > date2
```

See also: Condition, Expressions, Comparison Operators,
          Logical Operators, Datatype Comparison Rules

Handling Nulls

Nulls can cause unexpected results. You can avoid some common mistakes by keeping the following rules in mind:

```
    * comparisons involving nulls always yield NULL
    * applying the logical operator NOT to a null yields NULL
    * in conditional control statements, if the condition evaluates
      to NULL, its associated sequence of statements is not executed
```

In the example below, you might expect the sequence of statements to execute because "x" and "y" seem unequal. But remember, nulls are indeterminate. Whether "x" is equal to "y" or not is unknown. Therefore, the IF condition evaluates to NULL and the sequence of statements is bypassed.

```
    x := 5;
    y := NULL;
    ...
    IF x != y THEN  -- evaluates to NULL, not TRUE
         sequence_of_statements;  -- not executed
    END IF;
```

In the next example, you might expect the sequence of statements to execute because "a" and "b" seem equal. But, again, that is unknown, so the IF condition evaluates to NULL and the sequence of statements is bypassed.

```
    a := NULL;
    b := NULL;
    ...
    IF a = b THEN  -- evaluates to NULL, not TRUE
        sequence_of_statements;  -- not executed
```

NOT Operator

Applying the logical operator NOT to a null yields NULL. Thus, the following two statements are not always equivalent:

```
    IF x > y THEN       |     IF NOT x > y THEN
        high := x;      |         high := y;
    ELSE                |     ELSE
        high := y;      |         high := x;
    END IF;             |     END IF;
```

The sequence of statements in the ELSE clause is executed when the IF condition evaluates to FALSE or NULL. So, if either or both "x" and "y" are null, the first IF statement assigns the value of "y" to "high," but the second IF statement assigns the value of "x" to "high." If neither "x" nor "y" is null, both IF statements assign the same value to "high."

Zero-Length Strings

PL/SQL treats any zero-length string like a null. This includes values returned by character functions and Boolean expressions. For example, the following statements assign nulls to the target variables:

```
    null_string := TO_VARCHAR2('');
    zip_code := SUBSTR(address, 25, 0);
    valid := (name != '');
```

So, use the IS NULL operator to test for null strings, as follows:

```
    IF my_string IS NULL THEN ...
```

See also: Condition, Boolean Expressions, Comparison Operators, Logical Operators

Built-in Functions

PL/SQL provides more than 70 powerful functions to help you manipulate data. You can use them wherever expressions of the same type are allowed. Furthermore, you can nest them. The built-in functions fall into the following categories:

```
    * error-reporting functions
    * number functions
    * character functions
    * conversion functions
    * date functions
    * miscellaneous functions
```

You can use all the built-in functions in SQL statements except the error-reporting functions SQLCODE and SQLERRM. In addition, you can use all the functions in procedural statements except the miscellaneous functions DECODE, DUMP, and VSIZE.

The SQL group functions AVG, MIN, MAX, COUNT, SUM, STDDEV, and VARIANCE are not built into PL/SQL. Nevertheless, you can use them in SQL statements (but not in procedural statements).

See also: SQL Functions, SQLCODE and SQLERRM Functions, Group Functions,
Datatype Conversion

PL/SQL Tables

Objects of type TABLE are called "PL/SQL tables," which are modelled on (but not the same as) database tables. PL/SQL tables use a primary key to give you array-like access to rows. The size of a PL/SQL table is unconstrained. That is, the number of rows in a PL/SQL table can increase dynamically.

PL/SQL tables can have one column and a primary key, neither of which can be named. The column can belong to any scalar type, but the primary key must belong to type BINARY_INTEGER.

Declaring PL/SQL Tables

PL/SQL tables must be declared in two steps. First, you define a TABLE type, then declare PL/SQL tables of that type. You can declare TABLE types in the declarative part of any block, subprogram, or package using the syntax

```
    TYPE type_name IS TABLE OF
        { column_type | variable%TYPE | table.column%TYPE 'D [NOT NULL]
        INDEX BY BINARY_INTEGER;
```

where "type_name" is a type specifier used in subsequent declarations of PL/SQL tables and "column_type" is any scalar datatype such as CHAR, DATE, or NUMBER. You can use the %TYPE attribute to specify a column datatype.

In this example, you declare a TABLE type called "EnameTabTyp":

```
    DECLARE
        TYPE EnameTabTyp IS TABLE OF CHAR(10)
            INDEX BY BINARY_INTEGER;
```

You could have used %TYPE to provide the column datatype, as follows:

```
    DECLARE
        TYPE EnameTabTyp IS TABLE OF emp.ename%TYPE
            INDEX BY BINARY_INTEGER;
```

Once you define type "EnameTabTyp," you can declare PL/SQL tables of that type, as follows:

```
    ename_tab  EnameTabTyp;
```

The identifier "ename_tab" represents an entire PL/SQL table.

Like scalar variables, PL/SQL tables can be declared as the formal parameters of procedures and functions. Some packaged examples follow:

```
    PACKAGE emp_actions IS
        TYPE EnameTabTyp IS TABLE OF emp.ename%TYPE
            INDEX BY BINARY_INTEGER;
        TYPE SalTabTyp IS TABLE OF emp.sal%TYPE
            INDEX BY BINARY_INTEGER;
        ename_tab  EnameTabTyp;
        sal_tab    SalTabTyp;
        ...
        PROCEDURE hire_batch
            (ename_tab  EnameTabTyp,
             sal_tab    SalTabTyp,
             ...);
        PROCEDURE log_names
            (ename_tab  EnameTabTyp,
             num        BINARY_INTEGER);
        ...
    END emp_actions;
```

Referencing PL/SQL Tables

To reference rows in a PL/SQL table, you specify a primary key value using the array-like syntax

```
    plsql_table_name(primary_key_value)
```

where "primary_key_value" belongs to type BINARY_INTEGER. For example, you reference the third row in PL/SQL table "ename_tab" as follows:

```
    ename_tab(3) ...
```

You can assign the value of a PL/SQL expression to a specific row using the following syntax:

```
    plsql_table_name(primary_key_value) := plsql_expression;
```

In the next example, you assign the sum of variables "salary" and "increase" to the fifth row in PL/SQL table "sal_tab":

```
    sal_tab(5) := salary + increase;
```

Until a row is assigned a value, it does not exist. If you try to reference an uninitialized row, PL/SQL raises the predefined exception NO_DATA_FOUND. Consider the following example:

```
    DECLARE
        TYPE JobTabTyp IS TABLE OF CHAR(14)
            INDEX BY BINARY_INTEGER;
        job_tab  JobTabTyp;
    BEGIN
        job_tab(1) := 'CLERK';
        IF job_tab(2) = 'CLERK' THEN  -- raises NO_DATA_FOUND
            ...
        END IF;
        ...
    EXCEPTION
```

```
        WHEN NO_DATA_FOUND THEN
            -- here because job_tab(2) does not exist
            ...
    END;
```

Inserting/Fetching Rows

You must use a loop to INSERT values from a PL/SQL table into a database column. Likewise, you must use a loop to FETCH values from a database column into a PL/SQL table. For example, given the declarations

```
    DECLARE
        TYPE EmpnoTabTyp IS TABLE OF NUMBER(4)
            INDEX BY BINARY_INTEGER;
        TYPE EnameTabTyp IS TABLE OF CHAR(10)
            INDEX BY BINARY_INTEGER;
        ...
        empno_tab  EmpnoTabTyp;
        ename_tab  EnameTabTyp;
        ...
```

you might use the following procedure to INSERT values from the PL/SQL tables into the "emp" database table:

```
    PROCEDURE insert_emp_data
        (rows       BINARY_INTEGER,
         empno_tab  EmpnoTabTyp,
         ename_tab  EnameTabTyp,
         ...) IS
    BEGIN
        FOR i IN 1..rows LOOP
            INSERT INTO emp (empno, ename, ...)
                VALUES (empno_tab(i), ename_tab(i), ...);
        END LOOP;
    END;
```

Conversely, you might use the next procedure to FETCH all rows from the database table into PL/SQL tables "empno_tab" and "ename_tab":

```
    PROCEDURE fetch_emp_data
        (rows       OUT BINARY_INTEGER,
         empno_tab  OUT EmpnoTabTyp,
         ename_tab  OUT EnameTabTyp,
         ...) IS
    BEGIN
        rows := 0;
        FOR emprec IN (SELECT * FROM emp) LOOP
            rows := rows + 1;
            empno_tab(rows) := emprec.empno;
            ename_tab(rows) := emprec.ename;
            ...
        END LOOP;
    END;
```

However, you cannot reference PL/SQL tables in the INTO clause. For example, the following SELECT statement is illegal:

```
    PROCEDURE fetch_emp_data
        (rows       OUT BINARY_INTEGER,
         empno_tab  OUT EmpnoTabTyp,
         ename_tab  OUT EnameTabTyp,
         ...) IS
    BEGIN
        SELECT empno, ename
            INTO empno_tab, ename_tab  -- illegal
            FROM emp;
        ...
    END;
```

Deleting Rows

You cannot delete individual rows from a PL/SQL table because the DELETE statement cannot specify PL/SQL tables. However, you can use a simple workaround to delete entire PL/SQL tables. When you want to delete a PL/SQL table, simply assign a null to it, as shown in the following example:

```
    DECLARE
        TYPE NumTabTyp IS TABLE OF NUMBER
            INDEX BY BINARY_INTEGER;
        sal_tab    NumTabTyp;
        ...
    BEGIN
        /* Load salary table. */
        FOR i IN 1..50 LOOP
            sal_tab(i) := i;
        END LOOP;
        ...
        /* Delete salary table. */
        sal_tab := NULL;  -- releases all PL/SQL table resources
        ...
    END;
```

See also: Datatypes, Variables and Constants, Records,
          FETCH Statement, INSERT, DELETE

User-Defined Records

Objects of type RECORD are called "records." They have uniquely named fields, which can belong to different datatypes. For example, suppose you have different kinds of data about an employee such as name, salary, hire date, and so on. This data is dissimilar in type but logically related. A record that contains such fields as the name, salary, and hire date of an employee would let you treat the data as a logical unit.

Declaring Records

Records must be declared in two steps. First, you define a RECORD type, then declare user-defined records of that type. You can declare RECORD types in the declarative part of any block, subprogram, or package using the syntax

```
    TYPE type_name IS RECORD
         (field_name1 {field_type | variable%TYPE | table.column%TYPE
              | table%ROWTYPE'D [NOT NULL],
          field_name2 {field_type | variable%TYPE | table.column%TYPE
              | table%ROWTYPE'D [NOT NULL],
          ...);
```

where "type_name" is a type specifier used in subsequent declarations of records and "field_type" is any datatype. You can use the %TYPE or %ROWTYPE attribute to specify a field datatype. In the following example, you declare a RECORD type named "DeptRecTyp":

```
    DECLARE
        TYPE DeptRecTyp IS RECORD
            (deptno  NUMBER(2) NOT NULL := 20,
             dname   dept.dname%TYPE,
             loc     dept.loc%TYPE);
        ...
```

Notice that the field declarations are like variable declarations. Each field has a unique name and specific datatype. You can add the NOT NULL constraint to any field declaration and so prevent the assigning of nulls to that field. Remember, fields declared as NOT NULL must be initialized.

Once you define type "DeptRecTyp," you can declare records of that type, as follows:

```
    dept_rec  DeptRecTyp;
```

The identifier "dept_rec" represents an entire record.

A record can be initialized in its declaration, as this example shows:

```
    DECLARE
        TYPE TimeTyp IS RECORD
            (second  SMALLINT := 0,
             minute  SMALLINT := 0,
             hour    SMALLINT := 0);
        ...
```

When you declare a record of type "TimeTyp," its three fields assume an initial value of zero.

Like scalar variables, user-defined records can be declared as the formal parameters of procedures and functions.   A packaged example follows:

```
    PACKAGE emp_actions IS
        TYPE EmpRecTyp IS RECORD
            (empno     NUMBER(4) NOT NULL := 1001,
             ename     CHAR(10),
             job       CHAR(14),
             mgr       NUMBER(4),
             hiredate  DATE
             sal       NUMBER(7,2),
             comm      NUMBER(7,2),
             deptno    NUMBER(4));
        ...
        PROCEDURE hire_employee (emp_rec EmpRecTyp);
        ...
    END emp_actions;
```

Referencing Records

To reference individual fields in a record, you use dot notation and the following syntax:

```
    record_name.field_name
```

For example, you reference the "ename" field in the "emp_rec" record as follows:

```
    emp_rec.ename ...
```

You can assign the value of a PL/SQL expression to a specific field by using the following syntax:

```
    record_name.field_name := plsql_expression;
```

In the next example, you convert an employee name to upper case:

```
    emp_rec.ename := UPPER(emp_rec.ename);
```

Instead of assigning values separately to each field in a record, you can assign values to all fields at once.   This can be done in two ways. First, you can assign one record to another if they belong to the same datatype.   For example, given the declarations

```
    DECLARE
```

```
        TYPE  DeptRecTyp IS RECORD (...);
        dept_rec1  DeptRecTyp;
        dept_rec2  DeptRecTyp;
        ...
```

the following assignment is legal:

```
    BEGIN
        ...
        dept_rec1 := dept_rec2;
```

Second, you can assign a list of column values to a record by using the SELECT or FETCH statement, as the example below shows.   Just make sure the column names appear in the same order as the fields in your record.

```
    DECLARE
        TYPE DeptRecTyp IS RECORD
            (deptno  NUMBER(2),
             dname   CHAR(14),
             loc     CHAR(13));
        dept_rec  DeptRecTyp;
        ...
    BEGIN
        SELECT deptno, dname, loc INTO dept_rec FROM dept
            WHERE deptno = 30;
        ...
    END;
```

Even if their fields match exactly, records of different types cannot be assigned to each other.   Furthermore, a user-defined record and a %ROWTYPE record always belong to different types, as the following example shows:

```
    DECLARE
        TYPE DeptRecTyp IS RECORD
            (deptno  NUMBER(2),
             dname   CHAR(14),
             loc     CHAR(13));
        dept_rec1  DeptRecTyp;
        dept_rec2  dept%ROWTYPE;
        ...
    BEGIN
        ...
        dept_rec1 := dept_rec2;  -- illegal
```

Also, records cannot be tested for equality or inequality.   For instance, the following IF condition is illegal:

```
    IF dept_rec1 = dept_rec2 THEN ...  -- illegal
```

Nesting Records

PL/SQL lets you declare and reference nested records.   That is, a record can be the component of another record, as this example shows:

```
DECLARE
    TYPE TimeTyp IS RECORD
        (minute  SMALLINT,
         hour    SMALLINT);
    TYPE MeetingTyp IS RECORD
        (day      DATE,
         time     TimeTyp,     -- nested record
         place    CHAR(20),
         purpose  CHAR(50));
    TYPE PartyTyp IS RECORD
        (day   DATE,
         time  TimeTyp,        -- nested record
         loc   CHAR(15));
    meeting  MeetingTyp;
    seminar  MeetingTyp;
    party    PartyTyp;
    ...
BEGIN
    meeting.day := '26-JUN-91';
    meeting.time.minute := 45;
    meeting.time.hour := 10;
    ...
END;
```

The next example shows that you can assign one nested record to another if they belong to the same datatype:

```
seminar.time := meeting.time;  -- legal
```

Such assignments are allowed even if the containing records belong to different datatypes, as the following example shows:

```
party.time := meeting.time;  -- legal
```

See also: Datatypes, Variables and Constants, PL/SQL Tables, %TYPE Attribute, %ROWTYPE Attribute

Structure Theorem

In PL/SQL, statements are connected by simple but powerful control structures that have a single entry and exit point. Collectively, these structures can handle any situation. And, their proper use leads naturally to a well-structured program.

According to the structure theorem, any computer program can be written using the three basic control structures flow-charted below. These structures can be combined in any way necessary to deal with a given problem.

```
           SELECTION                  ITERATION          SEQUENCE

               |                          |                 |
               |                          |                 |
          T   / \   F                    / \   F         +-----+
       +----<     >----+         +----><     >----+      |     |
       |      \ /      |         |       \ /      |      +-----+
       |               |         |        | T     |         |
    +-----+         +-----+      |        |       |         |
    |     |         |     |      |     +-----+    |      +-----+
    +-----+         +-----+      |     |     |    |      |     |
       |               |         |     +-----+    |      +-----+
       |               |         |        |       |         |
       +------( )------+         +--------+       |         |
               |                                  |         |
```

The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a Boolean value (TRUE, FALSE, or NULL). The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

See also: Condition, IF Statement, LOOP Statement, GOTO Statement, NULL Statement

IF Statement

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-THEN

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF), as follows:

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

The sequence of statements is executed only if the condition evaluates to TRUE. If the condition evaluates to FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement. An example follows:

```
IF sales > quota THEN
    compute_bonus(empid);
    UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
END IF;
```

IF-THEN-ELSE

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as follows:

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

The sequence of statements in the ELSE clause is executed only if the condition evaluates to FALSE or NULL. Thus, the ELSE clause ensures that a sequence of statements is executed. In the following example, the first or second UPDATE statement is executed when the condition is true or false, respectively:

```
IF trans_type = 'CR' THEN
    UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
    UPDATE accounts SET balance = balance - debit WHERE ...
END IF;
```

The THEN and ELSE clauses can include IF statements. That is, IF statements can be nested, as the following example shows:

```
IF trans_type = 'CR' THEN
    UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
    IF new_balance >= minimum_balance THEN
        UPDATE accounts SET balance = balance - debit WHERE ...
    ELSE
        RAISE insufficient_funds;
    END IF;
END IF;
```

IF-THEN-ELSIF

Sometimes you want to select an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSIF (not ELSEIF) to introduce additional conditions, as follows:

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

If the first condition evaluates to FALSE or NULL, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition evaluates to TRUE, its associated sequence of statements is executed and control passes to the next statement. If all conditions evaluate to FALSE or NULL, the sequence in the ELSE clause is executed. Consider the following example:

```
IF sales > 50000 THEN
    bonus := 1500;
ELSIF sales > 35000 THEN
    bonus := 500;
ELSE
    bonus := 100;
END IF;
INSERT INTO payroll VALUES (emp_id, bonus, ...);
```

If the value of "sales" is more than 50000, the first and second conditions are true. Nevertheless, "bonus" is assigned the proper value of 1500 because the second condition is never tested. When the first condition evaluates to TRUE, its associated statement is executed and control passes to the INSERT statement.

When possible, use the ELSIF clause instead of nested IF statements. That way, your code will be easier to read and understand. Compare the following IF statements:

```
    IF condition1 THEN                |     IF condition1 THEN
        statement1;                   |         statement1;
    ELSE                              |     ELSIF condition2 THEN
        IF condition2 THEN            |         statement2;
            statement2;               |     ELSIF condition3 THEN
        ELSE                          |         statement3;
            IF condition3 THEN        |     END IF;
                statement3;           |
            END IF;                   |
        END IF;                       |
    END IF;                           |
```

These statements are logically equivalent, but the first statement obscures the flow of logic, whereas the second statement reveals it.

See also: Condition, Structure Theorem, Boolean Expressions

LOOP Statement

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

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 the EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop.

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;
```

Before each iteration of the loop, the condition is evaluated. If the condition evaluates to TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If the condition evaluates to FALSE or NULL, the loop is bypassed and control passes to the next statement. An example follows:

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

The number of iterations depends on the condition and is unknown until the loop completes. Since the condition is tested at the top of the loop, the sequence might execute zero times. In the last example, if the initial value of "total" is greater than 25000, the condition evaluates to 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. The range is part of an "iteration scheme," which is enclosed by the keywords FOR and LOOP. The syntax follows:

```
FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
    sequence_of_statements;
END LOOP;
```

