

# PL/SQL - QUICK GUIDE

[http://www.tutorialspoint.com/plsql/plsql\\_quick\\_guide.htm](http://www.tutorialspoint.com/plsql/plsql_quick_guide.htm)

Copyright © tutorialspoint.com

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are notable facts about PL/SQL:

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line SQL\*Plus interface.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in TimesTen in-memory database and IBM DB2.

## PL/SQL - ENVIRONMENT SETUP

PL/SQL is not a stand-alone programming language; it is a tool within the Oracle programming environment. SQL\*Plus is an interactive tool that allows you to type SQL and PL/SQL statements at the command prompt. These commands are then sent to the database for processing. Once the statements are processed, the results are sent back and displayed on screen.







To run PL/SQL programs you should have Oracle RDBMS Server installed in your machine which will take care of executing SQL commands. Most recent version of Oracle RDBMS is 11g. You can download a trial version of Oracle 11g from the following link:

[Download Oracle 11g Express Edition](#)

You will have to download either 32bit or 64 bit version of the installation as per your operating system. Usually there are two files, as I have downloaded for 64 bit Windows7. You will also use similar steps on your operating system, does not matter if it is Linux or Solaris.

- **win64\_11gR2\_database\_1of2.zip**
- **win64\_11gR2\_database\_2of2.zip**

After downloading above two files, you will need to unzip them in a single directory **database** and under that you will find following sub-directories:

 doc	3/24/2010 12:15 AM	File folder	
 install	3/30/2010 8:05 AM	File folder	
 response	3/30/2010 9:31 AM	File folder	
 stage	3/30/2010 9:31 AM	File folder	
 setup	3/12/2010 1:11 AM	Application	334 KB
 welcome	3/16/2010 1:42 PM	HTML Document	6 KB

Finally click on **setup** file to start the installation and follow the given steps till the end. If everything has been done successfully then its time to verify your installation. At your command prompt use the following command if you are using Windows:

```
sqlplus "/ as sysdba"
```

If everything is fine, you should have SQL prompt where you will write your PL/SQL commands and scripts:

## Text Editor

Running large programs from command prompt may land you in inadvertently losing some of the work. So a better option is to use command files. To use the command files:

- Type your code in a text editor, like Notepad, Notepad+, or EditPlus etc.
- Save the file with the .sql extension in the home directory.
- Launch SQL\*Plus command prompt from the directory where you created your PL/SQL file.
- Type @file\_name at the SQL\*Plus command prompt to execute your program.

If you are not using a file to execute PL/SQL scripts, then simply copy your PL/SQL code and then right click on the black window having SQL prompt and use **paste** option to paste complete code at the command prompt. Finally, just press enter to execute the code, if it is not already executed.

## PL/SQL - BASIC SYNTAX

PL/SQL is a block-structured language, meaning that PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts:

S.N.	Sections & Description
1	<b>Declarations</b> This section starts with the keyword <b>DECLARE</b> . It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	<b>Executable Commands</b> This section is enclosed between the keywords <b>BEGIN</b> and <b>END</b> and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which

	may be just a NULL command to indicate that nothing should be executed.
3	<b>Exception Handling</b> This section starts with the keyword <b>EXCEPTION</b> . This section is again optional and contains exception(s) that handle errors in the program.

Every PL/SQL statement end with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Here is the basic structure of a PL/SQL block:

```
DECLARE
    <declarations section>
BEGIN
    <executable command(s)>
EXCEPTION
    <exception handling>
END;
```

### The 'Hello World' Example:

```
DECLARE
    message varchar2(20) := 'Hello, World!';
BEGIN
    dbms_output.put_line(message);
END;
/
```

The **end;** line signals the end of the PL/SQL block. To run the code from SQL command line, you may need to type **/** at the beginning of the first blank line after the last line of the code. When the above code is executed at SQL prompt, it produces following result:

```
Hello World

PL/SQL procedure successfully completed.
```

## PL/SQL - DATA TYPES

PL/SQL variables, constants and parameters must have a valid data types which specifies a storage format, constraints, and valid range of values. This tutorial will take you through **SCALAR** and **LOB** data types available in PL/SQL and other two data types will be covered in other chapters.

Category	Description
Scalar	Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN.
Large Object (LOB)	Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
Composite	Data items that have internal components that can be accessed individually. For example, collections and records.
Reference	Pointers to other data items.

### PL/SQL Scalar Data Types and Subtypes

PL/SQL Scalar Data Types and Subtypes come under the following categories:

Date Type	Description
Numeric	Numeric values, on which arithmetic operations are performed.
Character	Alphanumeric values that represent single characters or strings of characters.
Boolean	Logical values, on which logical operations are performed.
Datetime	Dates and times.

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding PL/SQL code in another program, such as a Java program.

## PL/SQL Numeric Data Types and Subtypes

Following is the detail of PL/SQL pre-defined numeric data types and their sub-types:

Data Type	Description
PLS_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_FLOAT	Single-precision IEEE 754-format floating-point number
BINARY_DOUBLE	Double-precision IEEE 754-format floating-point number
NUMBER(prec, scale)	Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0.
DEC(prec, scale)	ANSI specific fixed-point type with maximum precision of 38 decimal digits.
DECIMAL(prec, scale)	IBM specific fixed-point type with maximum precision of 38 decimal digits.
NUMERIC(pre, scale)	Floating type with maximum precision of 38 decimal digits.
DOUBLE PRECISION	ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
FLOAT	ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
INT	ANSI specific integer type with maximum precision of 38 decimal digits
INTEGER	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
SMALLINT	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
REAL	Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Following is a valid declaration:

```
DECLARE
    num1 INTEGER;
    num2 REAL;
    num3 DOUBLE PRECISION;
BEGIN
    null;
END;
/
```

When the above code is compiled and executed, it produces following result:

```
PL/SQL procedure successfully completed
```

## PL/SQL Character Data Types and Subtypes

Following is the detail of PL/SQL pre-defined character data types and their sub-types:

Data Type	Description
CHAR	Fixed-length character string with maximum size of 32,767 bytes
VARCHAR2	Variable-length character string with maximum size of 32,767 bytes
RAW	Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
NCHAR	Fixed-length national character string with maximum size of 32,767 bytes
NVARCHAR2	Variable-length national character string with maximum size of 32,767 bytes
LONG	Variable-length character string with maximum size of 32,760 bytes
LONG RAW	Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
ROWID	Physical row identifier, the address of a row in an ordinary table
UROWID	Universal row identifier (physical, logical, or foreign row identifier)

## PL/SQL Boolean Data Types

The **BOOLEAN** data type stores logical values that are used in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL.

However, SQL has no data type equivalent to BOOLEAN. Therefore Boolean values cannot be used in:

- SQL statements
- Built-in SQL functions (such as TO\_CHAR)
- PL/SQL functions invoked from SQL statements

## PL/SQL Datetime and Interval Types

The **DATE** datatype to store fixed-length datetimes, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD.

The default date format is set by the Oracle initialization parameter `NLS_DATE_FORMAT`. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year, for example, 01-OCT-12.

Each DATE includes the century, year, month, day, hour, minute, and second. The following table shows the valid values for each field:

Field Name	Valid Datetime Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the dynamic performance view V\$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V\$TIMEZONE_NAMES	Not applicable

## PL/SQL Large Object (LOB) Data Types

Large object (LOB) data types refer large to data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types:

Data Type	Description	Size
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)
CLOB	Used to store large blocks of character data in the database.	8 to 128 TB

NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB
-------	---	-------------

## PL/SQL User-Defined Subtypes

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.

PL/SQL predefines several subtypes in package STANDARD. For example, PL/SQL predefines the subtypes CHARACTER and INTEGER as follows:

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

You can define and use your own subtypes. The following program illustrates defining and using a user-defined subtype:

```
DECLARE
    SUBTYPE name IS char(20);
    SUBTYPE message IS varchar2(100);
    salutation name;
    greetings message;
BEGIN
    salutation := 'Reader ';
    greetings := 'Welcome to the World of PL/SQL';
    dbms_output.put_line('Hello ' || salutation || greetings);
END;
/
```

When the above code is executed at SQL prompt, it produces following result:

```
Hello Reader Welcome to the World of PL/SQL

PL/SQL procedure successfully completed.
```

## NULLs in PL/SQL

PL/SQL NULL values represent missing or unknown data and they are not an integer, a character, or any other specific data type. Note that NULL is not the same as an empty data string or the null character value '\0'. A null can be assigned but it cannot be equated with anything, including itself.

## PL/SQL - VARIABLES

The name of a PL/SQL variable consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive. You cannot use a reserved PL/SQL keywords as a variable name.

### Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is:

```
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]
```

Where, *variable\_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type which we already have discussed in last chapter. Some valid variable declarations along with their definition are shown below:

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations, For example:

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

## Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following:

- The **DEFAULT** keyword
- The **assignment** operator

For example:

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the **NOT NULL** constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometime program would produce unexpected result. Try following example which makes use of various types of variables:

```
DECLARE
    a integer := 10;
    b integer := 20;
    c integer;
    f real;
BEGIN
    c := a + b;
    dbms_output.put_line('Value of c: ' || c);
    f := 70.0/3.0;
    dbms_output.put_line('Value of f: ' || f);
END;
```

When the above code is executed, it produces following result:

```
Value of c: 30
Value of f: 23.333333333333333333

PL/SQL procedure successfully completed.
```

## Variable Scope in PL/SQL

PL/SQL allows the nesting of Blocks i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer



Block, it is also accessible to all nested inner Blocks. There are two types of variable scope:

- **Local variables** - variables declared in an inner block and not accessible to outer blocks.
- **Global variables** - variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form:

```
DECLARE
  -- Global variables
  num1 number := 95;
  num2 number := 85;
BEGIN
  dbms_output.put_line('Outer Variable num1: ' || num1);
  dbms_output.put_line('Outer Variable num2: ' || num2);
  DECLARE
    -- Local variables
    num1 number := 195;
    num2 number := 185;
  BEGIN
    dbms_output.put_line('Inner Variable num1: ' || num1);
    dbms_output.put_line('Inner Variable num2: ' || num2);
  END;
END;
/
```

When the above code is executed, it produces following result:

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.
```

## PL/SQL - CONSTANTS AND LITERALS

---

A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the NOT NULL constraint.

### Declaring a Constant

A constant is declared using the CONSTANT keyword. It requires an initial value and does not allow that value to be changed, For example:

```
PI CONSTANT NUMBER := 3.141592654;
```

```
DECLARE
  -- constant declaration
  pi constant number := 3.141592654;
  -- other declarations
  radius number(5,2);
  dia number(5,2);
  circumference number(7, 2);
  area number (10, 2);
BEGIN
  -- processing
  radius := 9.5;
  dia := radius * 2;
  circumference := 2.0 * pi * radius;
  area := pi * radius * radius;
  -- output
  dbms_output.put_line('Radius: ' || radius);
  dbms_output.put_line('Diameter: ' || dia);
```

```

dbms_output.put_line('Circumference: ' || circumference);
dbms_output.put_line('Area: ' || area);
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:

```

Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53

PL/SQL procedure successfully completed.

```

## The PL/SQL Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, TRUE, 786, NULL, 'tutorialspoint' are all literals of type Boolean, number, or string. PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals:

- Numeric Literals
- Character Literals
- String Literals
- BOOLEAN Literals
- Date and Time Literals

The following table provides examples from all these categories of literal values.

Literal Type	Example:
Numeric Literals	050 78 -14 0 +32767 6.6667 0.0 -12.0 3.14159 +7800.00 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3
Character Literals	'A' '%' '9' ' ' 'z' '('
String Literals	'Hello, world!' 'Tutorials Point' '19-NOV-12'
BOOLEAN Literals	TRUE, FALSE, and NULL.
Date and Time Literals	DATE '1978-12-25'; TIMESTAMP '2012-10-29 12:01:01';

To embed single quotes within a string literal, place two single quotes next to each other as shown below:

```

DECLARE
    message varchar2(20) := 'That's tutorialspoint.com!';
BEGIN
    dbms_output.put_line(message);
END;
/

```

When the above code is executed at SQL prompt, it produces following result:

```
That's tutorialspoint.com!
PL/SQL procedure successfully completed.
```

## PL/SQL - OPERATORS

---

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. PL/SQL language is rich in built-in operators and provides following type of operators:

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

This tutorial will explain the arithmetic, relational, comparison and logical operators one by one. The String operators will be discussed under the chapter: **PL/SQL - Strings**.

### Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 5 then:

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiply both operands	A * B will give 50
/	Divide numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

### Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
=	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A = B) is not true.
!= <>	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true.

~ =		
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

## Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either TRUE, FALSE, OR NULL.

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b.	If x = 10 then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If x = 'm' then, x in ('a', 'b', 'c') returns boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If x = 'm', then 'x is null' returns Boolean false.

## Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produces Boolean results. Assume variable A holds true and variable B holds false then:

Operator	Description	Example
and	Called logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
or	Called logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.

not	Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.
-----	---	------------------------

## PL/SQL Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example  $x = 7 + 3 * 2$ ; Here x is assigned 13, not 20 because operator \* has higher precedence than + so it first get multiplied with  $3*2$  and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

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

## PL/SQL - CONDITIONS

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

### IF-THEN STATEMENT

It is the simplest form of **IF** control statement, frequently used in decision making and changing the control flow of the program execution.

The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is **TRUE**, the statements get executed and if the condition is **FALSE** or **NULL** then the **IF** statement does nothing.