The range is evaluated when the FOR loop is first entered and is never reevaluated. 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;
```

If the lower bound equals the higher bound, the sequence of statements is executed once. If the lower bound is larger than the upper bound, the sequence of statements is not executed and control passes to the next statement.

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

Nevertheless, you write the range bounds in ascending (not descending) order.

The bounds of a loop range can be literals, variables, or expressions, but must evaluate to integers. For example, the following iteration schemes are legal:

```
j IN -5..5
k IN REVERSE first..last
step IN 0..TRUNC(high/low) * 2
code IN ASCII('A')..ASCII('J')
```

As you can see, the lower bound need not be 1. However, the loop counter increment (or decrement) must be 1.

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.

Inside a FOR loop, the loop counter can be referenced like a constant. So, the loop counter can appear in expressions but cannot be assigned values, as the following example shows:

```
    FOR ctr IN 1..10 LOOP
        ...
        IF NOT finished THEN
            INSERT INTO ... VALUES (ctr, ...);  -- legal
            factor := ctr * 2;  -- legal
            ...
        ELSE
            ctr := 10;  -- illegal
        END IF;
    END LOOP;
```

The loop counter is defined only within the loop.   You cannot reference it outside the loop.   After the loop is exited, the loop counter is undefined, as the following example shows:

```
    FOR ctr IN 1..10 LOOP
        ...
    END LOOP;
    sum := ctr - 1;  -- illegal
```

You need not declare the loop counter because it is declared implicitly as a local variable of type INTEGER.

See also: Structure Theorem, Cursor FOR Loop, EXIT Statement,
          Loop Label

EXIT Statement

You use the EXIT statement to complete a loop when further processing is undesirable or impossible. 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.

EXIT

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement. An example follows:

```
LOOP
    ...
    IF ... THEN
        ...
        EXIT;  -- exit loop immediately
    END IF;
END LOOP;
-- control resumes here
```

The next example shows that you cannot use the EXIT statement to complete a PL/SQL block:

```
BEGIN
    ...
    IF ... THEN
        ...
        EXIT;  -- illegal
    END IF;
END;
```

Remember, the EXIT statement must be placed inside a loop. To complete a PL/SQL block before the normal end of the block is reached, you can use the RETURN statement.

EXIT-WHEN

The EXIT-WHEN statement allows a loop to complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition evaluates to TRUE, the loop completes and control passes to the next statement after the loop. An example follows:

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

Until the condition evaluates to TRUE, the loop cannot complete. So, statements within the loop must change the value of the condition. In the last example, if the FETCH statement returns a row, the condition evaluates to FALSE. When the FETCH statement fails to return a row, the condition evaluates to TRUE, the loop completes, and control passes to the CLOSE statement.

The EXIT-WHEN statement allows a FOR loop to complete prematurely. For example, the following loop normally executes ten times, but as soon as the FETCH 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;
```

See also: Condition, LOOP Statement, Cursor FOR Loop, Loop Label

Loop Label

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.

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
```

See also: EXIT Statement, LOOP Statement, Block Label

GOTO Statement

The structure of PL/SQL is such that the GOTO statement is seldom needed. But, occasionally, it can simplify logic enough to warrant its use.

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. In the following example, you go to an executable statement farther down in a sequence of statements:

```
BEGIN
    ...
    GOTO insert_row;
    ...
    <<insert_row>>
    INSERT INTO emp VALUES ...
END;
```

In the next example, you go to a PL/SQL block farther up in a sequence of statements:

```
BEGIN
    ...
    <<update_row>>
    BEGIN
        UPDATE emp SET ...
        ...
    END;
    ...
    GOTO update_row;
    ...
END;
```

The label <<end_loop>> in the following example is illegal because it does not precede an executable statement:

```
DECLARE
    done  BOOLEAN;
BEGIN
    ...
    FOR i IN 1..50 LOOP
        IF done THEN
            GOTO end_loop;
        END IF;
        ...
    <<end_loop>>  -- illegal
    END LOOP;  -- not an executable statement
END;
```

To debug the last example, simply add the NULL statement, as follows:

```
DECLARE
    done  BOOLEAN;
BEGIN
    ...
    FOR i IN 1..50 LOOP
        IF done THEN
            GOTO end_loop;
        END IF;
        ...
    <<end_loop>>
    NULL;  -- an executable statement
    END LOOP;
END;
```

As the following example shows, a GOTO statement can branch to an enclosing block from the current block:

```
DECLARE
    my_ename  CHAR(10);
BEGIN
    ...
    <<get_name>>
    SELECT ename INTO my_ename FROM emp WHERE ...
    ...
    BEGIN
        ...
        GOTO get_name;  -- branch to enclosing block
    END;
END;
```

The GOTO statement branches to the first enclosing block in which the referenced label appears.

However, some possible destinations of a GOTO statement are illegal. Specifically, a GOTO statement cannot branch into an IF statement, LOOP statement, or sub-block.   For example, the following GOTO statement is illegal:

```
BEGIN
    ...
    GOTO update_row;  -- illegal branch into IF statement
    ...
    IF valid THEN
        ...
        <<update_row>>
        UPDATE emp SET ...
    END IF;
END;
```

Also, a GOTO statement cannot branch from one IF statement clause to another, as the following example shows:

```
BEGIN
```

```
    ...
    IF valid THEN
        ...
        GOTO update_row;  -- illegal branch into ELSE clause
    ELSE
        ...
        <<update_row>>
        UPDATE emp SET ...
    END IF;
END;
```

The next example shows that a GOTO statement cannot branch from an enclosing block into a sub-block:

```
BEGIN
    ...
    IF status = 'OBSOLETE' THEN
        GOTO delete_part;  -- illegal branch into sub-block
    END IF;
    ...
    BEGIN
        ...
        <<delete_part>>
        DELETE FROM parts WHERE ...
    END;
END;
```

Also, a GOTO statement cannot branch out of a subprogram, as the following example shows:

```
DECLARE
    ...
    PROCEDURE compute_bonus (emp_id NUMBER) IS
    BEGIN
        ...
        GOTO update_row;  -- illegal branch out of subprogram
    END;
BEGIN
    ...
    <<update_row>>
    UPDATE emp SET ...
END;
```

Finally, a GOTO statement cannot branch from an exception handler into the current block. For example, the following GOTO statement is illegal:

```
DECLARE
    ...
    pe_ratio  REAL;
BEGIN
    ...
    SELECT price / NVL(earnings, 0) INTO pe_ratio FROM ...
```

```
        <<insert_row>>
        INSERT INTO stats VALUES (pe_ratio, ...);
    EXCEPTION
        WHEN ZERO_DIVIDE THEN
            pe_ratio := 0;
            GOTO insert_row;  -- illegal branch into current block
        ...
    END;
```

However, a GOTO statement can branch from an exception handler into an enclosing block.

Overuse of GOTO statements can result in complex, unstructured code that is hard to understand and maintain.   So, use GOTO statements sparingly.

See also: Structure Theorem, EXIT Statement, Block Label

NULL Statement

The NULL statement explicitly specifies inaction; it does nothing other than pass control to the next statement. However, it can make the meaning and action of conditional statements clear and so 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 meets this requirement. So, you can use the NULL statement 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 receive 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. Stubs are dummy subprograms that allow you to defer the definition of procedures and functions until you test and debug the main program. In the next 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;
```

See also: IF Statement, Subprograms

SQL Support

By extending SQL, PL/SQL offers a unique combination of power and ease of use.   You can manipulate Oracle data flexibly and safely because PL/SQL supports all SQL data manipulation statements (except EXPLAIN PLAN), transaction control statements, functions, pseudocolumns, and operators.

PL/SQL does not allow SQL data definition statements such as CREATE, session control statements such as SET ROLE, or the system control statement ALTER SYSTEM.   However, a package named DBMS_SQL, which is supplied with the Oracle Server, allows PL/SQL to execute SQL data definition and data manipulation statements dynamically at run time.

SQL Functions

PL/SQL lets you use all the SQL functions including the following group functions, which summarize entire columns of Oracle data:

```
    * AVG
    * COUNT
    * MAX
    * MIN
    * STDDEV
    * SUM
    * VARIANCE
```

You can use the group functions in SQL statements, but not in procedural statements.   Group functions operate on entire columns unless you use the SELECT GROUP BY statement to sort returned rows into subgroups. If you omit the GROUP BY clause, the group function treats all returned rows as a single group.

You call a group function using the syntax

```
    function_name([ALL | DISTINCT] expr)
```

where "expr" is an expression that refers to one or more database columns.   If you specify the ALL option (the default), the group function considers all column values including duplicates.   For example, the following statement returns the sample standard deviation of all values in the "comm" column:

```
    SELECT STDDEV(comm) INTO comm_sigma FROM emp;
```

If you specify the DISTINCT option, the group function considers only distinct values.   For example, the following statement returns the number of different job titles in the "emp" table:

```
    SELECT COUNT(DISTINCT job) INTO job_count FROM emp;
```

The COUNT function lets you specify the asterisk (*) option, which returns the number of rows in a table.   For example, the following statement returns the number of employees in the "emp" table:

```
    SELECT COUNT(*) INTO emp_count FROM emp;
```

Except for COUNT(*), all group functions ignore nulls.

SQL Pseudocolumns

PL/SQL recognizes the following SQL pseudocolumns, which return specific data items:

```
    * CURRVAL
    * LEVEL
    * NEXTVAL
    * ROWID
    * ROWNUM
```

For example, NEXTVAL returns the next value in a database sequence. They are called "pseudocolumns" because they are not actual columns in a table but behave like columns. For instance, you can reference pseudocolumns in SQL statements. Furthermore, you can select values from a pseudocolumn. However, you cannot insert values into, update values in, or delete values from a pseudocolumn.

You can use pseudocolumns in SQL statements, but not in procedural statements. In the following example, you use the database sequence "empno_seq" and the pseudocolumn NEXTVAL to insert a new employee number into the "emp" table:

```
    INSERT INTO emp VALUES (empno_seq.NEXTVAL, new_ename, ...);
```

Sometimes, it is convenient to select pseudocolumn values from a dummy table, as follows:

```
    DECLARE
        new_empno  NUMBER(4);
        new_ename  CHAR(10);
        ...
    BEGIN
        ...
        SELECT empno_seq.NEXTVAL INTO new_empno FROM dual;
        INSERT INTO emp VALUES (new_empno, new_ename, ...);
    END;
```

SQL Operators

PL/SQL lets you use all the SQL comparison, set, and row operators in SQL statements. Typically, you use comparison operators in the WHERE clause of a data manipulation statement to form "predicates," which compare one expression to another and always evaluate to TRUE, FALSE, or NULL. You can use all the comparison operators listed below to form predicates. Moreover, you can combine predicates using the logical operators AND, OR, and NOT.

ALL	Compares a value to each value in a list or returned by a subquery and evaluates to TRUE if all of the individual comparisons yield a TRUE result.
ANY, SOME	Compares a value to each value in a list or returned by a subquery and evaluates to TRUE if any of the individual comparisons yields a TRUE result.
BETWEEN	Tests whether a value lies in a specified range.
EXISTS	Returns TRUE if a subquery returns at least one row.
IN	Tests for set membership.
IS NULL	Tests for nulls.
LIKE	Tests whether a character string matches a specified pattern, which can include wildcards.

Set operators combine the results of two queries into one result. You can use all the set operators, including INTERSECT, MINUS, UNION, and UNION ALL.

INTERSECT	Returns all distinct rows selected by both queries.
MINUS	Returns all distinct rows selected by the first query but not by the second.
UNION	Returns all distinct rows selected by either query.
UNION ALL	Returns all rows selected by either query, including all duplicates.

Row operators return or reference particular rows. You can use all the row operators, including ALL, DISTINCT, and PRIOR.

ALL	Retains duplicate rows in the result of a query or in an aggregate expression.
DISTINCT	Eliminates duplicate rows from the result of a query or from an aggregate expression.
PRIOR	Refers to the parent row of the current row returned by a tree-structured query. You must use this operator in the CONNECT BY clause of such a query to define the parent-child relationship.

See also: SQL Commands, SQL Functions, Operators, Pseudocolumns, Product-specific Packages

Optimizer Hints

For every SQL statement, the Oracle optimizer generates an execution plan, which is a series of steps that Oracle takes to execute the statement.   In some cases, you can suggest to Oracle the way to optimize a SQL statement.   These suggestions, called "hints," let you influence decisions made by the optimizer.

Hints are not directives; they merely help the optimizer do its job. Some hints limit the scope of information used to optimize a SQL statement, while others suggest overall strategies.   You can use hints to specify the

```
    * optimization approach for a SQL statement
    * access path for each referenced table
    * join order for a join
    * method used to join tables
```

Hence, hints fall into the following four categories:

```
    * Optimization Approach
    * Access Path
    * Join Order
    * Join Operation
```

For example, the two Optimization Approach hints COST and NOCOST tell the optimizer to take a cost-based or rule-based approach, respectively. You give hints to the optimizer by placing them in a comment immediately after the verb in a SELECT, UPDATE, or DELETE statement.   For instance, the optimizer uses the cost-based approach for the following statement:

```
    SELECT /*+ COST */ ename, job, sal INTO ...
```

See also: SQL Support, Comments (SQL), SELECT, UPDATE, DELETE

National Language Support

Although the widely-used 7- or 8-bit ASCII and EBCDIC character sets are adequate to represent the Roman alphabet, some Asian languages, such as Japanese, contain thousands of characters. These languages require 16 bits (two bytes) to represent each character. How does Oracle deal with such dissimilar languages?

Oracle provides National Language Support (NLS), which lets you process single-byte and multibyte character data and convert between character sets. It also lets your applications run in different language environments. With NLS, number and date formats adapt automatically to the language conventions specified for a user session. Thus, NLS allows users around the world to interact with Oracle in their native languages.

You control the operation of language-dependent features by specifying various NLS parameters. Default values for these parameters can be set in the Oracle initialization file. The following table shows what each NLS parameter specifies.

NLS Parameter	Specifies
NLS_LANGUAGE	language-dependent conventions
NLS_TERRITORY	territory-dependent conventions
NLS_DATE_FORMAT	date format
NLS_DATE_LANGUAGE	language for day and month names
NLS_NUMERIC_CHARACTERS	decimal character and group separator
NLS_CURRENCY	local currency symbol
NLS_ISO_CURRENCY	ISO currency symbol
NLS_SORT	sort sequence

The main parameters are NLS_LANGUAGE and NLS_TERRITORY. NLS_LANGUAGE specifies the default values for language-dependent features, which include

* language for Oracle Server messages
* language for day and month names
* sort sequence

NLS_LANGUAGE specifies the default values for territory-dependent features, which include

* date format
* decimal character
* group separator
* local currency symbol
* ISO currency symbol

You can control the operation of language-dependent NLS features for a user session by specifying the parameter NLS_LANG as follows

```
NLS_LANG = <language>_<territory>.<character set>
```

where "language" specifies the value of NLS_LANGUAGE for the user session, "territory" specifies the value of NLS_TERRITORY, and "character set" specifies the encoding scheme used for the terminal. An encoding scheme (usually called a character set or code page) is a range of numeric codes that corresponds to the set of characters a terminal can display.   It also includes codes that control communication with the terminal.

You define NLS_LANG as an environment variable (or the equivalent on your system).   For example, on UNIX using the C shell, you might define NLS_LANG as follows:

```
    define NLS_LANG French_France.WE8DEC
```

PL/SQL fully supports all the NLS features that allow your applications to process multilingual data stored in an Oracle database.   For example, you can declare foreign-language character variables and pass them to string functions such as INSTRB, LENGTHB, and SUBSTRB.   These functions have the same syntax as the INSTR, LENGTH, and SUBSTR functions, respectively, but operate on a per-byte basis rather than a per-character basis.

You can use the functions NLS_INITCAP, NLS_LOWER, and NLS_UPPER to handle special instances of case conversion.   And, you can use the function NLSSORT to specify WHERE-clause comparisons based on linguistic rather than binary ordering.   You can even pass NLS parameters to the TO_CHAR, TO_DATE, and TO_NUMBER functions.

See also: SQL Support, ALTER SESSION

Remote Access

PL/SQL lets you access remote databases.   In the example below, you query a remote database table in the "newyork" database via SQL*Net. The query is submitted to your local Oracle Server, but is forwarded to the remote database for execution.

```
    BEGIN
        SELECT ename, job INTO my_ename, my_job
            FROM emp@newyork
            WHERE empno = my_empno;
        ...
```

You can use the %TYPE attribute to provide the datatype of a column in a remote database table.   Likewise, you can use the %ROWTYPE attribute to declare a record that represents a row in a remote table.   Some examples follow:

```
    DECLARE
        emp_id    emp.empno@newyork%TYPE;
        dept_rec  dept@newyork%ROWTYPE;
        ...
```

You can even call standalone and packaged subprograms stored in a remote Oracle database.

See also: Naming Conventions, %TYPE Attribute, %ROWTYPE Attribute, Global Names, Location Transparency, CREATE DATABASE LINK, CREATE SYNONYM

Location Transparency

To provide location transparency for remote database objects such as tables and views, you can create synonyms.   In the following example, you create a synonym for the "emp" table in the "newyork" database:

```
    CREATE SYNONYM emp2 FOR emp@newyork;
```

Then, as the next example shows, you can reference a remote object using its synonym:

```
    BEGIN
        SELECT ename, job INTO my_ename, my_job
            FROM emp2  -- synonym for emp@newyork
            WHERE empno = my_empno;
        ...
```

Because they are database objects, you can also create synonyms for packages and standalone subprograms.   However, you cannot create synonyms for objects declared within packages or subprograms.   For example, you cannot create synonyms for packaged procedures.

See also: Naming Conventions, Global Names, Remote Access,
          CREATE DATABASE LINK, CREATE SYNONYM

Global Names

A distributed database is a single logical database comprising multiple physical databases at different nodes. In a distributed database system, each database is uniquely identified by a two-part global name. The first part is a database name such as "sales"; the second part is a network domain name. The figure below shows a fictional network domain that follows Internet conventions. Internet is a worldwide system of computer networks used by companies and schools to exchange information.

```
                     com
                   /  |  \
                 /    |    \
                    primus
                  /        \
                /            \
             asia        americas
           /  |           /      \
         /    |         /          \
            japan    mexico         us
              |        |          /    \
              |        |        /        \
            sales     mfg    sales      acctg
```

To form network domain names, you use dot notation and follow a path through the tree structure from leaf to root. The domain name tells you the name or location of a network host and the type of organization it is. In the following example, the trailing domain "com" tells you that Primus is a company or other commercial institution:

```
    mfg.mexico.americas.primus.com
```

Every database object is uniquely identified by its global object name. In the following example, you query a remote "emp" table:

```
    BEGIN
        SELECT ename, job INTO my_ename, my_job
            FROM emp@sales.japan.asia.primus.com
            WHERE empno = my_empno;
        ...
```

See also: Naming Conventions, Location Transparency, Remote Access

Cursors

Oracle uses work areas called "private SQL areas" to execute SQL statements and store processing information. A PL/SQL construct called a "cursor" lets you name a private SQL area and access its stored information. There are two kinds of cursors: implicit and explicit.

PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return more than one row, you can explicitly declare a cursor to process the rows individually.

Explicit Cursors

The set of rows returned by a multirow query is called the "active set." Its size is the number of rows that meet your search criteria. An explicit cursor points to the current row in the active set. This allows your program to process the rows one at a time.