#### Syntax:

Syntax for IF-THEN statement is:

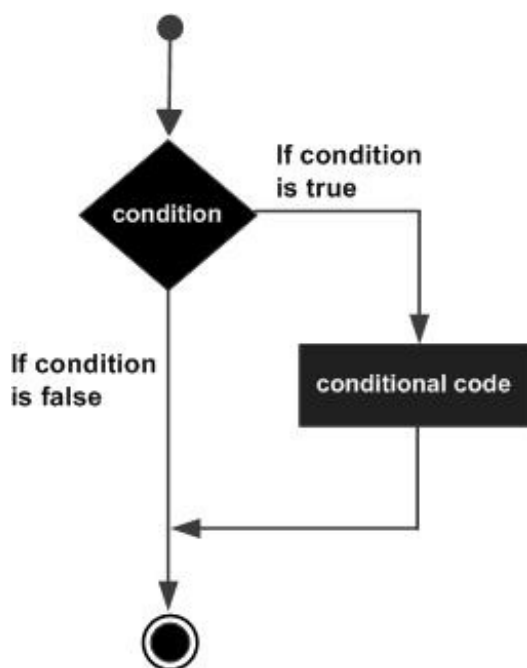
```
IF condition THEN
    S;
END IF;
```

Where *condition* is a Boolean or relational condition and *S* is a simple or compound statement. Example of an IF-THEN statement is:

```
IF (a <= 20) THEN
    c:= c+1;
END IF;
```

If the boolean expression *condition* evaluates to true then the block of code inside the if statement will be executed. If boolean expression evaluates to false then the first set of code after the end of the if statement (after the closing end if) will be executed.

### Flow Diagram:



## IF-THEN-ELSE STATEMENT

---

A sequence of **IF-THEN** statements can be followed by an optional sequence of **ELSE** statements, which executes when the condition is **FALSE**.

### Syntax:

Syntax for the IF-THEN-ELSE statement is:

```
IF condition THEN
    S1;
ELSE
    S2;
END IF;
```

Where, *S1* and *S2* are different sequence of statements. In the IF-THEN-ELSE statements, when the test *condition* is TRUE, the statement *S1* is executed and *S2* is skipped; when the test *condition* is FALSE, then *S1* is bypassed and statement *S2* is executed, For example,

---

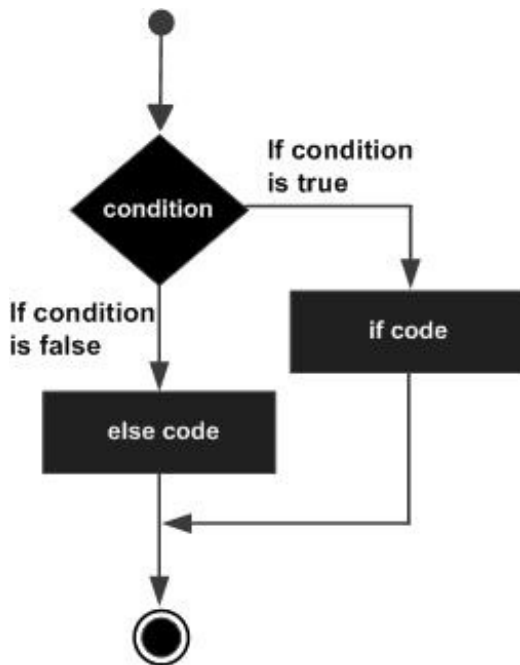
```

IF color = red THEN
    dbms_output.put_line('You have chosen a red car')
ELSE
    dbms_output.put_line('Please choose a color for your car');
END IF;

```

If the boolean expression *condition* evaluates to true then the if-then block of code will be executed otherwise the else block of code will be executed.

### Flow Diagram:



## IF-THEN-ELSIF STATEMENT

---

The **IF-THEN-ELSIF** statement allows you to choose between several alternatives. An **IF-THEN** statement can be followed by an optional **ELSIF...ELSE** statement. The **ELSIF** clause lets you add additional conditions.

When using **IF-THEN-ELSIF** statements there are few points to keep in mind.

- Its ELSIF not ELSEIF
- An IF-THEN statement can have zero or one ELSE's and it must come after any ELSEIF's.
- An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
- Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

### Syntax:

The syntax of an IF-THEN-ELSIF Statement in PL/SQL programming language is:

```

IF(boolean_expression 1) THEN
    S1; -- Executes when the boolean expression 1 is true
ELSIF( boolean_expression 2) THEN
    S2; -- Executes when the boolean expression 2 is true
ELSIF( boolean_expression 3) THEN
    S3; -- Executes when the boolean expression 3 is true
ELSE
    S4; -- executes when the none of the above condition is true
END IF;

```

# CASE STATEMENT

---

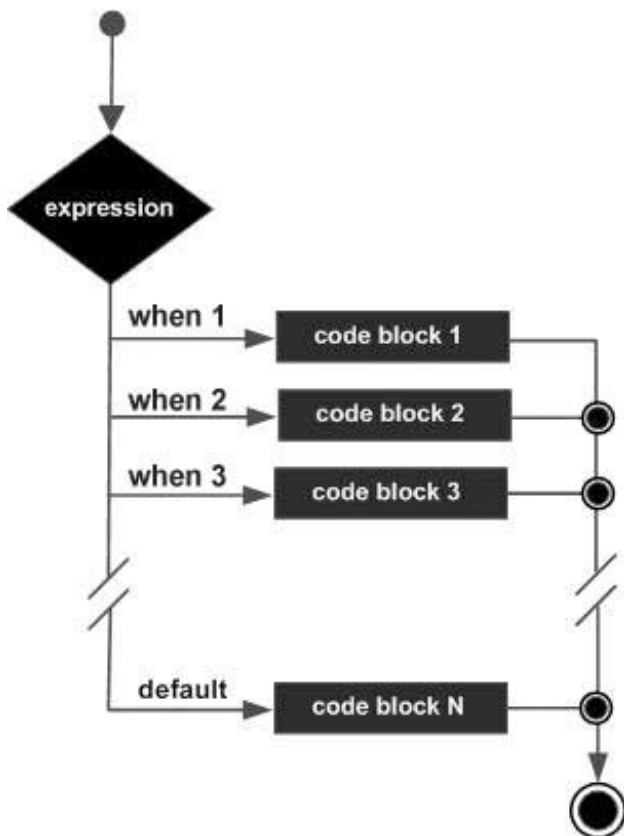
Like the **IF** statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the **CASE** statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.

## Syntax:

The syntax for case statement in PL/SQL is:

```
CASE selector
  WHEN 'value1' THEN S1;
  WHEN 'value2' THEN S2;
  WHEN 'value3' THEN S3;
  ...
  ELSE Sn; -- default case
END CASE;
```

## Flow Diagram:



# SEARCHED CASE STATEMENT

---

The searched **CASE** statement has no selector, and its **WHEN** clauses contain search conditions that give Boolean values.