Multirow query processing is somewhat like file processing. For example, a COBOL program opens a file to process records, then closes the file. Likewise, a PL/SQL program opens a cursor to process rows returned by a query, then closes the cursor. Just as a file pointer marks the current position in an open file, a cursor marks the current position in an active set.

You use three commands to control the cursor: OPEN, FETCH, and CLOSE. First, you initialize the cursor with the OPEN statement, which identifies the active 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.

Forward references are not allowed in PL/SQL. So, you must declare a cursor before referencing it in other statements. You define a cursor in the declarative part of a PL/SQL block, subprogram, or package by naming it and specifying a query. In the following example, you declare a cursor named "c1":

```
DECLARE
    CURSOR c1 IS SELECT ename, deptno FROM emp WHERE sal > 2000;
    ...
```

The cursor name is an undeclared identifier, not a PL/SQL variable; it is used only to reference the query. You cannot assign values to a cursor name or use it in an expression.

Explicit cursors can take parameters, as the example below shows. A cursor parameter can appear in a query wherever a constant can appear.

```
CURSOR c1 (median IN NUMBER) IS
    SELECT job, ename FROM emp WHERE sal > median;
```

To declare formal cursor parameters, you use the syntax

```
    CURSOR name [ (parameter [, parameter, ...]) ] IS
        SELECT ...
```

where "parameter" stands for the following syntax:

```
    variable_name [IN] datatype [{:= | DEFAULT'D value]
```

The formal parameters of a cursor must be IN parameters. As the example below shows, you can initialize cursor parameters to default values. That way, you can pass different numbers of actual parameters to a cursor, accepting or overriding the default values as you please. Moreover, you can add new formal parameters without having to change every reference to the cursor.

```
    DECLARE
        CURSOR c1
            (low  INTEGER DEFAULT 0,
             high INTEGER DEFAULT 99) IS SELECT ...
```

The scope of cursor parameters is local to the cursor, meaning that they can be referenced only within the query used in the cursor declaration. The values of cursor parameters are used by the associated query when the cursor is OPENed.

Implicit Cursors

Oracle implicitly opens a cursor to process each SQL statement not associated with an explicitly declared cursor. PL/SQL lets you refer to the most recent implicit cursor as the "SQL" cursor. So, although you cannot use the OPEN, FETCH, and CLOSE statements to control an implicit cursor, you can still use cursor attributes to access information about the most recently executed SQL statement.

The values of cursor attributes always refer to the most recently executed SQL statement, wherever that statement appears. It might be in a different scope (for example, in a sub-block). So, if you want to save an attribute value for later use, assign it to a Boolean variable immediately. The following example shows how failing to save an attribute value can result in a logic bug:

```
    UPDATE parts SET qty = qty - 1 WHERE partno = part_id;
    check_parts;  -- procedure call
    IF SQL%NOTFOUND THEN  -- dangerous!
       ...
    END IF;
```

In this example, it is dangerous to rely on the IF condition because the procedure "check_parts" might have changed the value of %NOTFOUND. You can debug the code as follows:

```
    UPDATE parts SET qty = qty - 1 WHERE partno = part_id;
    sql_notfound := SQL%NOTFOUND;
    check_parts;
    IF sql_notfound THEN
       ...
    END IF;
```

Before Oracle opens the SQL cursor, the implicit cursor attributes evaluate to NULL.

```
See also: OPEN Statement, FETCH Statement, CLOSE Statement,
          %NOTFOUND Attribute, %FOUND Attribute, %ROWCOUNT Attribute,
          %ISOPEN Attribute, Packaged Cursors, Cursor FOR Loop
```

OPEN Statement

The OPEN statement executes the query associated with an explicitly declared cursor.   OPENing the cursor executes the query and identifies the active set, which consists of all rows that meet the query search criteria.   For cursors declared using the FOR UPDATE Clause, the OPEN statement also locks those rows.   An example of the OPEN statement follows:

```
    OPEN c1;
```

Rows in the active set are not retrieved when the OPEN statement is executed.   Rather, the FETCH statement retrieves the rows.

Passing Parameters

You can pass parameters to a cursor.   For example, given the cursor declaration

```
    CURSOR c1 (my_ename CHAR, my_comm NUMBER) IS SELECT ...
```

any of the following statements opens the cursor:

```
    OPEN c1('ATTLEY', 300);
    OPEN c1(employee_name, 150);
    OPEN c1('THURSTON', my_comm);
```

In the last example, the variable referenced in the OPEN statement has the same name as the parameter in the cursor declaration.   When "my_comm" is used in the cursor declaration, it refers to the formal parameter "my_comm."   When it is used outside the declaration, it refers to the PL/SQL variable "my_comm."   For clarity, use unique identifiers.

Unless you want to accept default values, each formal parameter in the cursor declaration must have a corresponding actual parameter in the OPEN statement.   Formal parameters declared with a default value need not have a corresponding actual parameter.   They can simply assume their default values when the OPEN statement is executed.

The formal parameters of a cursor must be IN parameters.   Therefore, they cannot return values to actual parameters.   Each actual parameter must belong to a datatype compatible with the datatype of its corresponding formal parameter.

See also: Cursors, FETCH Statement, CLOSE Statement

FETCH Statement

The FETCH statement retrieves the rows in the active set one at a time. Each time FETCH is executed, the cursor advances to the next row in the active set. An example of the FETCH statement follows:

```
    FETCH c1 INTO my_empno, my_ename, my_deptno;
```

For each column value returned by the query associated with the cursor, there must be a corresponding variable in the INTO list. Also, their datatypes must be compatible. Typically, you use FETCH as follows:

```
    OPEN c1;
    LOOP
        FETCH c1 INTO my_record;
        EXIT WHEN c1%NOTFOUND;
        -- process retrieved data
    END LOOP;
```

Any variables in the WHERE clause of the query associated with the cursor are evaluated only when the cursor is OPENed. As the following example shows, the query can reference PL/SQL variables within its scope:

```
    DECLARE
        my_sal  emp.sal%TYPE;
        my_job  emp.job%TYPE;
        factor  INTEGER := 2;
        CURSOR c1 IS
            SELECT factor*sal FROM emp WHERE job = my_job;
    BEGIN
        ...
        OPEN c1;  -- here factor equals 2
        LOOP
            FETCH c1 INTO my_sal;
            EXIT WHEN c1%NOTFOUND;
            ...
            factor := factor + 1;  -- does not affect FETCH
        END LOOP;
    END;
```

In this example, each retrieved salary is multiplied by 2, even though "factor" is incremented after each FETCH. To change the active set or the values of variables in the query, you must CLOSE and reOPEN the cursor with the input variables set to their new values.

However, you can use a different INTO list on separate FETCHes with the same cursor. Each FETCH retrieves another row and assigns values to the INTO variables, as the following example shows:

```
    DECLARE
        CURSOR c1 IS SELECT ename FROM emp;
        name1  emp.ename%TYPE;
        name2  emp.ename%TYPE;
        name3  emp.ename%TYPE;
    BEGIN
        OPEN c1;
        FETCH c1 INTO name1;  -- this fetches first row
        FETCH c1 INTO name2;  -- this fetches second row
        FETCH c1 INTO name3;  -- this fetches third row
        ...
        CLOSE c1;
    END;
```

If you execute a FETCH but there are no more rows left in the active set, the values of the fetch-list variables are indeterminate.

See also: Cursors, OPEN Statement, CLOSE Statement

CLOSE Statement

The CLOSE statement disables the cursor, and the active set becomes undefined.   An example of the CLOSE statement follows:

```
    CLOSE c1;
```

Once a cursor is CLOSEd, you can reOPEN it.   Any other operation on a closed cursor raises the predefined exception INVALID_CURSOR.

See also: Cursors, OPEN Statement, FETCH Statement

%NOTFOUND Attribute

Each cursor that you explicitly define has four attributes: %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN. When appended to the cursor name, these attributes let you access useful information about the execution of a multirow query.

The SQL implicit cursor has the same four attributes. When appended to the cursor name (SQL), these attributes let you access information about the most recently executed INSERT, UPDATE, DELETE, or SELECT INTO statement.

You can use cursor attributes in procedural statements but not in SQL statements.

Explicit Cursors

When a cursor is OPENed, the rows that satisfy the associated query are identified and form the active set. Before the first fetch, %NOTFOUND evaluates to NULL. Rows are FETCHed from the active set one at a time. If the last fetch returned a row, %NOTFOUND evaluates to FALSE. If the last fetch failed to return a row (because the active set was empty), %NOTFOUND evaluates to TRUE. FETCH is expected to fail eventually, so when that happens, no exception is raised.

In the following example, you use %NOTFOUND to exit a loop when FETCH fails to return a row:

```
LOOP
    FETCH c1 INTO my_ename, my_deptno;
    EXIT WHEN c1%NOTFOUND;
    ...
END LOOP;
```

You can open multiple cursors, then use %NOTFOUND to tell which cursors have rows left to fetch. If a cursor is not open, referencing it with %NOTFOUND raises the predefined exception INVALID_CURSOR.

Implicit Cursors

%NOTFOUND evaluates to TRUE if an INSERT, UPDATE, or DELETE affected no rows or a SELECT INTO returned no rows. Otherwise, %NOTFOUND evaluates to FALSE. In the following example, you use %NOTFOUND to insert a new row if an update fails:

```
UPDATE emp SET sal = sal * 1.05 WHERE empno = my_empno;
IF SQL%NOTFOUND THEN  -- update failed
    INSERT INTO temp VALUES (...);
END IF;
```

If a SELECT INTO fails to return a row, the predefined exception NO_DATA_FOUND is raised whether you check %NOTFOUND on the next line or not. Consider the following example:

```
DECLARE
    my_sal  NUMBER(7,2);
    my_empno  NUMBER(4);
BEGIN
    ...
    SELECT sal INTO my_sal FROM emp WHERE empno = my_empno;
      -- might raise NO_DATA_FOUND
    IF SQL%NOTFOUND THEN  -- condition tested only when false
        ...  -- this action is never taken
    END IF;
EXCEPTION
    ...
END;
```

The check is useless because the IF condition is tested only when %NOTFOUND is false. When NO_DATA_FOUND is raised, normal execution stops and control transfers to the exception-handling part of the block. In this situation, %NOTFOUND is useful in the OTHERS exception handler, as the example below shows. Instead of coding a NO_DATA_FOUND handler, you find out if that exception was raised by checking %NOTFOUND.

```
DECLARE
    my_sal    NUMBER(7,2);
    my_empno  NUMBER(4);
BEGIN
    ...
    SELECT sal INTO my_sal FROM emp WHERE empno =my_empno;
      -- might raise NO_DATA_FOUND
EXCEPTION
    WHEN OTHERS THEN
        IF SQL%NOTFOUND THEN  -- check for 'no data found'
            ...
        END IF;
        ...
END;
```

However, a SELECT INTO that calls a SQL group function never raises the exception NO_DATA_FOUND. That is because group functions such as AVG and SUM always return a value or a null. In such cases, %NOTFOUND always evaluates to FALSE. Consider this example:

```
DECLARE
    my_sal     NUMBER(7,2);
    my_deptno  NUMBER(2);
BEGIN
    ...
    SELECT MAX(sal) INTO my_sal FROM emp WHERE deptno = my_deptno;
      -- never raises NO_DATA_FOUND
    IF SQL%NOTFOUND THEN  -- always tested but never true
        ...  -- this action is never taken
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN  -- never invoked
        ...
END;
```

See also: Cursors, %FOUND Attribute, %ROWCOUNT Attribute,
          %ISOPEN Attribute

%FOUND Attribute

Each cursor that you explicitly define has four attributes: %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN. When appended to the cursor name, these attributes let you access useful information about the execution of a multirow query.

The SQL implicit cursor has the same four attributes. When appended to the cursor name (SQL), these attributes let you access information about the most recently executed INSERT, UPDATE, DELETE, or SELECT INTO statement.

You can use cursor attributes in procedural statements but not in SQL statements.

Explicit Cursors

%FOUND is the logical opposite of %NOTFOUND. After an explicit cursor is open but before the first fetch, %FOUND evaluates to NULL. Thereafter, it evaluates to TRUE if the last fetch returned a row or to FALSE if no row was returned.

In the following example, you use %FOUND to select an action:

```
LOOP
    FETCH c1 INTO my_ename, my_deptno;
    IF c1%FOUND THEN  -- fetch succeeded
        INSERT INTO temp VALUES (...);
    ELSE  -- fetch failed, so exit loop
        EXIT;
    END IF;
    ...
END LOOP;
```

You can open multiple cursors, then use %FOUND to tell which cursors have rows left to fetch. If a cursor is not open, referencing it with %FOUND raises INVALID_CURSOR.

Implicit Cursors

%FOUND is the logical opposite of %NOTFOUND. Until a SQL data manipulation statement is executed, %FOUND evaluates to NULL. Thereafter, %FOUND evaluates to TRUE if an INSERT, UPDATE, or DELETE affected one or more rows or a SELECT INTO returned one or more rows. Otherwise, %FOUND evaluates to FALSE. In the following example, you use %FOUND to insert a row if a deletion succeeds:

```
DELETE FROM temp_emp WHERE empno = my_empno;
IF SQL%FOUND THEN  -- delete succeeded
    INSERT INTO emp VALUES (my_empno, my_ename, ...);
END IF;
```

See also: Cursors, %NOTFOUND Attribute, %ROWCOUNT Attribute,
%ISOPEN Attribute

%ROWCOUNT Attribute

Each cursor that you explicitly define has four attributes: %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN. When appended to the cursor name, these attributes let you access useful information about the execution of a multirow query.

The SQL implicit cursor has the same four attributes. When appended to the cursor name (SQL), these attributes let you access information about the most recently executed INSERT, UPDATE, DELETE, or SELECT INTO statement.

You can use cursor attributes in procedural statements but not in SQL statements.

Explicit Cursors

When you open its cursor, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT returns a zero. Thereafter, it returns the number of rows fetched so far. The number is incremented if the latest fetch returned a row. In the next example, you use %ROWCOUNT to take action if more than ten rows have been fetched:

```
LOOP
    FETCH c1 INTO my_ename, my_deptno;
    IF c1%ROWCOUNT > 10 THEN
        ...
    END IF;
    ...
END LOOP;
```

You can open multiple cursors, then use %ROWCOUNT to tell how many rows have been fetched so far. If a cursor is not open, referencing it with %ROWCOUNT raises INVALID_CURSOR.

Implicit Cursors

%ROWCOUNT returns the number of rows affected by an INSERT, UPDATE, or DELETE or returned by a SELECT INTO. %ROWCOUNT returns a zero if an INSERT, UPDATE, or DELETE affected no rows or a SELECT INTO returned no rows. In the following example, you use %ROWCOUNT to take action if more than ten rows have been deleted:

```
DELETE FROM emp WHERE ...
IF SQL%ROWCOUNT > 10 THEN
    ...
END IF;
```

If a SELECT INTO returns more than one row, the predefined exception TOO_MANY_ROWS is raised and %ROWCOUNT is set to 1, not the actual number of rows that satisfy the query. In this situation, %ROWCOUNT is useful in the OTHERS exception handler, as the example below shows.

Instead of coding a TOO_MANY_ROWS handler, you find out if the exception TOO_MANY_ROWS was raised by checking %ROWCOUNT.

```
DECLARE
    my_sal    NUMBER(7,2);
    my_ename  CHAR(10);
BEGIN
    ...
    SELECT sal INTO my_sal FROM emp WHERE ename = my_ename;
      -- might raise TOO_MANY_ROWS
    ...
EXCEPTION
    WHEN OTHERS THEN
        IF SQL%ROWCOUNT > 0 THEN  -- check for 'too many rows'
            ...
        END IF;
        ...
END;
```

See also: Cursors, %NOTFOUND Attribute, %FOUND Attribute,
          %ISOPEN Attribute

%ISOPEN Attribute

Each cursor that you explicitly define has four attributes: %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN. When appended to the cursor name, these attributes let you access useful information about the execution of a multirow query.

The SQL implicit cursor has the same four attributes. When appended to the cursor name (SQL), these attributes let you access information about the most recently executed INSERT, UPDATE, DELETE, or SELECT INTO statement.

You can use cursor attributes in procedural statements but not in SQL statements.

Explicit Cursors

%ISOPEN evaluates to TRUE if its cursor is open; otherwise, %ISOPEN evaluates to FALSE. In the following example, you use %ISOPEN to select an action:

```
IF c1%ISOPEN THEN  -- cursor is open
    ...
ELSE  -- cursor is closed, so open it
    OPEN c1;
END IF;
```

Implicit Cursors

Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %ISOPEN always evaluates to FALSE.

See also: Cursors, %NOTFOUND Attribute, %FOUND Attribute,
          %ROWCOUNT Attribute

Packaged Cursors

You can separate a cursor specification from its body for placement in a package by using the RETURN clause, as the following example shows:

```
CREATE PACKAGE emp_actions AS
    /* Declare cursor specification. */
    CURSOR c1 RETURN emp%ROWTYPE;
    ...
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
    /* Define cursor body. */
    CURSOR c1 RETURN emp%ROWTYPE
        SELECT * FROM emp WHERE sal > 3000;
    ...
END emp_actions;
```

That way, you can change the cursor body without changing the cursor specification. For instance, you might want to change the WHERE clause in the last example, as follows:

```
 CURSOR c1 RETURN emp%ROWTYPE
     SELECT * FROM emp WHERE deptno = 20;
```

A cursor specification has no SELECT statement because the RETURN clause defines the datatype of the result value. You can use the %ROWTYPE attribute in a RETURN clause to provide a record type that represents a row in a database table.

A cursor body must have a SELECT statement and the same RETURN clause as its corresponding cursor specification. Furthermore, the number and datatypes of select-items in the SELECT statement must match the RETURN clause.

See also: Cursors, Packages, CREATE PACKAGE, CREATE PACKAGE BODY

Cursor FOR Loop

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 active set into fields in the record, and closes the cursor when all rows have been processed.