## Syntax:

The syntax for searched case statement in PL/SQL is:

```
CASE
  WHEN selector = 'value1' THEN S1;
  WHEN selector = 'value2' THEN S2;
  WHEN selector = 'value3' THEN S3;
```

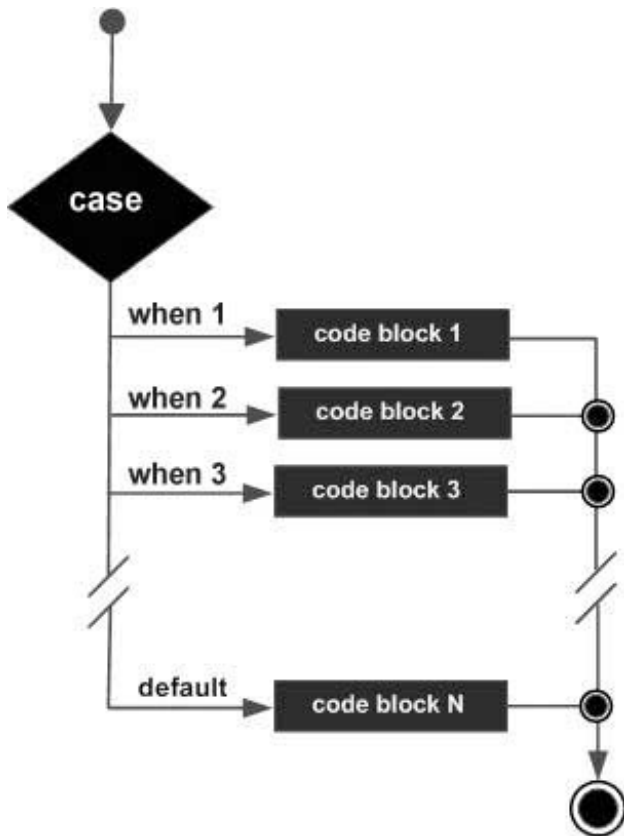


```

...
ELSE Sn;  -- default case
END CASE;

```

## Flow Diagram:



## NESTED IF-THEN-ELSE STATEMENTS

It is always legal in PL/SQL programming to nest **IF-ELSE** statements, which means you can use one **IF** or **ELSE IF** statement inside another **IF** or **ELSE IF** statement(s).

### Syntax:

```

IF( boolean_expression 1) THEN
    -- executes when the boolean expression 1 is true
    IF(boolean_expression 2) THEN
        -- executes when the boolean expression 2 is true
        sequence-of-statements;
    END IF;
ELSE
    -- executes when the boolean expression 1 is not true
    else-statements;
END IF;

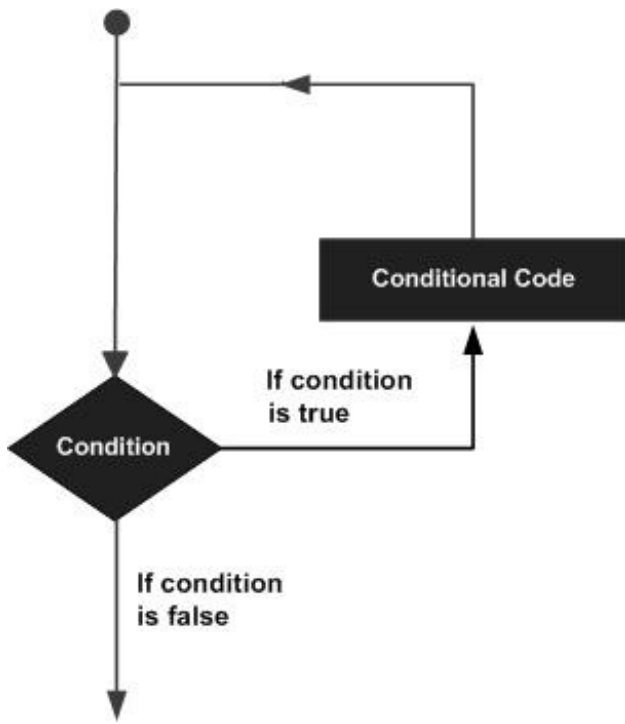
```

## PL/SQL - LOOPS

There may be a situation when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



## BASIC LOOP STATEMENT

---

Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP** statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

### Syntax:

The syntax of a basic loop in PL/SQL programming language is:

```
LOOP
    Sequence of statements;
END LOOP;
```

Here sequence of statement(s) may be a single statement or a block of statements. An EXIT statement or an EXIT WHEN statement is required to break the loop.

## WHILE LOOP STATEMENT

---

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

### Syntax:

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

## FOR LOOP STATEMENT

---

A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

### Syntax:

---

```
FOR counter IN initial_value .. final_value LOOP
    sequence_of_statements;
END LOOP;
```

Following are some special characteristics of PL/SQL for loop:

- The *initial\_value* and *final\_value* of the loop variable or *counter* can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception `VALUE_ERROR`.
- The *initial\_value* need not to be 1; however, the **loop counter increment (or decrement) must be 1**.
- PL/SQL allows determine the loop range dynamically at run time.

## NESTED LOOPS

---

PL/SQL allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

The syntax for a nested basic LOOP statement in PL/SQL is as follows:

```
LOOP
    Sequence of statements1
    LOOP
        Sequence of statements2
    END LOOP;
END LOOP;
```

The syntax for a nested FOR LOOP statement in PL/SQL is as follows:

```
FOR counter1 IN initial_value1 .. final_value1 LOOP
    sequence_of_statements1
    FOR counter2 IN initial_value2 .. final_value2 LOOP
        sequence_of_statements2
    END LOOP;
END LOOP;
```

The syntax for a nested WHILE LOOP statement in Pascal is as follows:

```
WHILE condition1 LOOP
    sequence_of_statements1
    WHILE condition2 LOOP
        sequence_of_statements2
    END LOOP;
END LOOP;
```

## EXIT STATEMENT

---

The **EXIT** statement in PL/SQL programming language has following two usages:

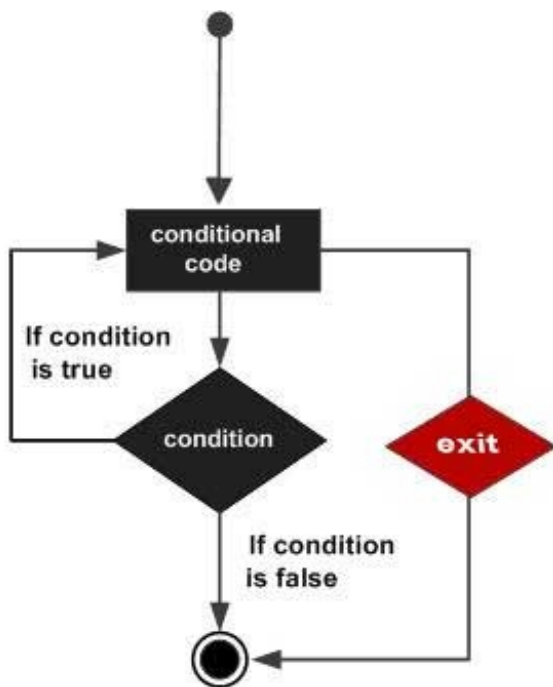
- When the **EXIT** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- If you are using nested loops (i.e. one loop inside another loop), the **EXIT** statement will stop the execution of the innermost loop and start executing the next line of code after the block.