In the example below, the FOR loop index "emp_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
    salary_total  REAL := 0.0;
    CURSOR c1 IS SELECT ename, sal, hiredate, deptno FROM emp;
    ...
BEGIN
    FOR emp_rec IN c1 LOOP
        ...
        salary_total :=  salary_total + emp_rec.sal;
    END LOOP;
    ...
END;
```

When the cursor FOR loop is entered, the cursor name cannot refer to a cursor that was already opened (by an OPEN statement or by an enclosing cursor FOR loop).

Before each iteration of the FOR loop, PL/SQL fetches into the implicitly declared record, which is equivalent to a record explicitly declared as follows:

```
emp_rec  c1%ROWTYPE;
```

The record is defined only inside the loop.   You cannot refer to its fields outside the loop.   For example, the following reference is illegal:

```
BEGIN
    FOR emp_rec IN c1 LOOP
        ...
    END LOOP;
    IF emp_rec.deptno = 20 THEN ...  -- illegal; outside loop
    ...
END;
```

The sequence of statements inside the loop is executed once for each row that satisfies the query associated with the cursor.   When you leave the loop, the cursor is closed automatically.   This is true even if you use an EXIT or GOTO statement to leave the loop prematurely or if an exception is raised inside the loop.

Using Aliases

Fields in the implicitly declared record hold column values from the most recently fetched row. The fields have the same names as corresponding columns in the query select list. But, what happens if a select-item is an expression? Consider the following example:

```
    CURSOR c1 IS
        SELECT empno, sal+NVL(comm,0), job FROM ...
```

In such cases, you must include an alias for the select-item. In the following example, "wages" is an alias for "sal+NVL(comm,0)":

```
    CURSOR c1 IS
        SELECT empno, sal+NVL(comm,0) wages, job FROM ...
```

To reference the corresponding field, you use the alias instead of a column name, as follows:

```
    IF emp_rec.wages < 1000 THEN ...
```

Passing Parameters

You can pass parameters to a cursor used in a cursor FOR loop. In the following example, you pass a department number:

```
    DECLARE
        CURSOR c1 (dnum NUMBER) IS
            SELECT sal, comm FROM emp WHERE deptno = dnum;
        ...
    BEGIN
        FOR emp_rec IN c1(20) LOOP
            ...
        END LOOP;
        ...
    END;
```

See also: Cursors, %ROWTYPE Attribute, LOOP Statement

ORDER BY Aliases

Conceptually, the rows returned by a multirow SELECT statement form a result table. Like the rows in a database table, the rows in a result table are not arranged in any particular order. However, you can use the ORDER BY clause to sort the rows in a result table by one or more columns.

If you want to sort the rows in a result table by a calculated (and therefore nameless) column, you can specify the corresponding select-item or its ordinal position. In the following example, the number 3 denotes the calculated column "sal+comm," which is the third select-item:

```
DECLARE
   CURSOR c1 IS
      SELECT empno, ename, sal+comm
         FROM emp
         WHERE job = 'SALESPERSON'
         ORDER BY 3;  -- or ORDER BY sal+comm
  ...
```

However, under SQL92, this method is a "deprecated feature" retained only for compatibility with SQL89 and likely to be removed from future versions of the standard.

A less error-prone method that ensures compliance with future SQL standards is available. You can specify a column alias for use in the ORDER BY clause. In the following example, the alias "wages" denotes the calculated column "sal+comm":

```
    DECLARE
       CURSOR c1 IS
          SELECT empno, ename, sal+comm  wages
             FROM emp
             WHERE job = 'SALESPERSON'
             ORDER BY wages;
      ...
```

Furthermore, you can use the optional keyword AS in the SELECT clause to improve readability, as follows:

```
    DECLARE
       CURSOR c1 IS
          SELECT empno, ename, sal+comm AS wages
             FROM emp
             WHERE job = 'SALESPERSON'
             ORDER BY wages;
      ...
```

Name Resolution

Avoid using column names as aliases. In the ORDER BY clause, an alias takes precedence over a column name. For example, the following ORDER BY clause sorts rows by the calculated column "sal+comm," not by the column "sal":

```
DECLARE
   CURSOR c1 IS
      SELECT empno, ename, sal+comm AS sal
         FROM emp
         WHERE job = 'SALESPERSON'
         ORDER BY sal;  -- sorts by sal+comm
  ...
```

If you use the name of a selected column as the alias of a calculated column, the Oracle Server issues an error message. An example of this mistake follows:

```
DECLARE
   CURSOR c1 IS
      SELECT empno, ename, sal, sal+comm AS sal
         FROM emp
         WHERE job = 'SALESPERSON'
         ORDER BY sal;  -- causes execution error
  ...
```

See also: Cursors, Packaged Cursors

Transaction Processing

Oracle is transaction oriented; that is, Oracle uses transactions to ensure data integrity. A transaction is a series of SQL data manipulation statements that does a logical unit of work. For example, two UPDATE statements might credit one bank account and debit another.

The first SQL statement in your program begins a transaction. When one transaction ends, the next SQL statement begins another transaction automatically. Thus, every SQL statement is part of a transaction.

You use the COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION statements to control transactions. COMMIT makes permanent any database changes made during the current transaction. Until you commit your changes, other users cannot see them. ROLLBACK ends the current transaction and undoes any changes made since the transaction began. SAVEPOINT marks the current point in the processing of a transaction. Used with ROLLBACK, SAVEPOINT undoes part of a transaction. SET TRANSACTION establishes a read-only transaction.

Distributed Transactions

A distributed transaction includes at least one SQL statement that updates data at multiple nodes in a distributed database. If the update affects only one node, the transaction is remote, not distributed.

If part of a distributed transaction fails, you must roll back the whole transaction or roll back to a savepoint. Oracle issues an error message in this situation. So, include a check for the error in every application that does distributed transactions. Note, however, that a PL/SQL block or subprogram cannot catch exceptions raised by a remote subprogram.

See also: COMMIT, LOCK TABLE, ROLLBACK, SAVEPOINT, SET TRANSACTION

Using COMMIT

The COMMIT statement ends the current transaction and makes permanent any changes made during that transaction. Until you commit the changes, other users cannot access the changed data; they see the data as it was before you made the changes.

Consider a simple transaction that transfers money from one bank account to another. The transaction requires two UPDATEs because it debits the first account, then credits the second. In the example below, after crediting the second account, you issue a COMMIT, which makes the changes permanent. Only then do other users see the changes.

```
BEGIN
    ...
    UPDATE accounts SET bal = my_bal - debit
        WHERE acctno = my_acctno;
    ...
    UPDATE accounts SET bal = my_bal + credit
        WHERE acctno = my_acctno;
    COMMIT;
END;
```

The COMMIT statement releases all row and table locks. It also erases any savepoints marked since the last COMMIT or ROLLBACK.

See also: ROLLBACK, SAVEPOINT, SET TRANSACTION, Using ROLLBACK, Using SAVEPOINT, Using SET TRANSACTION,

Using ROLLBACK

The ROLLBACK statement is the inverse of COMMIT. It ends the current transaction and undoes any changes made during that transaction. ROLLBACK is useful for two reasons. First, if you make a mistake, such as deleting the wrong row from a table, you can use ROLLBACK to restore the original data. ROLLBACK TO allows you to erase back to an intermediate statement in the current transaction so that you do not have to erase all your changes.

Second, ROLLBACK is useful when you start a transaction that you cannot finish because an exception is raised or a SQL statement fails. In such cases, ROLLBACK lets you return to the starting point to take corrective action and perhaps try again.

Consider the example below, in which you insert information about an employee into three different database tables. All three tables have a column that holds employee numbers and is constrained by a unique index. If an INSERT statement tries to store a duplicate employee number, the predefined exception DUP_VAL_ON_INDEX is raised. In that case, you want to undo all changes. So, you issue a ROLLBACK in the exception handler.

```
DECLARE
    emp_id  INTEGER;
    ...
BEGIN
    SELECT empno, ... INTO emp_id, ... FROM new_emp WHERE ...
    ...
    INSERT INTO emp VALUES (emp_id, ...);
    INSERT INTO tax VALUES (emp_id, ...);
    INSERT INTO pay VALUES (emp_id, ...);
    ...
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK;
    ...
END;
```

See also: COMMIT, SAVEPOINT, SET TRANSACTION, Using COMMIT, Using SAVEPOINT, Using SET TRANSACTION

Using SAVEPOINT

SAVEPOINT names and marks the current point in the processing of a transaction. Used with the ROLLBACK TO statement, savepoints let you undo parts of a transaction instead of the whole transaction. In the example below, you mark a savepoint before doing an insert. If the INSERT statement tries to store a duplicate value in the "empno" column, the predefined exception DUP_VAL_ON_INDEX is raised. In that case, you roll back to the savepoint, undoing just the insert.

```
DECLARE
    emp_id  emp.empno%TYPE;
BEGIN
    ...
    UPDATE emp SET ... WHERE empno = emp_id;
    DELETE FROM emp WHERE ...
    ...
    SAVEPOINT do_insert;
    INSERT INTO emp VALUES (emp_id, ...);
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK TO do_insert;
END;
```

When you roll back to a savepoint, any savepoints marked after that savepoint are erased. However, the savepoint to which you roll back is not erased. For example, if you mark five savepoints, then roll back to the third, only the fourth and fifth are erased. A simple ROLLBACK or COMMIT erases all savepoints.

If you mark a savepoint within a recursive subprogram, new instances of the SAVEPOINT statement are executed at each level in the recursive descent. However, you can only ROLLBACK TO the most recently marked savepoint.

Savepoint names are undeclared identifiers and can be reused within a transaction. This moves the savepoint from its old position to the current point in the transaction. Thus, a rollback to the savepoint affects only the current part of your transaction. Consider the following example:

```
BEGIN
    ...
    SAVEPOINT my_point;
    UPDATE emp SET ... WHERE empno = emp_id;
    ...
    SAVEPOINT my_point;  -- move my_point to current point
    INSERT INTO emp VALUES (emp_id, ...);
    ...
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK TO my_point;
END;
```

By default, the number of active savepoints per session is limited to 5. An active savepoint is one marked since the last commit or rollback. You or your DBA can raise the limit (up to 255) by increasing the value of the Oracle initialization parameter SAVEPOINTS.

See also: COMMIT, ROLLBACK, SET TRANSACTION, Using COMMIT, Using ROLLBACK, Using SET TRANSACTION

Implicit Rollbacks

Before executing an INSERT, UPDATE, or DELETE statement, Oracle marks an implicit savepoint (unavailable to you). If the statement fails, Oracle rolls back to the savepoint. For example, if an INSERT statement raises an exception by trying to insert a duplicate value in a unique index, the statement is rolled back.

Normally, just the failed SQL statement is rolled back, not the whole transaction. However, if the statement raises an unhandled exception, the host environment determines what is rolled back.

If you exit a stored subprogram with an unhandled exception, PL/SQL does not assign values to OUT parameters. Also, PL/SQL does not roll back database work done by the subprogram.

See also: COMMIT, ROLLBACK, Using COMMIT, Using ROLLBACK

Using SET TRANSACTION

The default state for all transactions is statement-level read consistency. This guarantees that a query sees only changes committed before it began executing, plus any changes made by prior statements in the current transaction. If other users commit changes to the relevant database tables, subsequent queries see those changes.

However, you can use the SET TRANSACTION statement to establish a read-only transaction, which provides transaction-level read consistency. This guarantees that a query sees only changes committed before the current transaction began. The SET TRANSACTION READ ONLY statement takes no additional parameters. An example follows:

```
SET TRANSACTION READ ONLY;
```

The SET TRANSACTION statement must be the first SQL statement in a read-only transaction and can only appear once in a transaction. Remember, if a transaction is set to READ ONLY, subsequent queries see only changes committed before the transaction began. The use of READ ONLY does not affect other users or transactions.

Only the SELECT, COMMIT, and ROLLBACK statements are allowed in a read-only transaction. For example, including an INSERT or DELETE statement raises an exception.

During a read-only transaction, all queries refer to the same snapshot of the database, providing a multitable, multiquery, read-consistent view. Other users can continue to query or update data as usual. A commit or rollback ends the transaction. In the example below, as a store manager, you use a read-only transaction to gather sales figures for the day, the past week, and the past month. The figures are unaffected by other users updating the database during the transaction.

```
DECLARE
    daily_sales    REAL;
    weekly_sales   REAL;
    monthly_sales  REAL;
BEGIN
    SET TRANSACTION READ ONLY;

    SELECT SUM(amt) INTO daily_sales FROM sales
        WHERE dte = SYSDATE;
    SELECT SUM(amt) INTO weekly_sales FROM sales
        WHERE dte > SYSDATE - 7;
    SELECT SUM(amt) INTO monthly_sales FROM sales
        WHERE dte > SYSDATE - 30;
    COMMIT;  -- simply ends the transaction since there
             -- are no changes to make permanent
    ...
END;
```

See also: SELECT, COMMIT, ROLLBACK, Using COMMIT, Using ROLLBACK

FOR UPDATE Clause

When declaring a cursor that will be referenced in the WHERE CURRENT OF clause of an UPDATE or DELETE statement, you must use the FOR UPDATE clause to acquire exclusive row locks.   If present, the FOR UPDATE clause must appear at the end of the cursor declaration, as the following example shows:

```
    DECLARE
        CURSOR c1 IS SELECT empno, sal FROM emp
            WHERE job = 'SALESMAN' AND comm > sal FOR UPDATE;
```

The FOR UPDATE clause indicates that rows will be updated or deleted and locks all rows in the active set.   This is useful when you want to base an update on the existing values in a row.   In that case, you must make sure the row is not changed by another user before the update.   All rows in the active set are locked when you OPEN the cursor.   The rows are unlocked when you COMMIT the transaction.   So, you cannot FETCH from a FOR UPDATE cursor after a COMMIT.

When querying multiple tables, you can use the FOR UPDATE OF clause to confine row locking to particular tables.   Rows in a table are locked only if the FOR UPDATE OF clause refers to a column in that table. For example, the following query locks rows in the "emp" table but not in the "dept" table:

```
    DECLARE
        CURSOR c1 IS SELECT ename, dname FROM emp, dept
            WHERE emp.deptno = dept.deptno AND job = 'MANAGER'
            FOR UPDATE OF sal;
```

You use the WHERE CURRENT OF clause in an UPDATE or DELETE statement to refer to the latest row FETCHed from a cursor, as the following example shows:

```
    DECLARE
        CURSOR c1 IS SELECT empno, job, sal FROM emp FOR UPDATE;
        ...
    BEGIN
        OPEN c1;
        LOOP
            FETCH c1 INTO ...
            ...
            UPDATE emp SET sal = new_sal WHERE CURRENT OF c1;
        END LOOP;
        ...
```

See also: DELETE, UPDATE, LOCK TABLE

Using LOCK TABLE

The LOCK TABLE statement lets you lock entire database tables in a specified lock mode so that you can share or deny access to tables while maintaining their integrity.   For example, the statement below locks the "emp" table in "row share" mode.   Row share locks allow concurrent access to a table; they prevent other users from locking the entire table for exclusive use.   Table locks are released when your transaction issues a COMMIT or ROLLBACK.

```
    LOCK TABLE emp IN ROW SHARE MODE NOWAIT;
```

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an "exclusive" lock. While one user has an exclusive lock on a table, no other users can INSERT, UPDATE, or DELETE rows in that table.

The optional keyword NOWAIT tells Oracle not to wait if the table has been locked by another user.   Control is immediately returned to your program so that it can do other work before trying again to acquire the lock.   If you omit the keyword NOWAIT, Oracle waits until the table is available; the wait has no set limit.

A table lock never keeps other users from querying a table, and a query never acquires a table lock.   Only if two different transactions try to modify the same row will one transaction wait for the other to complete.

See also: LOCK TABLE, FOR UPDATE Clause

Using DDL and Dynamic SQL

In this section, you learn why PL/SQL does not support SQL data definition language (DDL) and how to solve the problem.

Efficiency vs Flexibility

Before a PL/SQL program can be executed, it must be compiled (translated into machine language). The PL/SQL compiler resolves references to Oracle objects by looking up their definitions in the data dictionary. Then, the compiler assigns storage addresses to program variables that will hold Oracle data so that Oracle can look up the addresses at run time. This process is called "binding."

How a database language implements binding affects runtime efficiency and flexibility. Binding at compile time, called "static" or "early" binding, increases efficiency because the definitions of database objects are looked up then, not at run time. On the other hand, binding at run time, called "dynamic" or "late" binding, increases flexibility because the definitions of database objects can remain unknown until then.

Designed primarily for high-speed transaction processing, PL/SQL increases efficiency by bundling SQL statements and avoiding runtime compilation. Unlike SQL, which is compiled and executed statement-by-statement at run time (late binding), PL/SQL is processed into machine-readable "p-code" at compile time (early binding). At run time, the PL/SQL engine simply executes the p-code.

The Problem

However, this design imposes some limitations. For example, the p-code includes references to database objects such as tables and stored procedures. The PL/SQL compiler can resolve such references only if the database objects are known at compile time. In the following example, the compiler cannot process the procedure until the table is defined, but the table is undefined until the procedure is executed:

```
CREATE PROCEDURE create_table AS
BEGIN
   CREATE TABLE dept (deptno NUMBER(2), ...);  -- illegal
   ...
END create_table;
```

In the next example, the compiler cannot bind the table reference in the DROP TABLE statement because the table name is unknown until the procedure is executed:

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
BEGIN
   DROP TABLE table_name;  -- illegal
   ...
```

```
    END drop_table;
```

The Solution

Help is available. A package named DBMS_SQL, which is supplied with the Oracle Server, allows PL/SQL to execute SQL data definition and data manipulation statements dynamically at run time.

See also: Product-specific Packages

Ending Transactions

It is good programming practice to commit or roll back every transaction explicitly. Whether you issue the COMMIT or ROLLBACK in your PL/SQL program or in the host environment depends on the flow of application logic. If you neglect to commit or roll back a transaction explicitly, the host environment determines its final state.

For example, in the SQL*Plus environment, if your PL/SQL block does not include a COMMIT or ROLLBACK statement, the final state of your transaction depends on what you do after running the block. If you execute a data definition, data control, or COMMIT statement or if you issue the EXIT, DISCONNECT, or QUIT command, Oracle commits the transaction. If you execute a ROLLBACK statement or abort the SQL*Plus session, Oracle rolls back the transaction.

In the Oracle Precompiler environment, if your program does not terminate normally, Oracle rolls back your transaction. A program terminates normally when it explicitly commits or rolls back work and disconnects from Oracle using the RELEASE parameter, as follows:

```
    EXEC SQL COMMIT WORK RELEASE;
```

In the OCI environment, if you issue the OLOGOF call, Oracle commits your transaction automatically. Otherwise, Oracle rolls back the transaction.

See also: Transaction Processing, COMMIT, ROLLBACK, Using COMMIT, Using ROLLBACK

Database Triggers

A database trigger is a stored PL/SQL program unit associated with a specific database table. Oracle executes (fires) the database trigger automatically whenever a given SQL operation affects the table. So, unlike subprograms, which must be invoked explicitly, database triggers are invoked implicitly. Among other things, you can use database triggers to

```
    * audit data modifications
    * log events transparently
    * enforce complex business rules
    * derive column values automatically
    * implement complex security authorizations
    * maintain replicate tables
```

You can associate up to 12 database triggers with a given table. To create a database trigger, you must have CREATE TRIGGER privileges and either own the associated table, have ALTER privileges for the associated table, or have ALTER ANY TABLE privileges.

A database trigger has three parts: a triggering event, an optional trigger constraint, and a trigger action. When the event occurs, the database trigger fires and an anonymous PL/SQL block performs the action. Database triggers fire with the privileges of the owner, not the current user. So, the owner must have appropriate access to all objects referenced by the trigger action.

The example below illustrates transparent event logging. The database trigger named "reorder" ensures that a part is reordered when its quantity on hand drops below the reorder point.

```
    CREATE TRIGGER reorder
        /* triggering event */
        AFTER UPDATE OF qty_on_hand ON inventory  -- table
        FOR EACH ROW
        /* trigger constraint */
        WHEN (new.reorderable = 'T')
    BEGIN
        /* trigger action */
        IF :new.qty_on_hand < :new.reorder_point THEN
            INSERT INTO pending_orders
                VALUES (:new.part_no, :new.reorder_qty, SYSDATE);
        END IF;
    END;
```

The name in the ON clause identifies the database table associated with the database trigger. The triggering event specifies the SQL data manipulation statement that affects the table. In this case, the statement is UPDATE. If the trigger statement fails, it is rolled back. The keyword AFTER specifies that the database trigger fires after the update is done.

By default, a database trigger fires once per table. The FOR EACH ROW option specifies that the trigger fires once per row. For the trigger to fire, however, the Boolean expression in the WHEN clause must evaluate to TRUE.

The prefix ":new" is a correlation name that refers to the newly updated column value. Within a database trigger, you can reference ":new" and ":old" values of changing rows. Notice that the colon is not used in the WHEN clause. You can use the REFERENCING clause (not shown) to replace ":new" and ":old" with other correlation names.