### Syntax:

The syntax for a **EXIT** statement in PL/SQL is as follows:

```
EXIT;
```

## Flow Diagram:



## CONTINUE STATEMENT

---

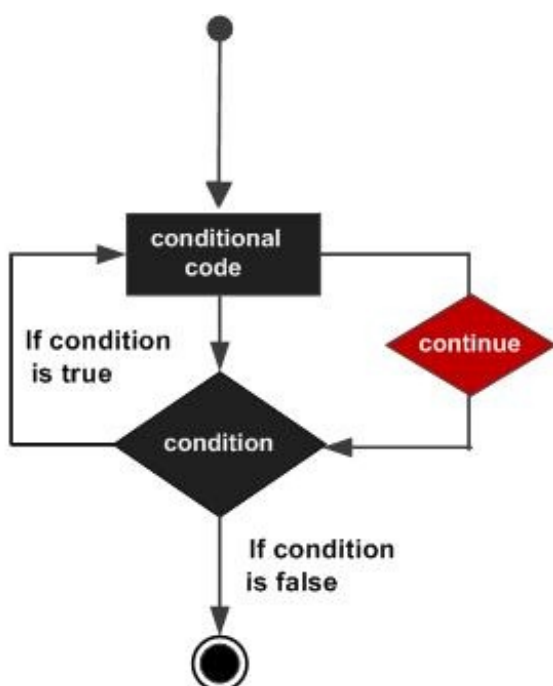
The **CONTINUE** statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

### Syntax:

The syntax for a CONTINUE statement is as follows:

```
CONTINUE;
```

## Flow Diagram:



# GOTO STATEMENT

---

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

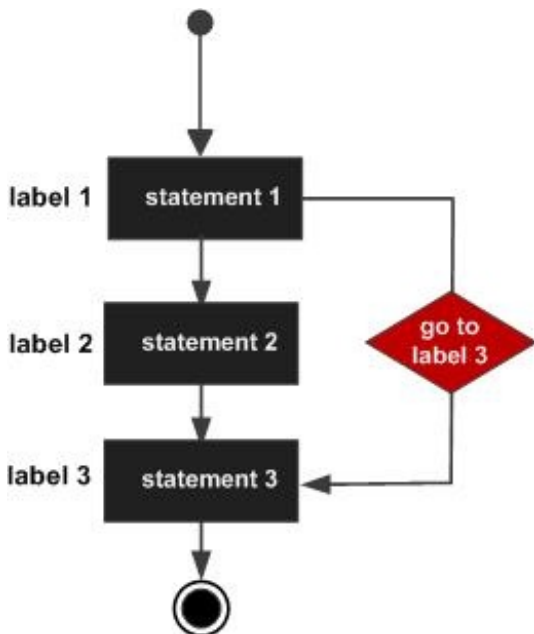
**NOTE:** Use of GOTO statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

## Syntax:

The syntax for a GOTO statement in PL/SQL is as follows:

```
GOTO label;  
..  
..  
<< label >>  
statement;
```

## Flow Diagram:



# PL/SQL - STRINGS

---

The string in PL/SQL is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all. PL/SQL offers three kinds of strings:

- **Fixed-length strings:** In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.
- **Variable-length strings:** In such strings, a maximum length upto 32,767, for the string is specified and no padding takes place.
- **Character large objects (CLOBs):** These are variable-length strings that can be up to 128 terabytes.

PL/SQL strings could be either variables or literals. A string literal is enclosed within quotation marks. For example,

```
'This is a string literal.' Or 'hello world'
```

To include a single quote inside a string literal, you need to type two single quotes next to one another, like:

```
'this isn't what it looks like'
```

## Declaring String Variables

Oracle database provides numerous string datatypes, like, CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an 'N' are 'national character set' datatypes, that store Unicode character data.

If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type. The following example illustrates declaring and using some string variables:

```
DECLARE
    name varchar2(20);
    company varchar2(30);
    introduction clob;
    choice char(1);
BEGIN
    name := 'John Smith';
    company := 'Infotech';
    introduction := ' Hello! I'm John Smith from Infotech.';
    choice := 'y';
    IF choice = 'y' THEN
        dbms_output.put_line(name);
        dbms_output.put_line(company);
        dbms_output.put_line(introduction);
    END IF;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
John Smith
Infotech Corporation
Hello! I'm John Smith from Infotech.

PL/SQL procedure successfully completed
```

To declare a fixed-length string, use the CHAR datatype. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. So following two declarations below are identical:

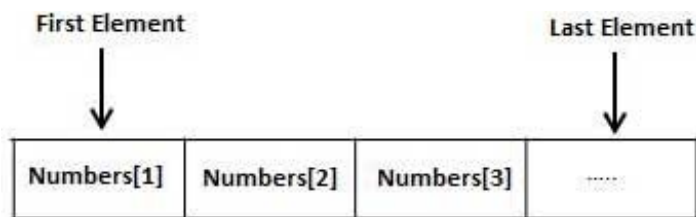
```
red_flag CHAR(1) := 'Y';
red_flag CHAR    := 'Y';
```

## PL/SQL - ARRAYS

---

PL/SQL programming language provides a data structure called the VARRAY, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter 'PL/SQL Collections'.

Each element in a varray has an index associated with it. It also has a maximum size that can be changed dynamically.

## Creating a Varray Type

A varray type is created with the CREATE TYPE statement. You must specify the maximum size and the type of elements stored in the varray.

The basic syntax for creating a VARRAY type at the schema level is:

```
CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>
```

Where,

- *varray\_type\_name* is a valid attribute name,
- *n* is the number of elements (maximum) in the varray,
- *element\_type* is the data type of the elements of the array.

Maximum size of a varray can be changed using the ALTER TYPE statement.

For example,

```
CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);
/

Type created.
```

The basic syntax for creating a VARRAY type within a PL/SQL block is:

```
TYPE varray_type_name IS VARRAY(n) of <element_type>
```

For example:

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
Type grades IS VARRAY(5) OF INTEGER;
```

## PL/SQL - PROCEDURES

---

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program, which is called the calling program.

A subprogram can be created:

- At schema level

- Inside a package
- Inside a PL/SQL block

A schema level subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms:

- **Functions:** these subprograms return a single value, mainly used to compute and return a value.
- **Procedures:** these subprograms do not return a value directly, mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure** and we will cover **PL/SQL function** in next chapter.

## Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may have a parameter list. Like anonymous PL/SQL blocks and, the named blocks a subprograms will also have following three parts:

S.N.	Parts & Description
1	<b>Declarative Part</b> It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	<b>Executable Part</b> This is a mandatory part and contains statements that perform the designated action.
3	<b>Exception-handling</b> This is again an optional part. It contains the code that handles run-time errors.

## Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.



- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

## PL/SQL - FUNCTIONS

---

A PL/SQL function is same as a procedure except that it returns a value.

### Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
    < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- *RETURN* clause specifies that data type you are going to return from the function.
- *function-body* contains the executable part.
- *function-body* must contain a **RETURN statement**.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

## PL/SQL - CURSORS

---

Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursors
- Explicit cursors

## Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that need to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK\_ROWCOUNT and %BULK\_EXCEPTIONS, designed for use with the FORALL statement.

## Explicit Cursors

Explicit cursors are programmer defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is :

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor involves four steps:

- Declaring the cursor for initializing in the memory
- Opening the cursor for allocating memory
- Fetching the cursor for retrieving data
- Closing the cursor to release allocated memory

## Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

```
CURSOR c_customers IS  
  SELECT id, name, address FROM customers;
```

## Opening the Cursor

Opening the cursor allocates memory for the cursor, and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open above defined cursor as follows:

```
OPEN c_customers;
```

## Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example we will fetch rows from the above open cursor as follows:

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

---

## Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close above opened cursor as follows:

```
CLOSE c_customers;
```

## Example:

Following is a complete example to illustrate the concepts of explicit cursors:

```
DECLARE
    c_id customers.id%type;
    c_name customers.name%type;
    c_addr customers.address%type;
    CURSOR c_customers is
        SELECT id, name, address FROM customers;
BEGIN
    OPEN c_customers;
    LOOP
        FETCH c_customers into c_id, c_name, c_addr;
        dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
        EXIT WHEN c_customers%notfound;
    END LOOP;
    CLOSE c_customers;
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

---

## PL/SQL - RECORDS

A PL/SQL **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

PL/SQL can handle following types of records:

- Table-based
- Cursor-based records
- User-defined records

### Table-Based Records

The %ROWTYPE attribute enables a programmer to create **table-based** and **cursor-based** records.

The following example would illustrate the concept of **table-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
    customer_rec customers%rowtype;
```

```

BEGIN
    SELECT * into customer_rec
    FROM customers
    WHERE id = 5;

    dbms_output.put_line('Customer ID: ' || customer_rec.id);
    dbms_output.put_line('Customer Name: ' || customer_rec.name);
    dbms_output.put_line('Customer Address: ' || customer_rec.address);
    dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:

```

Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000

PL/SQL procedure successfully completed.

```

## Cursor-Based Records

The following example would illustrate the concept of **cursor-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```

DECLARE
    CURSOR customer_cur is
        SELECT id, name, address
        FROM customers;
    customer_rec customer_cur%rowtype;
BEGIN
    OPEN customer_cur;
    LOOP
        FETCH customer_cur into customer_rec;
        EXIT WHEN customer_cur%notfound;
        DBMS_OUTPUT.put_line(customer_rec.id || ' ' || customer_rec.name);
    END LOOP;
END;
/

```

When the above code is executed at SQL prompt, it produces the following result:

```

1 Ramesh
2 Khilan
3 kaushik
4 Chaitali
5 Hardik
6 Komal

PL/SQL procedure successfully completed.

```

## User-Defined Records

PL/SQL provides a user-defined record type that allows you to define different record structures. Records consist of different fields. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject

- Book ID

## Defining a Record

The record type is defined as:

```
TYPE
type_name IS RECORD
( field_name1  datatype1  [NOT NULL]  [:= DEFAULT EXPRESSION],
  field_name2  datatype2  [NOT NULL]  [:= DEFAULT EXPRESSION],
  ...
  field_nameN  datatypeN  [NOT NULL]  [:= DEFAULT EXPRESSION]);
record-name  type_name;
```

Here is the way you would declare the Book record:

```
DECLARE
TYPE books IS RECORD
(title  varchar(50),
 author  varchar(50),
 subject varchar(100),
 book_id  number);
book1 books;
book2 books;
```

## Accessing Fields

To access any field of a record, we use the dot (.) operator. The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is the example to explain usage of record:

```
DECLARE
    type books is record
        (title varchar(50),
         author varchar(50),
         subject varchar(100),
         book_id number);
    book1 books;
    book2 books;
BEGIN
    -- Book 1 specification
    book1.title := 'C Programming';
    book1.author := 'Nuha Ali ';
    book1.subject := 'C Programming Tutorial';
    book1.book_id := 6495407;

    -- Book 2 specification
    book2.title := 'Telecom Billing';
    book2.author := 'Zara Ali';
    book2.subject := 'Telecom Billing Tutorial';
    book2.book_id := 6495700;

    -- Print book 1 record
    dbms_output.put_line('Book 1 title : ' || book1.title);
    dbms_output.put_line('Book 1 author : ' || book1.author);
    dbms_output.put_line('Book 1 subject : ' || book1.subject);
    dbms_output.put_line('Book 1 book_id : ' || book1.book_id);

    -- Print book 2 record
    dbms_output.put_line('Book 2 title : ' || book2.title);
    dbms_output.put_line('Book 2 author : ' || book2.author);
    dbms_output.put_line('Book 2 subject : ' || book2.subject);
    dbms_output.put_line('Book 2 book_id : ' || book2.book_id);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

PL/SQL procedure successfully completed.

## PL/SQL - EXCEPTIONS

---

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

- System-defined exceptions
- User-defined exceptions

### Syntax for Exception Handling

The General Syntax for exception handling is as follows. Here you can list down as many as exceptions you want to handle. The default exception will be handled using *WHEN others THEN*:

```
DECLARE
    <declarations section>
BEGIN
    <executable command(s)>
EXCEPTION
    <exception handling goes here >
    WHEN exception1 THEN
        exception1-handling-statements
    WHEN exception2 THEN
        exception2-handling-statements
    WHEN exception3 THEN
        exception3-handling-statements
    .....
    WHEN others THEN
        exception3-handling-statements
END;
```

### Example

Let us write some simple code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
    c_id customers.id%type := 8;
    c_name customers.name%type;
    c_addr customers.address%type;
BEGIN
    SELECT name, address INTO c_name, c_addr
    FROM customers
    WHERE id = c_id;

    DBMS_OUTPUT.PUT_LINE ('Name: ' || c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
    WHEN no_data_found THEN
        dbms_output.put_line('No such customer!');
    WHEN others THEN
        dbms_output.put_line('Error!');
```

```
END;  
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
No such customer!  
  