Except for transaction control statements such as COMMIT and ROLLBACK, any SQL or procedural statement, including subprogram calls, can appear in the BEGIN ... END block. A database trigger can also have DECLARE and EXCEPTION sections.

The next example shows that the trigger action can include calls to the built-in Oracle procedure "raise_application_error," which lets you issue user-defined error messages:

```
CREATE TRIGGER check_salary
    BEFORE INSERT OR UPDATE OF sal, job ON emp
    FOR EACH ROW
    WHEN (new.job != 'PRESIDENT')
DECLARE
    minsal  NUMBER;
    maxsal  NUMBER;
BEGIN
    /* Get salary range for a given job from table sals. */
    SELECT losal, hisal INTO minsal, maxsal FROM sals
        WHERE job = :new.job;
    /* If salary is out of range, increase is negative, *
     * or increase exceeds 10%, raise an exception.     */
    IF (:new.sal < minsal OR :new.sal > maxsal) THEN
        raise_application_error(-20225, 'Salary out of range');
    ELSIF (:new.sal < :old.sal) THEN
        raise_application_error(-20230, 'Negative increase');
    ELSIF (:new.sal > 1.1 * :old.sal) THEN
        raise_application_error(-20235, 'Increase exceeds 10%');
    END IF;
END;
```

See also: CREATE TRIGGER, Subprograms, raise_application_error Procedure

Exceptions

PL/SQL makes it easy to detect and process predefined and user-defined error conditions called "exceptions." When an error occurs, an exception is raised. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. To handle raised exceptions, you write separate routines called "exception handlers."

Predefined exceptions are raised implicitly by the runtime system. For example, if you try to divide a number by zero, the predefined exception ZERO_DIVIDE is raised automatically. User-defined exceptions must be raised explicitly by RAISE statements.

You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or database trigger. In the executable part, you check for the condition that needs special attention. If you find that the condition exists, you execute a RAISE statement. In the following example, if a salesperson's commission is null, you raise an exception named "comm_missing":

```
PROCEDURE calc_bonus (emp_id INTEGER, bonus OUT REAL) IS
    my_sal        NUMBER(7,2);
    my_comm       NUMBER(7,2);
    comm_missing  EXCEPTION;  -- declare exception
BEGIN
    SELECT sal, comm INTO my_sal, my_comm FROM emp
        WHERE empno = emp_id;
    IF my_comm IS NULL THEN
        RAISE comm_missing;  -- raise exception
    ELSE
        bonus := (my_sal * 0.05) + (my_comm * 0.15);
    END IF;
EXCEPTION  -- begin exception handlers
    WHEN comm_missing THEN
        -- process error
    WHEN OTHERS THEN  -- handles all other errors
        ...
END calc_bonus;
```

The optional OTHERS handler catches all exceptions that the procedure does not name specifically.

Declaring Exceptions

You declare an exception by introducing its name, followed by the keyword EXCEPTION. In the following example, you declare an exception named "past_due":

```
DECLARE
    past_due  EXCEPTION;
    acct_num  NUMBER(5);
BEGIN
```

```
        ...
```

Exception and variable declarations are similar. But, an exception is an error condition, not an object. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

Scope Rules

You cannot declare an exception twice in the same block. You can, however, declare the same exception in two different blocks.

Exceptions declared in a block are considered local to that block and global to all its sub-blocks. Because a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions declared in a sub-block.

If you redeclare a global exception in a sub-block, the local declaration prevails. So, the sub-block cannot reference the global exception unless it was declared in a labeled block, in which case the following syntax is valid:

```
    block_label.exception_name
```

The next example illustrates the scope rules:

```
    DECLARE
        past_due  EXCEPTION;
        acct_num  NUMBER;
    BEGIN
        ...
        ---------------- beginning of sub-block ----------------
        DECLARE
            past_due  EXCEPTION;  -- this declaration prevails
            acct_num  NUMBER;
        BEGIN
            ...
            IF ... THEN
                RAISE past_due;  -- this is not handled
            END IF;
            ...
        END;
        ------------------- end of sub-block -------------------
    EXCEPTION
        WHEN past_due THEN  -- does not handle RAISEd exception
            ...
    END;
```

The enclosing block does not handle the RAISEd exception because the declaration of "past_due" in the sub-block prevails. Though they share the same name, the two "past_due" exceptions are different, just as the two "acct_num" variables share the same name but are different variables. Therefore, the RAISE statement and the WHEN clause refer to different exceptions.   To have the enclosing block handle the RAISEd exception, you must remove its declaration from the sub-block or define an OTHERS handler.

See also: Exception Handlers, Predefined Exceptions, RAISE Statement

Exception Handlers

When an exception is raised, normal execution of your PL/SQL block or subprogram stops and control transfers to its exception-handling part, which is formatted as follows:

```
...
EXCEPTION
    WHEN exception_name1 THEN  -- handler
        sequence_of_statements1
    WHEN exception_name2 THEN  -- another handler
        sequence_of_statements2
    ...
    WHEN OTHERS THEN          -- optional handler
        sequence_of_statements3
END;
```

To catch raised exceptions, you must write exception handlers. Each handler consists of a WHEN clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised. These statements complete execution of the block or subprogram; control does not return to where the exception was raised. In other words, you cannot resume processing where you left off.

The optional OTHERS exception handler, which is always the last handler in a block or subprogram, acts as the handler for all exceptions not named specifically. Thus, a block or subprogram can have only one OTHERS handler. Consider the following example:

```
...
EXCEPTION
    WHEN ... THEN
        -- handle the error
    WHEN ... THEN
        -- handle the error
    ...
    WHEN OTHERS THEN
        -- handle all other errors
END;
```

Using the OTHERS handler guarantees that no exception will go unhandled.

If you want two or more exceptions to execute the same sequence of statements, list the exception names in the WHEN clause, separating them by the keyword OR, as follows:

```
EXCEPTION
    WHEN over_limit OR under_limit OR VALUE_ERROR THEN
        -- handle the error
    ...
```

If any of the exceptions in the list is raised, the associated sequence of statements is executed. The keyword OTHERS cannot appear in the list of exception names; it must appear by itself. You can have any number of exception handlers, and each handler can associate a list of exceptions with a sequence of statements. However, an exception name can appear only once in the exception-handling part of a PL/SQL block or subprogram.

The usual scope rules for PL/SQL variables apply, so only local and global variables can be referenced in an exception handler. However, when an exception is raised inside a cursor FOR loop, the cursor is closed implicitly before the handler is invoked. So, the values of explicit cursor attributes are not available in the handler.

See also: Exceptions, Predefined Exceptions, RAISE Statement

Predefined Exceptions

An internal exception is raised implicitly whenever a 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, the predefined exception NO_DATA_FOUND is raised if a SELECT INTO statement returns no rows.

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 table below. Also shown are the corresponding Oracle error codes.

Exception	Error	Raised if ...
CURSOR_ALREADY_OPEN	ORA-06511	you try to OPEN an already open cursor; you must CLOSE a cursor before you can reOPEN it
DUP_VAL_ON_INDEX	ORA-00001	you try to INSERT or UPDATE duplicate values in a UNIQUE database column
INVALID_CURSOR	ORA-01001	you try an illegal cursor operation such as closing an unopened cursor
INVALID_NUMBER	ORA-01722	the conversion of a character string to a number fails in a SQL statement
LOGIN_DENIED	ORA-01017	you log on to Oracle with an invalid username/password
NO_DATA_FOUND	ORA-01403	a SELECT INTO returns no rows, or you refer to an uninitialized row in a PL/SQL table
NOT_LOGGED_ON	ORA-01012	your PL/SQL program issues a database call without being logged on to Oracle
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem such as exiting a function that has no RETURN statement
STORAGE_ERROR	ORA-06500	PL/SQL runs out of memory or memory is corrupted
TIMEOUT_ON_RESOURCE	ORA-00051	a timeout occurs while Oracle is waiting for a resource
TOO_MANY_ROWS	ORA-01422	a SELECT INTO returns more than one row
TRANSACTION_BACKED_OUT	ORA-00061	the remote part of a transaction is rolled back because Oracle data might be inconsistent at some nodes
VALUE_ERROR	ORA-06502	the conversion of a character string to a number fails in a procedural statement, or an

		arithmetic, conversion, truncation, or constraint error occurs
ZERO_DIVIDE	ORA-01476	you try to divide a number by zero

Redeclaring Predefined Exceptions

Remember, PL/SQL declares predefined exceptions globally in package STANDARD, so you need not declare them yourself. Redeclaring predefined exceptions is error-prone because your local declaration overrides the global declaration. For example, if you declare an exception named "invalid_number" and then PL/SQL raises the predefined exception INVALID_NUMBER internally, a handler written for INVALID_NUMBER will not catch the internal exception. In such cases, you must use dot notation to specify the predefined exception, as follows:

```
        ...
    EXCEPTION
        WHEN invalid_number OR STANDARD.INVALID_NUMBER THEN
            -- handle the error
        ...
    END;
```

See also: Exceptions, EXCEPTION_INIT Pragma, Exception Handlers,
          RAISE Statement, Package STANDARD

EXCEPTION_INIT Pragma

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.   They do not affect the meaning of a program; they simply convey information to the compiler.

The predefined pragma EXCEPTION_INIT tells the PL/SQL 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 part, as shown in the following example:

```
DECLARE
    insufficient_privileges  EXCEPTION;
    PRAGMA EXCEPTION_INIT(insufficient_privileges, -1031);
        -----------------------------------------------------
        -- Oracle returns error number -1031 if, for example,
        -- you try to UPDATE a table for which you have
        -- only SELECT privileges
        -----------------------------------------------------
BEGIN
    ...
EXCEPTION
    WHEN insufficient_privileges THEN
        -- handle the error
    ...
END;
```

See also: Exceptions, Exception Handlers, RAISE Statement,

RAISE Statement

PL/SQL blocks and subprograms should RAISE an exception only when an error makes it undesirable or impossible to finish processing. You can code a RAISE statement for a given exception anywhere within the scope of that exception. In the following example, you alert your PL/SQL block to a user-defined exception named "out_of_stock":

```
DECLARE
    out_of_stock    EXCEPTION;
    number_on_hand  NUMBER(4);
BEGIN
    ...
    IF number_on_hand < 1 THEN
        RAISE out_of_stock;
    END IF;
    ...
EXCEPTION
    WHEN out_of_stock THEN
         -- handle the error
END;
```

As the following example shows, you can also raise a predefined exception explicitly:

```
RAISE INVALID_NUMBER;
```

That way, you can use an exception handler written for the predefined exception to process other errors, as the next example shows:

```
DECLARE
    acct_type  INTEGER;
    ...
BEGIN
    ...
    IF acct_type NOT IN (1, 2, 3) THEN
        RAISE INVALID_NUMBER;
    END IF;
    ...
EXCEPTION
    WHEN INVALID_NUMBER THEN
        ROLLBACK;
    ...
END;
```

See also: Exceptions, Exception Handlers, Predefined Exceptions

How Exceptions Propagate

When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search. In the latter case, PL/SQL returns an unhandled exception error to the host environment.

An exception can propagate beyond its scope, that is, beyond the block in which it was declared. Consider the following example:

```
DECLARE
    ...
BEGIN
    ...
    ---------------- beginning of sub-block ----------------
    DECLARE
        past_due  EXCEPTION;
    BEGIN
        ...
        IF ... THEN
            RAISE past_due;
        END IF;
        ...
    END;
    ------------------- end of sub-block -------------------
    ...
EXCEPTION
    ...
    WHEN OTHERS THEN
        ROLLBACK;
END;
```

Because the block in which it was declared has no handler for the exception named "past_due," it propagates to the enclosing block. But, according to the scope rules, enclosing blocks cannot reference exceptions declared in a sub-block. So, only an OTHERS handler can catch the exception.

Exceptions Raised in Declarations

Exceptions can be raised in declarations by a faulty initialization expression. For example, the following declaration implicitly raises the predefined exception VALUE_ERROR because "limit" cannot store numbers larger than 999:

```
DECLARE
    limit  CONSTANT  NUMBER(3) := 5000;  -- raises VALUE_ERROR
BEGIN
    ...
EXCEPTION
```

```
        WHEN VALUE_ERROR THEN  -- cannot catch the exception
           ...
    END;
```

Handlers in the current block cannot catch the raised exception because an exception raised in a declaration propagates immediately to the enclosing block.

Exceptions Raised in Handlers

Only one exception at a time can be active in the exception-handling part of a block or subprogram.   So, an exception raised inside a handler immediately propagates to the enclosing block, which is searched to find a handler for the newly raised exception.   From there on, the exception propagates normally.

See also: Exceptions, Exception Handlers, Predefined Exceptions, RAISE Statement

Reraising an Exception

Sometimes, you want to "reraise" an exception, that is, handle it locally, then pass it to an enclosing block. For example, you might want to roll back a transaction in the current block, then log the error in an enclosing block.

To reraise an exception, simply place a RAISE statement in the local handler, as shown in the following example:

```
DECLARE
    out_of_balance  EXCEPTION;
BEGIN
    ...
    ---------------- beginning of sub-block ----------------
    BEGIN
        ...
        IF ... THEN
            RAISE out_of_balance;  -- raise the exception
        END IF;
        ...
    EXCEPTION
        WHEN out_of_balance THEN
            -- handle the error
            RAISE;  -- reraise the current exception
        ...
    END;
    ------------------- end of sub-block -------------------
EXCEPTION
    WHEN out_of_balance THEN
        -- handle the error differently
    ...
END;
```

Omitting the exception name in a RAISE statement, which is allowed only in an exception handler, reraises the current exception.

See also: Exceptions, Exception Handlers, Predefined Exceptions, RAISE Statement

Using SQLCODE and SQLERRM

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

For internal exceptions, SQLCODE returns the number of the associated 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 message associated with the Oracle error that occurred. The message begins with the Oracle error code.

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 following message:

```
ORA-0000: normal, successful completion
```

You can pass an error number to SQLERRM, in which case SQLERRM returns the message associated with that error number. The error number passed to SQLERRM should be negative. Passing a zero to SQLERRM always returns the following message:

```
ORA-0000: normal, successful completion
```

Passing a positive number to SQLERRM always returns the message

```
User-Defined Exception
```

unless you pass +100, in which case SQLERRM returns this message:

```
ORA-01403: no data found
```

In the following example, SQLERRM gives unwanted results because it is passed positive rather than negative numbers:

```
DECLARE
    msg  CHAR(100);
BEGIN
    FOR num IN 1..9999 LOOP
        msg := SQLERRM(num);  -- should be SQLERRM(-num)
        INSERT INTO errors VALUES (msg);
    END LOOP;
END;
```

You cannot use SQLCODE or SQLERRM directly in a SQL statement. For example, the following statement is illegal:

```
    INSERT INTO errors VALUES (SQLCODE, SQLERRM);
```

Instead, you must assign their values to local variables, then use the variables in the SQL statement, as the following example shows:

```
    DECLARE
        err_num  NUMBER;
        err_msg  CHAR(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."   SQLCODE and SQLERRM are especially useful in the OTHERS exception handler because they tell you which internal exception was raised.

See also: Exceptions, EXCEPTION_INIT Pragma, Exception Handlers

Unhandled Exceptions

If it cannot find a handler for a raised exception, PL/SQL returns an unhandled exception error to the host environment, which determines the outcome.   For example, in the Oracle Precompilers environment, any database changes made by a failed SQL statement or PL/SQL block are rolled back.

Unhandled exceptions can affect stored subprograms.   If you exit a subprogram successfully, PL/SQL assigns values to OUT parameters. However, if you exit with an unhandled exception, PL/SQL does not assign values to OUT parameters.   Also, PL/SQL does not roll back database work done by the subprogram.

You can avoid unhandled exceptions by coding an OTHERS handler at the topmost level of every PL/SQL program.

See also: Exceptions, Exception Handlers

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.

Like unnamed or anonymous PL/SQL blocks, subprograms have a declarative part, an executable part, and an optional exception-handling part.
The declarative part contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These objects are local and cease to exist when you exit the subprogram. The executable part contains statements that assign values, control execution, and manipulate Oracle data. The exception-handling part contains exception handlers, which deal with exceptions raised during execution.

Consider the following procedure named "debit_account," which debits a bank account:

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
    old_balance  REAL;
    new_balance  REAL;
    overdrawn    EXCEPTION;
BEGIN
    SELECT bal INTO old_balance FROM accts
        WHERE acctno = acct_id;
    new_balance := old_balance - amount;
    IF new_balance < 0 THEN
        RAISE overdrawn;
    ELSE
        UPDATE accts SET bal = new_balance
            WHERE acctno = acct_id;
    END IF;
EXCEPTION
    WHEN overdrawn THEN
        ...
END debit_account;
```

When invoked or "called," this procedure accepts an account number and a debit amount. It uses the account number to select the account balance from the "accts" database table. Then, it uses the debit amount to compute a new balance. If the new balance is less than zero, an exception is raised; otherwise, the bank account is updated.

Subprograms can be defined using any Oracle tool that supports PL/SQL. They can be declared in PL/SQL blocks, procedures, functions, and packages. However, subprograms must be declared at the end of a declarative section after all other program objects. For example, the following procedure declaration is misplaced:

```
DECLARE
    PROCEDURE award_bonus (...) IS  -- misplaced; must come last
    BEGIN ... END;
    rating  NUMBER;
```

```
        ...
```

Generally, tools such as Oracle Forms that incorporate the PL/SQL engine can store subprograms locally for later, strictly local execution. However, to become available for general use by all tools, subprograms must be stored in an Oracle database.

See also: Procedures, Functions, Stored Subprograms

Procedures

A procedure is a subprogram that performs a specific action. You write procedures using the syntax

```
PROCEDURE name [ (parameter [, parameter, ...]) ] IS
    [local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [name];
```

where "parameter" stands for the following syntax:

```
var_name [IN | OUT | IN OUT] datatype [{:= | DEFAULT'D value]
```

Unlike the datatype specifier in a variable declaration, the datatype specifier in a parameter declaration must be unconstrained. For example, the following declaration of "name" is illegal:

```
PROCEDURE ... (name CHAR(20)) IS  -- illegal; should be CHAR
BEGIN ... END;
```

A procedure has two parts: the specification and the body. The procedure specification begins with the keyword PROCEDURE and ends with the procedure name or a parameter list. Parameter declarations are optional. Procedures that take no parameters are written without parentheses.

The procedure body begins with the keyword IS and ends with the keyword END followed by an optional procedure name. The procedure body has three parts: a declarative part, an executable part, and an optional exception-handling part.

The declarative part contains local declarations, which are placed between the keywords IS and BEGIN. The keyword DECLARE, which introduces declarations in an anonymous PL/SQL block, is not used. The executable part contains statements, which are placed between the keywords BEGIN and EXCEPTION (or END). At least one statement must appear in the executable part of a procedure. The NULL statement meets this requirement. The exception-handling part contains exception handlers, which are placed between the keywords EXCEPTION and END.

Consider the procedure "raise_salary," which increases the salary of an employee:

```
PROCEDURE raise_salary (emp_id INTEGER, increase REAL) IS
    current_salary  REAL;
    salary_missing  EXCEPTION;
BEGIN
    SELECT sal INTO current_salary FROM emp
        WHERE empno = emp_id;
    IF current_salary IS NULL THEN
```

```
            RAISE salary_missing;
        ELSE
            UPDATE emp SET sal = sal + increase
                WHERE empno = emp_id;
        END IF;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            INSERT INTO emp_audit VALUES (emp_id, 'No such number');
        WHEN salary_missing THEN
            INSERT INTO emp_audit VALUES (emp_id, 'Salary is null');
    END raise_salary;
```

When called, this procedure accepts an employee number and a salary increase amount. It uses the employee number to select the current salary from the "emp" database table. If the employee number is not found or if the current salary is null, an exception is raised. Otherwise, the salary is updated.