PL/SQL procedure successfully completed.
```

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO\_DATA\_FOUND** which is captured in **EXCEPTION** block.

## PL/SQL - TRIGGERS

---

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are in fact, written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

### Creating Triggers

The Syntax for creating a trigger is:

```
CREATE [OR REPLACE ] TRIGGER trigger_name  
{BEFORE | AFTER | INSTEAD OF }  
{INSERT [OR] | UPDATE [OR] | DELETE}  
[OF col_name]  
ON table_name  
[REFERENCING OLD AS o NEW AS n]  
[FOR EACH ROW]  
WHEN (condition)  
DECLARE  
    Declaration-statements  
BEGIN  
    Executable-statements  
EXCEPTION  
    Exception-handling-statements  
END;
```

Where,

- **CREATE [OR REPLACE] TRIGGER trigger\_name** : Creates or replace an existing trigger with the *trigger\_name*.
- **{ BEFORE | AFTER | INSTEAD OF }** : This specifies when the trigger would be executed. The **INSTEAD OF** clause is used for creating trigger on a view.
- **{ INSERT [OR] | UPDATE [OR] | DELETE }**: This specifies the DML operation.
- **[OF col\_name]**: This specifies the column name that would be updated.
- **[ON table\_name]**: This specifies the name of the table associated with the trigger.
- **[REFERENCING OLD AS o NEW AS n]**: This allows you to refer new and old values for various DML

statements, like INSERT, UPDATE, and DELETE.

- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

## Example:

To start with, we will be using the CUSTOMERS table:

```
Select * from customers;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00

The following program creates a **row level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff := :NEW.salary - :OLD.salary;
    dbms_output.put_line('Old salary: ' || :OLD.salary);
    dbms_output.put_line('New salary: ' || :NEW.salary);
    dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Trigger created.
```

Here following two points are important and should be noted carefully:

- OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.



# PL/SQL - PACKAGES

---

PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.

A package will have two mandatory parts:

- Package specification
- Package body or definition

## Package Specification

The specification is the interface to the package. It just **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
    PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

When the above code is executed at SQL prompt, it produces following result:

```
Package created.
```

## Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.

The **CREATE PACKAGE BODY** Statement is used for creating the package body. The following code snippet shows the package body declaration for the *cust\_sal* package created above. I assumed that we already have **CUSTOMERS** table created in our database as mentioned in [PL/SQL - Variables](#) chapter.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
    PROCEDURE find_sal(c_id customers.id%TYPE) IS
        c_sal customers.salary%TYPE;
    BEGIN
        SELECT salary INTO c_sal
        FROM customers
        WHERE id = c_id;
        dbms_output.put_line('Salary: ' || c_sal);
    END find_sal;
END cust_sal;
/
```

When the above code is executed at SQL prompt, it produces following result:

```
Package body created.
```

## Using the Package Elements

The package elements ( variables, procedures or functions) are accessed with the following syntax:

```
package_name.element_name;
```

Consider, we already have created above package in our database schema, the following program uses the *find\_sal* method of the *cust\_sal* package:

```
DECLARE
    code customers.id%type := &cc_id;
BEGIN
    cust_sal.find_sal(code);
END;
/
```

When the above code is executed at SQL prompt, it prompt to enter customer ID and when you enter an ID, it displays corresponding salary as follows:

```
Enter value for cc_id: 1
Salary: 3000

PL/SQL procedure successfully completed.
```

## PL/SQL - COLLECTIONS

---

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types:

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

### Index-By Table

An **index-by** table (also called an associative array) is a set of **key-value** pairs. Each key is unique, and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here we are creating an index-by table named **table\_name** whose keys will be of *subscript\_type* and associated values will be of *element\_type*

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;

table_name type_name;
```

### Example:

Following example how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
    TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
    salary_list salary;
    name    VARCHAR2(20);
BEGIN
    -- adding elements to the table
    salary_list('Rajnish') := 62000;
```

```

salary_list('Minakshi') := 75000;
salary_list('Martin') := 100000;
salary_list('James') := 78000;

-- printing the table
name := salary_list.FIRST;
WHILE name IS NOT null LOOP
    dbms_output.put_line
        ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
    name := salary_list.NEXT(name);
END LOOP;
END;
/

```

When the above code is executed at SQL prompt, it produces following result:

```

Salary of Rajnish is 62000
Salary of Minakshi is 75000
Salary of Martin is 100000
Salary of James is 78000

PL/SQL procedure successfully completed.

```

## Nested Tables

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects:

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
- An array is always dense i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

An **nested table** is created using the following syntax:

```

TYPE type_name IS TABLE OF element_type [NOT NULL];

table_name type_name;

```

This declaration is similar to declaration of an **index-by** table, but there is no INDEX BY clause.

A nested table can be stored in a database column and so it could be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

## Example:

The following examples illustrate the use of nested table:

```

DECLARE
    TYPE names_table IS TABLE OF VARCHAR2(10);
    TYPE grades IS TABLE OF INTEGER;

    names names_table;
    marks grades;
    total integer;
BEGIN
    names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
    marks:= grades(98, 97, 78, 87, 92);
    total := names.count;
    dbms_output.put_line('Total '|| total || ' Students');
    FOR i IN 1 .. total LOOP
        dbms_output.put_line('Student:'||names(i)||', Marks:' || marks(i));
    end loop;

```

```
END;  
/
```

When the above code is executed at SQL prompt, it produces following result:

```
Total 5 Students  
Student:Kavita, Marks:98  
Student:Pritam, Marks:97  
Student:Ayan, Marks:78  
Student:Rishav, Marks:87  
Student:Aziz, Marks:92  
  
PL/SQL procedure successfully completed.
```

## PL/SQL - TRANSACTIONS

---

A database **transaction** is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed i.e. made permanent to the database or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

### Starting an Ending a Transaction

A transaction has a **beginning** and an **end**. A transaction starts when one of the following events take place:

- The first SQL statement is performed after connecting into the database.
- At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place:

- A COMMIT or a ROLLBACK statement is issued.
- A DDL statement, like CREATE TABLE statement, is issued; because in that case a COMMIT is automatically performed.
- A DCL statement, such as a GRANT statement, is issued; because in that case a COMMIT is automatically performed.
- User disconnects from the database.
- User exits from SQL\*PLUS by issuing the EXIT command, a COMMIT is automatically performed.
- SQL\*Plus terminates abnormally, a ROLLBACK is automatically performed.
- A DML statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

### Committing a Transaction

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is:

```
COMMIT;
```

### Rolling Back Transactions

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is:

```
ROLLBACK [TO SAVEPOINT < savepoint_name>];
```

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using **savepoint** then simply use the following statement to rollback all the changes:

```
ROLLBACK;
```

## Savepoints

Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the SAVEPOINT command.

The general syntax for the SAVEPOINT command is:

```
SAVEPOINT < savepoint_name >;
```

## Automatic Transaction Control

To execute a COMMIT automatically whenever an INSERT, UPDATE or DELETE command is executed, you can set the AUTOCOMMIT environment variable as:

```
SET AUTOCOMMIT ON;
```

You can turn-off auto commit mode using the following command:

```
SET AUTOCOMMIT OFF;
```

## PL/SQL - DATE & TIME

---

PL/SQL provides two classes of date and time related data types:

- Datetime data types
- Interval data types

The Datetime data types are:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE

The Interval data types are:

- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

## The Datetime Data Types

Following are the Datetime data types:

- **DATE** - it stores date and time information in both character and number datatypes. It is made of information on century, year, month, date, hour, minute, and second. It is specified as:
- **TIMESTAMP** - it is an extension of the DATE datatype. It stores the year, month, and day of the DATE datatype, along with hour, minute, and second values. It is useful for storing precise time values.
- **TIMESTAMP WITH TIME ZONE** - it is a variant of TIMESTAMP that includes a time zone region name or a time zone offset in its value. The time zone offset is the difference (in hours and minutes) between local time and UTC. This datatype is useful for collecting and evaluating date information across geographic regions.
- **TIMESTAMP WITH LOCAL TIME ZONE** - it is another variant of TIMESTAMP that includes a time zone offset in its value.