A procedure is called as a PL/SQL statement. For example, the procedure "raise_salary" might be called as follows:

```
    raise_salary(emp_num, amount);
```

See also: Functions, Subprograms, Stored Subprograms

Functions

A function is a subprogram that computes a value. Functions and procedures are structured alike, except that functions have a RETURN clause. You write functions using the syntax

```
    FUNCTION name [ (parameter [, parameter, ...]) ] RETURN datatype IS
        [local declarations]
    BEGIN
        executable statements
    [EXCEPTION
        exception handlers]
    END [name];
```

where "parameter" stands for the following syntax:

```
    var_name [IN | OUT | IN OUT] datatype [{:= | DEFAULT'D value]
```

The datatype specifier in a parameter declaration must be unconstrained.

Like a procedure, a function has two parts: the specification and the body. The function specification begins with the keyword FUNCTION and ends with the RETURN clause, which specifies the datatype of the result value. Parameter declarations are optional. Functions that take no parameters are written without parentheses.

The function body begins with the keyword IS and ends with the keyword END followed by an optional function name. The function body has three parts: a declarative part, an executable part, and an optional exception-handling part.

The declarative part contains local declarations, which are placed between the keywords IS and BEGIN. The keyword DECLARE is not used. The executable part contains statements, which are placed between the keywords BEGIN and EXCEPTION (or END). One or more RETURN statements must appear in the executable part of a function. The exception-handling part contains exception handlers, which are placed between the keywords EXCEPTION and END.

Consider the function "sal_ok," which determines if an employee salary is out of range:

```
    FUNCTION sal_ok (salary REAL, title REAL)
      RETURN BOOLEAN IS
        min_sal  REAL;
        max_sal  REAL;
    BEGIN
        SELECT losal, hisal INTO min_sal, max_sal FROM sals
            WHERE job = title;
        RETURN (salary >= min_sal) AND (salary <= max_sal);
    END sal_ok;
```

When called, this function accepts an employee salary and job title. It uses the job title to select range limits from the "sals" database

table.   The function identifier, "sal_ok," is set to a Boolean value
by the RETURN statement.   If the salary is out of range, "sal_ok" is set
to FALSE; otherwise, "sal_ok" is set to TRUE.

A function is called as part of an expression.   For example, the
function "sal_ok" might be called as follows:

```
IF sal_ok(new_sal, new_title) THEN
    ...
END IF;
...
promotable := sal_ok(new_sal, new_title) AND (rating > 3);
```

In both cases, the function identifier is an expression that yields a
result.

Calls to functions defined within a PL/SQL block or subprogram can
appear in procedural statements, but not in SQL statements.   For
example, the following INSERT statement is illegal:

```
DECLARE
    empnum  INTEGER;
    ...
    FUNCTION bonus (emp_id INTEGER) RETURN REAL IS
    BEGIN ... END bonus;
BEGIN
    ...
    INSERT INTO payroll
        VALUES (empnum, ..., bonus(empnum));  -- illegal call
END;
```

However, calls to a stored function can appear in SQL statements if the
function meets certain requirements.

See also: Procedures, Subprograms, Stored Subprograms, Calling Stored
          Subprograms, Calling Stored Functions from SQL

RETURN Statement

The RETURN statement immediately completes the execution of a subprogram and returns control to the caller. Execution then resumes with the statement following the subprogram call. Do not confuse the RETURN statement with the RETURN clause, which specifies the datatype of the result value in a function specification.

A subprogram can contain several RETURN statements, none of which need be the last lexical statement. Executing any of them completes the subprogram immediately. However, it is poor programming practice to have multiple exit points in a subprogram.

In procedures, a RETURN statement cannot contain an expression. The statement simply returns control to the caller before the normal end of the procedure is reached.

However, in functions, a RETURN statement must contain an expression, which is evaluated when the statement is executed. The resulting value is assigned to the function identifier. Observe how the function "balance" RETURNs the balance of a specified bank account:

```
    FUNCTION balance (acct_id INTEGER) RETURN REAL IS
        acct_bal  REAL;
    BEGIN
        SELECT bal INTO acct_bal FROM accts WHERE acctno = acct_id;
        RETURN acct_bal;
    END balance;
```

A function must contain at least one RETURN statement. Otherwise, PL/SQL raises the predefined exception PROGRAM_ERROR at run time.

See also: Procedures, Functions, EXIT Statement

Forward Declarations

PL/SQL requires that you declare an identifier before using it. Therefore, you must declare a subprogram before calling it. For example, the following declaration of procedure "award_bonus" is illegal because "award_bonus" calls procedure "calc_rating," which is not yet declared when the call is made:

```
DECLARE
    PROCEDURE award_bonus (...) IS
    BEGIN
        calc_rating(...);  -- undeclared identifier
        ...
    END;

    PROCEDURE calc_rating (...) IS
    BEGIN
        ...
    END;
    ...
```

In this case, you can solve the problem easily by placing procedure "calc_rating" before procedure "award_bonus." However, the easy solution does not always work. For example, suppose the procedures are mutually recursive (call each other) or you want to define them in alphabetical order.

PL/SQL solves this problem by providing a special subprogram declaration called a "forward declaration." You can use forward declarations to

```
* define subprograms in logical or alphabetical order
* define mutually recursive subprograms
* group subprograms in a package
```

A forward declaration consists of a subprogram specification terminated by a semicolon. In the next example, the forward declaration advises PL/SQL that the body of procedure "calc_rating" can be found later in the block.

```
DECLARE
    PROCEDURE calc_rating (...);  -- forward declaration

    /* Define subprograms in alphabetical order. */

    PROCEDURE award_bonus (...) IS
    BEGIN
        calc_rating(...);
        ...
    END;

    PROCEDURE calc_rating (...) IS
    BEGIN
        ...
        END;
        ...
```

Although the formal parameter list appears in the forward declaration, it must also appear in the subprogram body. You can place the subprogram body anywhere after the forward declaration, but they must appear in the same block, subprogram, or package.

See also: Subprograms, Packaged Subprograms, Recursion

Packaged Subprograms

Forward declarations let you group logically related subprograms in a package. The subprogram specifications go in the package specification, and the subprogram bodies go in the package body, where they are invisible to applications. Thus, packages allow you to hide implementation details. An example follows:

```
    PACKAGE emp_actions IS  -- package specification
        PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);
        PROCEDURE fire_employee (emp_id NUMBER);
    END emp_actions;

    PACKAGE BODY emp_actions IS  -- package body
        PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS
        BEGIN
            INSERT INTO emp VALUES (empno, ename, ...);
        END hire_employee;

        PROCEDURE fire_employee (emp_id NUMBER) IS
        BEGIN
            DELETE FROM emp WHERE empno = emp_id;
        END fire_employee;
    END emp_actions;
```

You can define subprograms in a package body without declaring their specifications in the package specification. However, such subprograms can be called only from inside the package.

See also: Subprograms, Packages, CREATE PACKAGE, CREATE PACKAGE BODY

Actual vs Formal Parameters

Subprograms pass information using parameters.   The variables or expressions referenced in the parameter list of a subprogram call are "actual" parameters.   For example, the following procedure call lists two actual parameters named "emp_num" and "amount":

```
    raise_salary(emp_num, amount);
```

The next procedure call shows that in some cases, expressions can be used as actual parameters:

```
    raise_salary(emp_num, merit + cola);
```

The variables declared in a subprogram specification and referenced in the subprogram body are "formal" parameters.   For example, the following procedure declares two formal parameters named "emp_id" and "increase":

```
    PROCEDURE raise_salary (emp_id INTEGER, increase REAL) IS
        current_salary  REAL;
        ...
    BEGIN
        SELECT sal INTO current_salary FROM emp WHERE empno = emp_id;
        ...
        UPDATE emp SET sal = sal + increase WHERE empno = emp_id;
    END raise_salary;
```

Though not necessary, it is good programming practice to use different names for actual and formal parameters.

When you call procedure "raise_salary," the actual parameters are evaluated and the result values are assigned to the corresponding formal parameters.   Before assigning the value of an actual parameter to a formal parameter, PL/SQL converts the datatype of the value if necessary.   For example, the following call to "raise_salary" is legal:

```
    raise_salary(emp_num, '2500');
```

The actual parameter and its corresponding formal parameter must belong to compatible datatypes.   For instance, PL/SQL cannot convert between the DATE and REAL datatypes.   Also, the result value must be convertible to the new datatype.   The following procedure call raises the predefined exception VALUE_ERROR because PL/SQL cannot convert the second actual parameter to a number:

```
    raise_salary(emp_num, '$2500');  -- note the dollar sign
```

See also: Procedures, Functions, Parameter Modes, Parameter Default Values

Positional and Named Notation

When calling a subprogram, you can write the actual parameters using either positional or named notation.   That is, you can indicate the association between an actual and formal parameter by position or name. For example, given the declarations

```
DECLARE
    acct  INTEGER;
    amt   REAL;
    PROCEDURE credit (acctno INTEGER, amount REAL) IS ...
    ...
```

you can call the procedure "credit" in four logically equivalent ways:

```
BEGIN
    credit(acct, amt);                      -- positional notation
    credit(amount => amt, acctno => acct);  -- named notation
    credit(acctno => acct, amount => amt);  -- named notation
    credit(acct, amount => amt);            -- mixed notation
    ...
END;
```

The first procedure call uses positional notation.   The PL/SQL compiler associates the first actual parameter, "acct," with the first formal parameter, "acctno."   And, the compiler associates the second actual parameter, "amt," with the second formal parameter, "amount."

The second procedure call uses named notation.   The arrow associates the formal parameter to the left of the arrow with the actual parameter to the right of the arrow.

The third procedure call also uses named notation and shows that you can list the parameter pairs in any order.   So, you need not know the order in which the formal parameters are listed.

The fourth procedure call shows that you can mix positional and named notation.   In this case, the first parameter uses positional notation, and the second parameter uses named notation.   Positional notation must precede named notation.   The reverse is not allowed.   For example, the following procedure call is illegal:

```
credit(acctno => acct, amt);  -- illegal
```

See also: Procedures, Functions, Parameter Modes, Parameter Default Values

Parameter Modes

You use parameter modes to define the behavior of formal parameters. The three parameter modes, IN (the default), OUT, and IN OUT, can be used with any subprogram. However, avoid using the OUT and IN OUT modes with functions. The purpose of a function is to take zero or more parameters and return a single value. It is poor programming practice to have a function return multiple values. Also, functions should be free of side effects, which change the values of variables not local to the subprogram.

IN Mode

An IN parameter lets you pass values to the subprogram being called. Inside the subprogram, an IN parameter acts like a constant. Therefore, it cannot be assigned a value. For example, the following assignment statement causes a compilation error:

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
    minimum_purchase  CONSTANT REAL := 10.0;
    service_charge    CONSTANT REAL := 0.50;
BEGIN
    IF amount < minimum_purchase THEN
        amount := amount + service_charge;  -- illegal
    END IF;
   ...
```

The actual parameter that corresponds to an IN formal parameter can be a constant, literal, initialized variable, or expression.

Unlike OUT and IN OUT parameters, IN parameters can be initialized to default values.

OUT Mode

An OUT parameter lets you return values to the caller of a subprogram. Inside the subprogram, an OUT parameter acts like an uninitialized variable. Therefore, its value cannot be assigned to another variable or reassigned to itself. For instance, the following assignment statement causes a compilation error:

```
PROCEDURE calc_bonus (emp_id INTEGER, bonus OUT REAL) IS
    hire_date  DATE;
BEGIN
    SELECT sal * 0.10, hiredate INTO bonus, hire_date FROM emp
        WHERE empno = emp_id;
    IF MONTHS_BETWEEN(SYSDATE, hire_date) > 60 THEN
        bonus := bonus + 500;  -- syntax error
    END IF;
    ...
```

The actual parameter that corresponds to an OUT formal parameter must be a variable; it cannot be a constant or expression. For example, the following procedure call is illegal:

```
    calc_bonus(7499, salary + commission);  -- syntax error
```

PL/SQL checks for this syntax error at compile time to prevent the overwriting of constants and expressions.

An OUT actual parameter can (but need not) have a value before the subprogram is called. However, the value is lost when you call the subprogram. Inside the subprogram, an OUT formal parameter cannot be used in an expression; the only operation allowed on the parameter is to assign it a value.

Before exiting a subprogram, explicitly assign values to all OUT formal parameters. Otherwise, the values of corresponding actual parameters are indeterminate. If you exit successfully, PL/SQL assigns values to the actual parameters. However, if you exit with an unhandled exception, PL/SQL does not assign values to the actual parameters.

IN OUT Mode

An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller. Inside the subprogram, an IN OUT parameter acts like an initialized variable. Therefore, it can be assigned a value and its value can be assigned to another variable. That means you can use an IN OUT formal parameter as if it were a normal variable. You can change its value or reference the value in any way, as the following example shows:

```
    PROCEDURE calc_bonus (emp_id INTEGER, bonus IN OUT REAL) IS
        hire_date      DATE;
        bonus_missing  EXCEPTION;
    BEGIN
        SELECT sal * 0.10, hiredate INTO bonus, hire_date FROM emp
            WHERE empno = emp_id;
        IF bonus IS NULL THEN
            RAISE bonus_missing;
        END IF;
        IF MONTHS_BETWEEN(SYSDATE, hire_date) > 60 THEN
            bonus := bonus + 500;
        END IF;
        ...
    EXCEPTION
        WHEN bonus_missing THEN
            ...
    END calc_bonus;
```

The actual parameter that corresponds to an IN OUT formal parameter must be a variable; it cannot be a constant or expression.

See also: Procedures, Functions, Actual vs Formal Parameters,
          Parameter Default Values

Parameter Default Values

As the example below shows, you can initialize IN parameters to default values. That way, you can pass different numbers of actual parameters to a subprogram, accepting or overriding the default values as you please. Moreover, you can add new formal parameters without having to change every call to the subprogram.

```
PROCEDURE create_dept
    (new_dname CHAR DEFAULT 'TEMP',
     new_loc   CHAR DEFAULT 'TEMP') IS
BEGIN
    INSERT INTO dept
        VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);
END create_dept;
```

If an actual parameter is not passed, the default value of its corresponding formal parameter is used. Consider the following calls to "create_dept":

```
BEGIN
    ...
    create_dept;
    create_dept('MARKETING');
    create_dept('MARKETING', 'NEW YORK');
    ...
END;
```

The first call passes no actual parameters, so both default values are used. The second call passes one actual parameter, so the default value for "new_loc" is used. The third call passes two actual parameters, so neither default value is used.

In most cases, you can use positional notation to override the default values of formal parameters. However, you cannot skip a formal parameter by leaving out its actual parameter. For example, the following call incorrectly associates the actual parameter 'NEW YORK' with the formal parameter "new_dname":

```
create_dept('NEW YORK');  -- incorrect
```

You cannot solve the problem by leaving a placeholder for the actual parameter. For example, the following call is illegal:

```
create_dept( , 'NEW YORK');  -- illegal
```

In such cases, you must use named notation, as follows:

```
create_dept(new_loc => 'NEW YORK');
```

See also: Procedures, Functions, Actual vs Formal Parameters,
Positional and Named Notation

Aliasing

To optimize execution, the PL/SQL compiler can choose different methods of parameter passing (copy or reference) for a subprogram call. The easy-to-avoid problem of aliasing occurs when the same actual parameter appears twice in a procedure call. Unless both formal parameters are IN parameters, the result is indeterminate because it depends on the methods of parameter passing chosen by the compiler. An example follows:

```
DECLARE
    str  CHAR(10);
    PROCEDURE reverse (in_str CHAR, out_str OUT CHAR) IS
        ...
    BEGIN
        -- reverse order of characters in string
        ...
        /* At this point, whether the value of in_str     *
         * is 'ABCD' or 'DCBA' depends on the methods of  *
         * parameter passing used by the PL/SQL compiler. */
    END reverse;
    ...
BEGIN
    str := 'ABCD';
    reverse(str, str);  -- indeterminate
    ...
END;
```

Aliasing also occurs when a global variable appears in a procedure call and then is referenced within the procedure. Consider the following example:

```
DECLARE
    rent  REAL;
    PROCEDURE raise_rent (increase IN OUT REAL) IS
        ...
    BEGIN
        rent := rent + increase;
        ...
    END raise_rent;
    ...
BEGIN
    ...
    raise_rent(rent);  -- indeterminate
END;
```

Again, the result is indeterminate.

See also: Procedures, Functions, Actual vs Formal Parameters

Overloading

PL/SQL lets you overload subprogram names. That is, you can use the same name for several different subprograms as long as their formal parameters differ in number, order, or datatype family.

Suppose you want to initialize the first "n" rows in two PL/SQL tables that were declared as follows:

```
DECLARE
    TYPE DateTabTyp IS TABLE OF DATE
        INDEX BY BINARY_INTEGER;
    TYPE RealTabTyp IS TABLE OF REAL
        INDEX BY BINARY_INTEGER;
    hiredate_tab  DateTabTyp;
    sal_tab       RealTabTyp;
    ...
```

You might write the following procedure to initialize the PL/SQL table named "hiredate_tab":

```
PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
BEGIN
    FOR i IN 1..n LOOP
        tab(i) := SYSDATE;
    END LOOP;
END initialize;
```

And, you might write the next procedure to initialize the PL/SQL table named "sal_tab":

```
PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
BEGIN
    FOR i IN 1..n LOOP
        tab(i) := 0.0;
    END LOOP;
END initialize;
```

Because the processing in these two procedures is the same, it is logical to give them the same name.

You can place the two overloaded "initialize" procedures in the same block, subprogram, or package. PL/SQL determines which of the two procedures is being called by checking their formal parameters. Consider the example below. If you call "initialize" with a "DateTabTyp" parameter, PL/SQL uses the first version of "initialize." But, if you call "initialize" with a "RealTabTyp" parameter, PL/SQL uses the second version.