### Examples:

```
SELECT SYSDATE FROM DUAL;
```

Output:

```
08/31/2012 5:25:34 PM
```

```
SELECT TO_CHAR(CURRENT_DATE, 'DD-MM-YYYY HH:MI:SS') FROM DUAL;
```

Output:

```
31-08-2012 05:26:14
```

```
SELECT ADD_MONTHS(SYSDATE, 5) FROM DUAL;
```

Output:

```
01/31/2013 5:26:31 PM
```

```
SELECT LOCALTIMESTAMP FROM DUAL;
```

Output:

```
8/31/2012 5:26:55.347000 PM
```

## The Interval Data Types

Following are the Interval data types:

- **INTERVAL YEAR TO MONTH** - it stores a period of time using the YEAR and MONTH datetime fields.
- **INTERVAL DAY TO SECOND** - it stores a period of time in terms of days, hours, minutes, and seconds.

## PL/SQL - DBMS OUTPUT

---

The **DBMS\_OUTPUT** is a built in package that enables you to display output, display debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers.

## Example:

```
DECLARE
    lines dbms_output.chararr;
    num_lines number;
BEGIN
    -- enable the buffer with default size 20000
    dbms_output.enable;

    dbms_output.put_line('Hello Reader!');
    dbms_output.put_line('Hope you have enjoyed the tutorials!');
    dbms_output.put_line('Have a great time exploring pl/sql!');

    num_lines := 3;

    dbms_output.get_lines(lines, num_lines);

    FOR i IN 1..num_lines LOOP
        dbms_output.put_line(lines(i));
    END LOOP;
END;
/
```

When the above code is executed at SQL prompt, it produces following result:

```
Hello Reader!
Hope you have enjoyed the tutorials!
Have a great time exploring pl/sql!

PL/SQL procedure successfully completed.
```

## PL/SQL - OBJECT ORIENTED

---

PL/SQL allows defining an object type, which helps in designing object-oriented database in Oracle. An object type allows you to create composite types. Using objects allow you implementing real world objects with specific structure of data and methods for operating it. Objects have attributes and methods. Attributes are properties of an object and are used for storing an object's state; and methods are used for modeling its behaviors.

Objects are created using the CREATE [OR REPLACE] TYPE statement. Below is an example to create a simple **address** object consisting of few attributes:

```
CREATE OR REPLACE TYPE address AS OBJECT
(house_no varchar2(10),
 street varchar2(30),
 city varchar2(20),
 state varchar2(10),
 pincode varchar2(10)
);
/
```

When the above code is executed at SQL prompt, it produces following result:

```
Type created.
```

Let's create one more object **customer** where we will wrap **attributs** and **methods** together to have object oriented feeling:

```
CREATE OR REPLACE TYPE customer AS OBJECT
(code number(5),
 name varchar2(30),
 contact_no varchar2(12),
 addr address,
 member procedure display
```

```
);  
/
```

When the above code is executed at SQL prompt, it produces following result:

```
Type created.
```

## Instantiating an Object

Defining an object type provides a blueprint for the object. To use this object, you need to create instances of this object. You can access the attributes and methods of the object using the instance name and **the access operator (.)** as follows:

```
DECLARE  
    residence address;  
BEGIN  
    residence := address('103A', 'M.G.Road', 'Jaipur', 'Rajasthan','201301');  
    dbms_output.put_line('House No: ' || residence.house_no);  
    dbms_output.put_line('Street: ' || residence.street);  
    dbms_output.put_line('City: ' || residence.city);  
    dbms_output.put_line('State: ' || residence.state);  
    dbms_output.put_line('Pincode: ' || residence.pincode);  
END;  
/
```

When the above code is executed at SQL prompt, it produces following result:

```
House No: 103A  
Street: M.G.Road  
City: Jaipur  
State: Rajasthan  
Pincode: 201301  
  
PL/SQL procedure successfully completed.
```

## Inheritance for PL/SQL Objects:

PL/SQL allows creating object from existing base objects. To implement inheritance, the base objects should be declared as NOT FINAL. The default is FINAL.

The following programs illustrates inheritance in PL/SQL Objects. Let us create another object named TableTop which is inheriting from the Rectangle object. Creating the base *rectangle* object:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT  
(length number,  
    width number,  
    member function enlarge( inc number) return rectangle,  
    NOT FINAL member procedure display) NOT FINAL  
/
```

When the above code is executed at SQL prompt, it produces following result:

```
Type created.
```

Creating the base type body:

```
CREATE OR REPLACE TYPE BODY rectangle AS  
    MEMBER FUNCTION enlarge(inc number) return rectangle IS  
    BEGIN  
        return rectangle(self.length + inc, self.width + inc);  
    END enlarge;
```



```

MEMBER PROCEDURE display IS
BEGIN
    dbms_output.put_line('Length: ' || length);
    dbms_output.put_line('Width: ' || width);
END display;
END;
/

```

When the above code is executed at SQL prompt, it produces following result:

```
Type body created.
```

Creating the child object *tabletop*:

```

CREATE OR REPLACE TYPE tabletop UNDER rectangle
(
    material varchar2(20);
    OVERRIDING member procedure display
)
/

```

When the above code is executed at SQL prompt, it produces following result:

```
Type created.
```

Creating the type body for the child object *tabletop*:

```

CREATE OR REPLACE TYPE BODY tabletop AS
OVERRIDING MEMBER PROCEDURE display IS
BEGIN
    dbms_output.put_line('Length: ' || length);
    dbms_output.put_line('Width: ' || width);
    dbms_output.put_line('Material: ' || material);
END display;
/

```

When the above code is executed at SQL prompt, it produces following result:

```
Type body created.
```

Using the *tabletop* object and its member functions:

```

DECLARE
    t1 tabletop;
    t2 tabletop;
BEGIN
    t1:= tabletop(20, 10, 'Wood');
    t2 := tabletop(50, 30, 'Steel');
    t1.display;
    t2.display;
END;
/

```

When the above code is executed at SQL prompt, it produces following result:

```

Length: 20
Width: 10
Material: Wood
Length: 50
Width: 30
Material: Steel

PL/SQL procedure successfully completed.

```

## Abstract Objects in PL/SQL

The NOT INSTANTIABLE clause allows you to declare an abstract object. You cannot use an abstract object as it is; you will have to create a subtype or child type of such objects to use its functionalities.

For example,

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
 width number,
 NOT INSTANTIABLE NOT FINAL MEMBER PROCEDURE display)
NOT INSTANTIABLE NOT FINAL
/
```

When the above code is executed at SQL prompt, it produces following result:

```
Type created.
```