```
DECLARE
    TYPE DateTabTyp IS TABLE OF DATE
        INDEX BY BINARY_INTEGER;
    TYPE RealTabTyp IS TABLE OF REAL
        INDEX BY BINARY_INTEGER;
    hiredate_tab  DateTabTyp;
    comm_tab      RealTabTyp;
    indx          BINARY_INTEGER;
    ...
BEGIN
    indx := 50;
    initialize(hiredate_tab, indx);  -- calls first version
    initialize(comm_tab, indx);      -- calls second version
    ...
END;
```

Restrictions

Only local or packaged subprograms can be overloaded. So, you cannot overload standalone subprograms.   Also, you cannot overload two subprograms if their formal parameters differ only in name or parameter mode.   For example, you cannot overload the following procedures:

```
PROCEDURE reconcile (acctno IN INTEGER) IS
BEGIN
    ...
END reconcile;

PROCEDURE reconcile (acctno OUT INTEGER) IS
BEGIN
    ...
END reconcile;
```

Furthermore, you cannot overload two subprograms if their formal parameters differ only in datatype and the different datatypes are in the same family.   For instance, you cannot overload the following procedures because the datatypes INTEGER and REAL are in the same family:

```
PROCEDURE charge_back (amount INTEGER) IS
BEGIN
    ...
END charge_back;

PROCEDURE charge_back (amount REAL) IS
BEGIN
    ...
END charge_back;
```

Finally, you cannot overload two functions that differ only in return type (the datatype of the result value) even if the types are in different families.   For example, you cannot overload the following functions:

```
FUNCTION acct_ok (acctno INTEGER) RETURN BOOLEAN IS
BEGIN
    ...
END acct_ok;

FUNCTION acct_ok (acctno INTEGER) RETURN INTEGER IS
BEGIN
    ...
END acct_ok;
```

See also: Subprograms, Actual vs Formal Parameters, Scope and Visibility

Recursion

Recursion is a powerful technique for simplifying the design of algorithms. Basically, recursion means self-reference. In a recursive mathematical sequence, each term is derived by applying a formula to preceding terms. The Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, ...) is an example. Each term in the sequence (after the second) is the sum of the two terms that immediately precede it.

In a recursive definition, something is defined in terms of simpler versions of itself. Consider the definition of "n" factorial (the product of all integers from 1 to "n"):

```
n! = n * (n - 1)!
```

A recursive subprogram is one that calls itself. Think of a recursive call as a call to some other subprogram that does the same task as your subprogram. Each recursive call creates a new instance of any objects declared in the subprogram, including parameters, variables, cursors, and exceptions. Likewise, new instances of SQL statements are created at each level in the recursive descent.

There must be at least two paths through a recursive subprogram: one that leads to the recursive call and one that does not. That is, at least one path must lead to a terminating condition. Otherwise, the recursion would (theoretically) go on forever. In practice, if a recursive subprogram strays into infinite regress, PL/SQL eventually runs out of memory and raises the predefined exception STORAGE_ERROR.

To solve some programming problems, you must repeat a sequence of statements until a condition is met. You can use iteration or recursion to solve such problems. Recursion is appropriate when the problem can be broken down into simpler versions of itself. For example, you can evaluate 3! as follows:

```
0! = 1
1! = 1 * 0! = 1 * 1 = 1
2! = 2 * 1! = 2 * 1 = 2
3! = 3 * 2! = 3 * 2 = 6
```

To implement this algorithm, you might write the following recursive function, which returns the factorial of a positive integer:

```
FUNCTION fac (n POSITIVE) RETURN INTEGER IS  -- returns n!
BEGIN
    IF n = 1 THEN  -- terminating condition
        RETURN 1;
    ELSE
        RETURN n * fac(n - 1);  -- recursive call
    END IF;
END fac;
```

At each recursive call, "n" is decremented. Eventually, "n" becomes 1 and the recursion stops.

See also: Subprograms, Mutual Recursion, Recursion vs Iteration

Mutual Recursion

Subprograms are mutually recursive if they directly or indirectly call each other. In the example below, the Boolean functions "odd" and "even," which determine whether a number is odd or even, call each other directly. The forward declaration of "odd" is necessary because "even" calls "odd," which is not yet declared when the call is made.

```
FUNCTION odd (n NATURAL) RETURN BOOLEAN;  -- forward declaration

FUNCTION even (n NATURAL) RETURN BOOLEAN IS
BEGIN
    IF n = 0 THEN
        RETURN TRUE;
    ELSE
        RETURN odd(n - 1);  -- mutually recursive call
    END IF;
END even;

FUNCTION odd (n NATURAL) RETURN BOOLEAN IS
BEGIN
    IF n = 0 THEN
        RETURN FALSE;
    ELSE
        RETURN even(n - 1);  -- mutually recursive call
    END IF;
END odd;
```

When a positive integer "n" is passed to "odd" or "even", the functions call each other by turns. At each call, "n" is decremented. Ultimately, "n" becomes zero and the final call returns TRUE or FALSE. For instance, passing the number 4 to "odd" results in this sequence of calls:

```
odd(4)
even(3)
odd(2)
even(1)
odd(0)  -- returns FALSE
```

On the other hand, passing the number 4 to "even" results in the following sequence of calls:

```
even(4)
odd(3)
even(2)
odd(1)
even(0)  -- returns TRUE
```

See also: Subprograms, Recursion, Recursion vs Iteration

Recursion vs Iteration

Unlike iteration, recursion is not essential to PL/SQL programming. Any problem that can be solved using recursion can be solved using iteration. Furthermore, the concept of iteration is easier to grasp because examples of recursion are uncommon in everyday life. Consequently, the iterative version of a subprogram is usually easier to design than the recursive version. However, the recursive version is usually simpler, smaller, and therefore easier to debug. Compare the following functions, which compute the "n"th number in the Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, 21, ...):

```
-- recursive version
FUNCTION fib (n POSITIVE) RETURN INTEGER IS
BEGIN
    IF (n = 1) OR (n = 2) THEN
        RETURN 1;
    ELSE
        RETURN fib(n - 1) + fib(n - 2);
    END IF;
END fib;

-- iterative version
FUNCTION fib (n POSITIVE) RETURN INTEGER IS
    pos1  INTEGER := 1;
    pos2  INTEGER := 0;
    cum   INTEGER;
BEGIN
    IF (n = 1) OR (n = 2) THEN
        RETURN 1;
    ELSE
        cum := pos1 + pos2;
        FOR i IN 3..n LOOP
            pos2 := pos1;
            pos1 := cum;
            cum := pos1 + pos2;
        END LOOP;
        RETURN cum;
    END IF;
END fib;
```

The recursive version of "fib" is more elegant than the iterative version. However, the iterative version is more efficient; it runs faster and uses less storage. That is because each recursive call requires additional time and memory. As the number of recursive calls gets larger, so does the difference in efficiency. Still, if you expect the number of recursive calls to be small, you might choose the recursive version for its readability.

See also: Subprograms, Recursion, Mutual Recursion

Stored Subprograms

Subprograms (procedures and functions) can be compiled separately and stored permanently in an Oracle database, ready to be executed. A subprogram explicitly CREATEd using an Oracle tool is called a "stored subprogram."

You can issue the CREATE PROCEDURE and CREATE FUNCTION statements interactively from SQL*Plus or SQL*DBA. For example, you might create a procedure named "fire_employee" as follows:

```
CREATE PROCEDURE fire_employee (emp_id NUMBER) AS
BEGIN
    DELETE FROM emp WHERE empno = emp_id;
END;
```

Once compiled and stored in the data dictionary, a subprogram is a database object, which can be referenced by any number of applications connected to that database.

Stored subprograms defined within a package are called "packaged" subprograms; those defined independently are called "standalone" subprograms.

You can call stored subprograms from a database trigger, another stored subprogram, an Oracle Precompiler application, an OCI application, or interactively from SQL*Plus or SQL*DBA. For example, you might call the standalone procedure "fire_employee" from SQL*Plus, as follows:

```
SQL> EXECUTE fire_employee(7499);
```

Subprograms are stored in parsed, compiled form. So, when called, they are loaded and passed to the PL/SQL engine immediately. Also, stored subprograms take advantage of the Oracle shared memory capability. Only one copy of a subprogram need be loaded into memory for execution by many users.

See also: Subprograms, Creating Stored Subprograms, Calling Stored Subprograms, Calling Stored Functions from SQL, CREATE FUNCTION, CREATE PROCEDURE

Creating Stored Subprograms

You can CREATE subprograms and store them permanently in an Oracle database for general use.   You can issue the CREATE PROCEDURE and CREATE FUNCTION statements interactively from SQL*Plus or SQL*DBA. For example, you might CREATE the procedure "fire_employee" as follows:

```
    CREATE PROCEDURE fire_employee (emp_id NUMBER) AS
    BEGIN
        DELETE FROM emp WHERE empno = emp_id;
    END;
```

Furthermore, you can embed the SQL statements CREATE FUNCTION and CREATE PROCEDURE in a host program, as the following example shows:

```
    EXEC SQL CREATE OR REPLACE
        FUNCTION sal_ok (salary REAL, title CHAR) RETURN BOOLEAN AS
            min_sal  REAL;
            max_sal  REAL;
        BEGIN
            SELECT losal, hisal INTO min_sal, max_sal FROM sals
                WHERE job = title;
            RETURN (salary >= min_sal) AND (salary <= max_sal);
        END sal_ok;
    END-EXEC;
```

The embedded CREATE FUNCTION and CREATE PROCEDURE statements are hybrids.   Like all other embedded CREATE statements, they begin with the keywords EXEC SQL.   But, unlike other embedded CREATE statements, they end with the PL/SQL terminator END-EXEC.

You specify the REPLACE clause in the CREATE statement to redefine an existing subprogram without having to drop the subprogram, recreate it, and regrant privileges on it.   Note, however, that privileges granted to roles cannot be used to create stored subprograms.

When you CREATE a subprogram for storage in the database, Oracle automatically compiles the source code, caches the object code in a "shared SQL area" in the System Global Area (SGA), and stores the source and object code in the data dictionary.   The object code stays cached in the SGA, where it can be executed quickly.   When necessary, Oracle applies a least-recently-used algorithm that selects shared SQL areas to be flushed from the SGA to make room for others.

See also: Subprograms, Stored Subprograms, Calling Stored Subprograms,
          CREATE FUNCTION, CREATE PROCEDURE

Calling Stored Subprograms

You can call stored subprograms from a database trigger, another stored subprogram, an Oracle Precompiler application, an OCI application, or an Oracle tool such as SQL*Plus.   Some examples follow.

A stored subprogram can call another stored subprogram.   For instance, the following call to the standalone procedure "create_dept" might appear in the body of a packaged subprogram:

```
    create_dept(name, location);
```

An Oracle Precompiler or OCI application can call stored subprograms using anonymous PL/SQL blocks.   In the following example, you call the standalone procedure "create_dept" from an Oracle Precompiler program:

```
    EXEC SQL EXECUTE
        BEGIN
            create_dept(:name, :location);
        END;
    END-EXEC;
```

The actual parameters "name" and "location" are host variables.
In the next example, the procedure "create_dept" is part of a package named "emp_actions," so you must use dot notation to qualify the procedure call:

```
    EXEC SQL EXECUTE
        BEGIN
            emp_actions.create_dept(:name, :location);
        END;
    END-EXEC;
```

You can call stored subprograms interactively from Oracle tools such as SQL*Plus and SQL*DBA.   For example, you might call the standalone procedure "create_dept" from SQL*Plus as follows:

```
    SQL> EXECUTE create_dept('MARKETING', 'NEW YORK');
```

This call is equivalent to the following call issued from an anonymous PL/SQL block:

```
    SQL> BEGIN create_dept('MARKETING', 'NEW YORK'); END;
```

Remote Access

You can use the following syntax to call standalone and packaged subprograms stored in a remote Oracle database:

```
    procedure_name@db_link(param1, param2, ...);
    package_name.procedure_name@db_link(param1, param2, ...);
```

In the next example, you call the stored procedure "raise_salary," which is defined in the package "emp_actions" in the "newyork" database:

```
    BEGIN
        emp_actions.raise_salary@newyork(emp_num, amount);
```

You can create synonyms to provide location transparency for remote standalone (but not packaged) subprograms, as the following example shows:

```
    CREATE SYNONYM create_dept FOR create_dept@newyork;
```

See also: Subprograms, Stored Subprograms, Creating Stored Subprograms, Calling Stored Functions from SQL, CREATE SYNONYM

Calling Stored Functions from SQL

A stored function is a user-defined PL/SQL function created with an Oracle tool and stored in the data dictionary. It is a database object, which can be referenced by any number of applications connected to that database. There are two types of stored functions: packaged and standalone. Packaged functions are defined within a PL/SQL package; standalone functions are defined independently.

Unlike functions, which are called as part of an expression, procedures are called as statements. Therefore, procedures cannot be called directly from SQL expressions. However, you can call stored functions from the SELECT, VALUES, SET, WHERE, START WITH, GROUP BY, HAVING, and ORDER BY clauses--wherever expressions are allowed in a SQL statement.

Calling Syntax

To call a stored function from a SQL expression, you use the following syntax:

```
    [schema.][package.]function[@dblink][(arg[, arg] ... )]
```

You must write the arguments (actual parameters) using positional notation; you cannot use named or mixed notation. In the following example, you call the standalone function "gross_pay," which is stored in a remote Oracle database:

```
    SELECT gross_pay@newyork(eenum,stime,otime) INTO pay FROM dual;
```

Using Default Values

The stored function "gross_pay" initializes two of its the formal parameters to default values using the DEFAULT clause, as follows:

```
    CREATE FUNCTION gross_pay
          (emp_id IN NUMBER,
           st_hrs IN NUMBER DEFAULT 40,
           ot_hrs IN NUMBER DEFAULT 0) RETURN NUMBER AS
       ...
```

When calling "gross_pay" from a procedural statement, you can always accept the default value of "st_hrs." That is because you can use named notation, which lets you skip parameters, as in

```
    IF gross_pay(eenum,ot_hrs => otime) > pay_limit THEN ...
```

However, when calling "gross_pay" from a SQL expression, you cannot accept the default value of "st_hrs" unless you accept the default value of "ot_hrs." That is because you cannot use named notation.

Meeting Basic Requirements

To be callable from SQL expressions, a user-defined PL/SQL function
must meet the following basic requirements:

```
* It must be a stored function, not a function defined within
  a PL/SQL block or subprogram.

* It must be a row function, not a column (group) function;
  that is, it cannot take an entire column of data as its
  argument.

* All its formal parameters must be IN parameters; none can be
  an OUT or IN OUT parameter.

* The datatypes of its formal parameters must be Oracle Server
  internal types such as CHAR, DATE, or NUMBER, not PL/SQL
  types such as BOOLEAN, RECORD, or TABLE.

* Its return type (the datatype of its result value) must
  be an Oracle Server internal type.
```

For example, the following stored function meets the basic requirements:

```
CREATE FUNCTION gross_pay
      (emp_id IN NUMBER,
       st_hrs IN NUMBER DEFAULT 40,
       ot_hrs IN NUMBER DEFAULT 0) RETURN NUMBER AS
   st_rate  NUMBER;
   ot_rate  NUMBER;
BEGIN
   SELECT srate, orate INTO st_rate, ot_rate FROM payroll
      WHERE acctno = emp_id;
   RETURN st_hrs * st_rate + ot_hrs * ot_rate;
END gross_pay;
```

Controlling Side Effects

To execute a SQL statement that calls a stored function, the Oracle
Server must know the "purity level" of the function. That is, the
extent to which the function is free of side effects. In this
context, "side effects" are references to database tables or packaged
variables.

Side effects can prevent the parallelization of a query, yield order-dependent (and therefore indeterminate) results, or require that
package state be maintained across user sessions (which is not allowed).
Therefore, the following rules apply to stored functions called from
SQL expressions:

```
* The function cannot modify database tables; therefore, it
  cannot execute an INSERT, UPDATE, or DELETE statement.

* Remote or parallelized functions cannot read or write the
```

```
   values of packaged variables.

 * Only functions called from a SELECT, VALUES, or SET clause
   can write the values of packaged variables.

 * The function cannot call another subprogram that breaks
   one of the foregoing rules.  Also, the function cannot reference
   a database view that breaks one of the foregoing rules.  (Oracle
   replaces references to a view with a stored SELECT operation,
   which can include function calls.)
```

For standalone functions, Oracle can enforce these rules by checking the function body. However, the body of a packaged function is hidden; only its specification is visible. So, for packaged functions, you must use the pragma (compiler directive) RESTRICT_REFERENCES to enforce the rules.

The pragma tells the PL/SQL compiler to deny the packaged function read/write access to database tables, packaged variables, or both. If you compile a function body that violates the pragma, you get a compilation error.

Calling Packaged Functions

To call a packaged function from SQL expressions, you must assert its purity level by coding the pragma RESTRICT_REFERENCES in the package specification (not in the package body). The pragma must follow the function declaration but need not follow it immediately. Only one pragma can reference a given function declaration.

To code the pragma RESTRICT_REFERENCES, you use the syntax

```
PRAGMA RESTRICT_REFERENCES (
   function_name, WNDS [, WNPS] [, RNDS] [, RNPS]);
```

where:

```
WNDS   means "writes no database state" (does not modify
       database tables).

WNPS   means "writes no package state" (does not change the
       values of packaged variables).

RNDS   means "reads no database state" (does not query database
       tables).

RNPS   means "reads no package state" (does not reference the
       values of packaged variables).
```

You can pass the arguments in any order but you must pass the argument WNDS. No argument implies another. For instance, RNPS does not imply WNPS.

In the example below, the function "compound" neither reads nor writes database or package state, so you can assert the maximum purity level. Always assert the highest purity level that a function allows. That way, the PL/SQL compiler will never reject the function unnecessarily.

```
CREATE PACKAGE finance AS  -- package specification
   interest  REAL;  -- public packaged variable
   ...
   FUNCTION compound
         (years  IN NUMBER,
          amount IN NUMBER,
          rate   IN NUMBER) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES (compound, WNDS, WNPS, RNDS, RNPS);
END finance;

CREATE PACKAGE BODY finance AS  --package body
   FUNCTION compound
         (years  IN NUMBER,
          amount IN NUMBER,
          rate   IN NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN amount * POWER((rate / 100) + 1, years);
   END compound;
   ...          -- no pragma in package body
END finance;
```

Referencing Packages with an Initialization Part

Packages can have an initialization part, which is hidden in the package body. Typically, the initialization part holds statements that initialize public variables. In the following example, the SELECT statement initializes the public variable "prime_rate":

```
CREATE PACKAGE loans AS
   prime_rate  REAL;
   ...
END loans;

CREATE PACKAGE BODY loans AS
   ...
BEGIN  -- initialization part
   SELECT prime INTO prime_rate FROM rates;
END loans;
```

The initialization code is run only once--the first time the package is referenced. If the code reads or writes database state or package state other than its own, it can cause side effects. Moreover, a stored function that references the package (and thereby runs the initialization code) can cause side effects indirectly. So, to call the function from SQL expressions, you must use the pragma RESTRICT_REFERENCES to assert or imply the purity level of the initialization code.

To assert the purity level of the initialization code, you use a variant of the pragma RESTRICT_REFERENCES, in which the function name is replaced by a package name. You code the pragma in the package specification, where it is visible to other users. That way, anyone referencing the package can see the restrictions and conform to them.

To code the variant pragma RESTRICT_REFERENCES, you use the syntax

```
PRAGMA RESTRICT_REFERENCES (
   package_name, WNDS [, WNPS] [, RNDS] [, RNPS]);
```

where the arguments WNDS, WNPS, RNDS, and RNPS have the usual meaning. In the example below, the initialization code reads database state and writes package state. However, you can assert WNPS because the code is writing the state of its own package, which is permitted. So, you assert WNDS, WNPS, RNPS--the highest purity level the function allows. (If the public variable "prime_rate" were in another package, you could not assert WNPS.)

```
CREATE PACKAGE loans AS
   PRAGMA RESTRICT_REFERENCES (loans, WNDS, WNPS, RNPS);
   prime_rate  REAL;
   ...
END loans;
...
CREATE PACKAGE BODY loans AS
   ...
BEGIN  -- initialization part
   SELECT prime INTO prime_rate FROM rates;
END loans;
```

You can place the pragma anywhere in the package specification, but placing it at the top (where it stands out) is a good idea.

To imply the purity level of the initialization code, your package must have a RESTRICT_REFERENCES pragma for one of the functions it declares. From the pragma, Oracle can infer the purity level of the initialization code (because the code cannot break any rule enforced by a pragma). In the next example, the pragma for the function "discount" implies that the purity level of the initialization code is at least WNDS:

```
CREATE PACKAGE loans AS
   ...
   FUNCTION discount (...) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES (discount, WNDS);
END loans;
...
```

To draw an inference, Oracle can combine the assertions of all RESTRICT_REFERENCES pragmas. For example, the following pragmas (combined) imply that the purity level of the initialization code is at least WNDS, RNDS:

```
CREATE PACKAGE loans AS
   ...
   FUNCTION discount (...) RETURN NUMBER;
   FUNCTION credit_ok (...) RETURN CHAR;
   PRAGMA RESTRICT_REFERENCES (discount, WNDS);
   PRAGMA RESTRICT_REFERENCES (credit_ok, RNDS);
END loans;
...
```

Avoiding Problems

To call a packaged function from SQL expressions, you must assert its purity level using the pragma RESTRICT_REFERENCES. However, if the package has an initialization part, the PL/SQL compiler might not let you assert the highest purity level the function allows. As a result, you might be unable to call the function remotely, in parallel, or from certain SQL clauses.

This happens when a packaged function is purer than the package initialization code. Remember, the first time a package is referenced, its initialization code is run. If that reference is a function call, any additional side effects caused by the initialization code occur during the call. So, in effect, the initialization code lowers the purity level of the function.

To avoid this problem, move the package initialization code into a subprogram. That way, your application can run the code explicitly (rather than implicitly during package instantiation) without affecting your packaged functions.

Name Precedence

In SQL statements, the names of database columns take precedence over the names of parameterless functions. For example, if user "scott" executes the statements

```
CREATE TABLE stats (rand_num NUMBER, ...);
CREATE FUNCTION rand_num RETURN NUMBER AS ...
```

then the following select-item refers to the column "rand_num":

```
SELECT rand_num, ... INTO start_val, ...
    FROM stats WHERE ...
```

In this case, to call the stored function "rand_num," you must specify the schema, as follows:

```
SELECT scott.rand_num, ... INTO start_val, ...
    FROM stats WHERE ...
```

Overloading

PL/SQL lets you overload packaged (but not standalone) functions.

That is, you can use the same name for different functions if their formal parameters differ in number, order, or datatype family.

However, a RESTRICT_REFERENCES pragma can apply to only one function declaration. So, a pragma that references the name of overloaded functions always applies to the nearest foregoing function declaration. In the following example, the pragma applies to the second declaration of "valid":

```
CREATE PACKAGE tests AS
   -- these functions return 'T' or 'F' (for TRUE or FALSE)
   FUNCTION valid (x NUMBER) RETURN CHAR;
   FUNCTION valid (x DATE) RETURN CHAR;
   PRAGMA RESTRICT_REFERENCES (valid, WNDS);
   ...
END tests;
```

See also: Subprograms, Stored Subprograms, Creating Stored Subprograms, Calling Stored Subprograms

Packages

A package is a database object that groups logically related PL/SQL types, objects, and subprograms. Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary. The specification is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms and so implements the specification.

Unlike subprograms, packages cannot be called, passed parameters, or nested. Still, the format of a package is similar to that of a subprogram:

```
PACKAGE name IS  -- specification (visible part)
    -- public type and object declarations
    -- subprogram specifications
END [name];

PACKAGE BODY name IS  -- body (hidden part)
    -- private type and object declarations
    -- subprogram bodies
[BEGIN
    -- initialization statements]
END [name];
```

The specification holds public declarations, which are visible to your application. The body holds implementation details and private declarations, which are hidden from your application.

You can debug, enhance, or replace a package body without changing the interface (package specification) to the package body.

Packages can be created interactively with SQL*Plus or SQL*DBA using the CREATE PACKAGE and CREATE PACKAGE BODY commands. In the following example, a record type, a cursor, and two employment procedures are packaged:

```
CREATE PACKAGE emp_actions AS  -- specification
    TYPE EmpRecTyp IS RECORD (emp_id INTEGER, salary REAL);
    CURSOR desc_salary RETURN EmpRecTyp;

    PROCEDURE hire_employee
        (ename   CHAR,
         job     CHAR,
         mgr     NUMBER,
         sal     NUMBER,
         comm    NUMBER,
         deptno  NUMBER);

    PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

CREATE 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   CHAR,
             job     CHAR,
             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;
```

Notice that the procedure "hire_employee" uses the database sequence "empno_seq" and the function SYSDATE to INSERT a new employee number and hire date, respectively.

Only the declarations in the package specification are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible. So, you can change the body without having to recompile calling programs.

See also: Package Specification, Package Body, CREATE PACKAGE, CREATE PACKAGE BODY

Package Specification

The package specification contains public declarations.   The scope of these declarations is local to your database schema and global to the package.   So, the declared objects are accessible from your application and from anywhere in the package.

The specification lists the package resources available to applications. All the information your application needs to use the resources is in the specification.   For example, the following declaration shows that the function named "factorial" takes one parameter of type INTEGER and returns a value of type INTEGER:

```
    FUNCTION factorial (n INTEGER) RETURN INTEGER;  -- returns n!
```

That is all the information you need to call the function.   You need not consider the underlying implementation of "factorial" (whether it is iterative or recursive, for example).

Only subprograms and cursors have an underlying implementation or "definition."   So, if a specification declares only types, constants, variables, and exceptions, the package body is unnecessary.   An example of such a package follows:

```
    -- package consisting of a specification only
    PACKAGE trans_data IS
        TYPE TimeTyp IS RECORD
            (minute  SMALLINT,
             hour    SMALLINT);
        TYPE TransTyp IS RECORD
            (category  VARCHAR2,
             account   INTEGER,
             amount    REAL,
             time      TimeTyp);
        minimum_balance     CONSTANT REAL := 10.00;
        number_processed    INTEGER;
        insufficient_funds  EXCEPTION;
    END trans_data;
```

The package "trans_data" needs no body because types, constants, variables, and exceptions do not have an underlying implementation. Such packages let you define global variables (usable by subprograms and triggers) that persist throughout a session.

`See also:` Packages, Package Body, CREATE PACKAGE

Package Body

The package body implements the package specification. That is, the package body contains the definition of every cursor and subprogram declared in the package specification. Keep in mind that subprograms defined in a package body are accessible outside the package only if their specifications also appear in the package specification.

The package body can also contain private declarations, which define types and objects necessary for the internal workings of the package. The scope of these declarations is local to the package body. Therefore, the declared types and objects are inaccessible except from within the package body. Unlike a package specification, the declarative part of a package body can contain subprogram bodies.

Following the declarative part of a package body is the optional initialization part, which typically holds statements that initialize some of the variables previously declared in the package. The initialization part of a package plays a minor role because, unlike subprograms, a package cannot be called or passed parameters. As a result, the initialization part of a package is run only once, the first time you reference the package.

Consider the package below named "emp_actions." After writing the package, you can develop applications that reference its types, call its subprograms, use its cursor, or raise its exception. When you CREATE the package, it is stored in an Oracle database for general use.

```
PACKAGE emp_actions IS
    /* Declare externally visible types, cursor, exception. */
    TYPE EmpRecTyp IS RECORD (emp_id INTEGER, salary REAL);
    TYPE DeptRecTyp IS RECORD (dept_id INTEGER, location CHAR);
    CURSOR desc_salary RETURN EmpRecTyp;
    salary_missing  EXCEPTION;
    /* Declare externally callable subprograms. */
    FUNCTION hire_employee
        (ename   CHAR,
         job     CHAR,
         mgr     INTEGER,
         sal     NUMBER,
         comm    NUMBER,
         deptno  INTEGER) RETURN INTEGER;
    PROCEDURE fire_employee (emp_id INTEGER);
    PROCEDURE raise_salary (emp_id INTEGER, increase NUMBER);
    FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp;
END emp_actions;

PACKAGE BODY emp_actions IS
    number_hired  INTEGER;  -- visible only in this package

    /* Fully define cursor specified in package. */
    CURSOR desc_salary RETURN EmpRecTyp IS
        SELECT empno, sal FROM emp ORDER BY sal DESC;
```

```
/* Fully define subprograms specified in package. */

FUNCTION hire_employee
     (ename   CHAR,
      job     CHAR,
      mgr     INTEGER,
      sal     NUMBER,
      comm    NUMBER,
      deptno  INTEGER) RETURN INTEGER IS
    new_empno  INTEGER;
BEGIN
    SELECT empno_seq.NEXTVAL INTO new_empno FROM DUAL;
    INSERT INTO emp VALUES (new_empno, ename, job,
        mgr, SYSDATE, sal, comm, deptno);
    number_hired := number_hired + 1;
    RETURN new_empno;
END hire_employee;

PROCEDURE fire_employee (emp_id INTEGER) IS
BEGIN
    DELETE FROM emp WHERE empno = emp_id;
END fire_employee;

PROCEDURE raise_salary (emp_id INTEGER, increase NUMBER) IS
    current_salary  NUMBER;
BEGIN
    SELECT sal INTO current_salary FROM emp
        WHERE empno = emp_id;
    IF current_salary IS NULL THEN
        RAISE salary_missing;
    ELSE
        UPDATE emp SET sal = sal + increase
            WHERE empno = emp_id;
    END IF;
END raise_salary;

FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp IS
    emp_rec  EmpRecTyp;
BEGIN
    OPEN desc_salary;
    FOR i IN 1..n LOOP
        FETCH desc_salary INTO emp_rec;
    END LOOP;
    CLOSE desc_salary;
    RETURN emp_rec;
END nth_highest_salary;

/* Define local function, available only in package. */
FUNCTION rank (emp_id INTEGER, job_title CHAR)
    RETURN INTEGER IS
/* Return rank (highest = 1) of employee in a given *
 * job classification based on performance rating.  */
```

```
        head_count  INTEGER;
        score       NUMBER;
    BEGIN
        SELECT COUNT(*) INTO head_count FROM emp
            WHERE job = job_title;
        SELECT rating INTO score FROM reviews
            WHERE empno = emp_id;
        score := score / 100;  -- maximum score is 100
        RETURN (head_count + 1) - ROUND(head_count * score);
    END rank;
BEGIN  -- initialization part starts here
    INSERT INTO emp_audit VALUES (SYSDATE, USER, 'EMP_ACTIONS');
    number_hired := 0;
END emp_actions;
```

Remember, the initialization part of a package is run just once, the first time you reference the package.   So, in the last example, only one row is inserted into the database table "emp_audit."   Likewise, the variable "number_hired" is initialized only once.   Every time the procedure "hire_employee" is called, the variable "number_hired" is updated.   However, the count kept by "number_hired" is session-specific. That is, the count reflects the number of new employees processed by one user, not the number processed by all users.

See also: Packages, Package Specification, CREATE PACKAGE BODY

Referencing Package Contents

To reference the types, objects, and subprograms declared within a package specification, you use dot notation as follows:

```
package_name.type_name
package_name.object_name
package_name.subprogram_name
```

You can reference package contents from database triggers, stored subprograms, embedded PL/SQL blocks, and anonymous PL/SQL blocks sent to Oracle interactively by SQL*Plus or SQL*DBA.   In the following example, you reference the packaged variable "minimum_balance," which is declared in the package "trans_data":

```
DECLARE
    new_balance  REAL;
    ...
BEGIN
    ...
    IF new_balance < trans_data.minimum_balance THEN
        ...
    END IF;
    ...
```

See also: Packages, Package Specification

Private vs Public Objects

Unlike items declared in the package specification, items declared in the body are restricted to use within the package.   PL/SQL code outside the package cannot reference such items, so they are termed "private." However, items declared in the package specification are visible outside the package.   Any PL/SQL code can reference such items, so they are termed "public."

When you must maintain items throughout a session or across transactions, place them in the declarative part of the package body. Remember, however, that the values of such items are session-specific. If, in addition, you must make the items public, place them in the package specification, where they are available for general use.

See also: Packages, Package Specification, Package Body

Calling Packaged Subprograms

Packaged subprograms must be referenced using dot notation, as the following example shows:

```
    emp_actions.hire_employee(name, title, ...);
```

This tells the PL/SQL compiler that the procedure "hire_employee" is found in the package "emp_actions."

You can call packaged subprograms from a database trigger, another stored subprogram, an Oracle Precompiler application, an OCI application, or an Oracle tool such as SQL*Plus.   Some examples follow.

A stored subprogram can call a packaged subprogram.   For instance, the following call to the packaged procedure "hire_employee" might appear in a standalone subprogram:

```
    emp_actions.hire_employee(name, title, ...);
```

An Oracle Precompiler or OCI application can call packaged subprograms using anonymous PL/SQL blocks.   In the following example, you call the packaged procedure "hire_employee" from an Oracle Precompiler program:

```
    EXEC SQL EXECUTE
        BEGIN
            emp_actions.hire_employee(:name, :title, ...);
        END;
    END-EXEC;
```

The actual parameters "name" and "title" are host variables.

You can call packaged subprograms interactively from Oracle tools such as SQL*Plus and SQL*DBA.   For example, you might call the packaged procedure "hire_employee" from SQL*Plus as follows:

```
    SQL> EXECUTE emp.actions.hire_employee('TATE', 'CLERK', ...);
```

Remote Access

You can use the following syntax to call packaged subprograms stored in a remote Oracle database:

```
    package_name.subprogram_name@db_link(param1, param2, ...);
```

In the following example, you call the packaged procedure "hire_employee" in the "newyork" database:

```
    emp_actions.hire_employee@newyork(name, title, ...);
```

See also: Packages, Subprograms, Stored Subprograms

Package State and Dependency

A package specification is always in one of two states: valid or invalid. A package specification is valid if neither its source code nor any object it references has been DROPped, REPLACEd, or ALTERed since the package specification was last compiled.

On the other hand, a package specification is invalid if its source code or any object it references has been DROPped, REPLACEd, or ALTERed since the package specification was last compiled. When it invalidates a package specification, Oracle also invalidates any objects that reference the package.

A package body is subject to the same rules, except that Oracle can recompile a package body without invalidating its corresponding package specification. This feature limits the extent to which invalidations result in cascading recompilations.

Session Characteristics

Variables, constants, and cursors declared in a package have the following unique characteristics:

```
* Every session has its own set of packaged variables,
  constants, and cursors.
* In a session, the first time you reference a package, its
  variables and cursor parameters are null unless you initialize
  them.
* During a session, the package user can change the values of
  packaged variables and cursor parameters.  When a session ends,
  the values are lost and must be reinitialized when the next
  session begins.
```

Declare variables and cursors in a package only when you want them to persist throughout a session.

Dependency

When a package specification is recompiled, Oracle invalidates dependent objects. These include standalone or packaged subprograms that call or reference objects declared in the recompiled package specification. If you call or reference a dependent object before it is recompiled, Oracle automatically recompiles it at run time.

When a package body is recompiled, Oracle determines if objects on which the package body depends are valid. These include standalone subprograms and package specifications called or referenced by a procedure or cursor defined in the recompiled package body. If any of these objects are invalid, Oracle recompiles them before recompiling the package body. If Oracle can recompile it successfully, the package body becomes valid. Otherwise, Oracle returns a runtime error and the package body remains invalid. Compilation errors are stored in the data dictionary with the package.

Oracle stores the package specification and body separately in the data dictionary.   Other objects that call or reference global package objects depend only on the package specification.   Therefore, you can redefine objects in the package body (which causes the body to be recompiled) without causing Oracle to invalidate their dependent objects.

See also: Packages, Package Specification, Package Body

Package STANDARD

A package named STANDARD defines the PL/SQL environment.   The package specification globally declares types, exceptions, and subprograms, which are available automatically to every PL/SQL program.   For example, package STANDARD declares the following built-in function named ABS, which returns the absolute value of its argument:

```
    FUNCTION ABS (n NUMBER) RETURN NUMBER;
```

The contents of package STANDARD are directly visible to applications. So, you can call ABS from a database trigger, a stored subprogram, an Oracle Precompiler application, an OCI application, and a variety of Oracle tools including SQL*Plus, SQL*DBA, Oracle Forms, and Oracle Reports.

If you redeclare ABS in a PL/SQL program, your local declaration overrides the global declaration.   However, you can still call the built-in function by using dot notation, as follows:

```
    ... STANDARD.ABS(x) ...
```

Most built-in functions are overloaded.   For example, package STANDARD contains the following declarations:

```
    FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
    FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
    FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
    FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

PL/SQL resolves a call to TO_CHAR by matching the number and datatypes of the formal and actual parameters.

See also: Packages, Product-specific Packages, Overloading

Product-specific Packages

To help you build PL/SQL-based applications, the Oracle Server and several Oracle tools supply packages containing product-specific subprograms. For instance, the Oracle Server supplies (among others) the packages DBMS_STANDARD and DBMS_SQL.

Package DBMS_STANDARD provides language facilities that help your application interact with Oracle. For example, this package provides a procedure named "raise_application_error," which lets you issue user-defined error messages. That way, you can report errors to an application and avoid returning unhandled exceptions.

Package DBMS_SQL allows PL/SQL to execute SQL data definition and data manipulation statements dynamically at run time. For example, when called, the following stored procedure drops a specified database table:

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS
   cid INTEGER;
BEGIN
   /* Open new cursor and return cursor ID. */
   cid := dbms_sql.open_cursor;
   /* Parse and immediately execute dynamic SQL statement built by
      concatenating table name to DROP TABLE command.  (Unlike DML
      statements, DDL statements are executed at parse time.) */
   dbms_sql.parse(cid, 'DROP TABLE ' || table_name, dbms_sql.v7);
   /* Close cursor. */
   dbms_sql.close_cursor(cid);
EXCEPTION
   /* If an exception is raised, close cursor before exiting. */
   WHEN OTHERS THEN
      dbms_sql.close_cursor(cid);
END drop_table;
```

See also: Packages, Package STANDARD, raise_application_error Procedure

raise_application_error Procedure

A package named DBMS_STANDARD provides language facilities that help your application interact with Oracle. This package provides a procedure named "raise_application_error," which lets you issue user-defined error messages. That way, you can report errors to an application and avoid returning unhandled exceptions. The calling syntax is

```
    raise_application_error(error_number, message [, TRUE | FALSE]);
```

where "error_number" is a negative integer in the range -20000 .. -20999 and "message" is a character string up to 2048 bytes in length. If the optional third parameter is TRUE, the error is placed on the stack of previous errors. If the parameter is FALSE (the default), the error replaces all previous errors.

Package DBMS_STANDARD is an extension of package STANDARD, so you need not qualify references to it.

An application can call "raise_application_error" only from an executing stored subprogram. When called, "raise_application_error" ends the subprogram, rolls back any database changes it made, and returns a user-defined error number and message to the application. The error number and message can be trapped like any Oracle error. An example follows:

```
    PROCEDURE raise_salary (emp_id NUMBER, increase NUMBER) IS
        current_salary  NUMBER;
    BEGIN
        SELECT sal INTO current_salary FROM emp
            WHERE empno = emp_id;
        IF current_salary IS NULL THEN
            raise_application_error(-20101, 'Salary is missing');
        ELSE
            UPDATE emp SET sal = current_salary + increase
                WHERE empno = emp_id;
        END IF;
    END raise_salary;
```

The calling application gets a PL/SQL exception, which it can process using the error-reporting functions SQLCODE and SQLERRM in an OTHERS handler. Furthermore, it can use EXCEPTION_INIT to map specific error numbers returned by "raise_application_error" to exceptions of its own, as follows:

```
    EXEC SQL EXECUTE
        DECLARE
            ...
            null_salary  EXCEPTION;
            PRAGMA EXCEPTION_INIT(null_salary, -20101);
        BEGIN
            ...
            raise_salary(:emp_number, :amount);
        EXCEPTION
            WHEN null_salary THEN
                INSERT INTO emp_audit VALUES (:emp_number, ...);
            ...
        END;
    END-EXEC;
```

This technique allows the calling application to handle error conditions in specific exception handlers.   Typically, "raise_application_error" is used in database triggers.

See also: Procedures, Database Triggers, Packages, Package STANDARD