**PL/SQL**

# PL/SQL Contents

* Basic Syntax, Data Types, Variables, Constants
* Operators, Conditions, Loops
* Strings, Arrays
* Procedures, Functions
* Cursors, Records, Exceptions, Triggers
* Packages
* Collections
* Transactions
* Date & Time
* DBMS Output
* Object Oriented

**What is PL/SQL?**

* **PL/SQL** stands for **Procedural Language** extension of SQL.
* PL/SQL is a combination of SQL along with the procedural features of programming languages.
* It was developed by Oracle Corporation in the early 90’s to enhance the capabilities of SQL.
* Oracle uses a **PL/SQL** engine to processes the PL/SQL statements.
* A PL/SQL language code can be stored in the client system (client-side) or in the database (server-side).

## PL/SQL Block

**A Simple PL/SQL Block:**

* Each PL/SQL program consists of SQL and PL/SQL statements which from a PL/SQL block.

|  |
| --- |
| DECLARE  Variable declaration  BEGIN  Program Execution  EXCEPTION  Exception handling  END; |

* Every PL/SQL statement ends with a semicolon **(;)**
* **PL/SQL Block consists of three sections:** 
  + The Declaration section (optional).
  + The Execution section (mandatory).
  + The Exception Handling (or Error) section (optional).

|  |
| --- |
| DECLARE  message varchar2(20):= 'Hello, World!';  BEGIN dbms\_output.put\_line(message); END; |

## Advantages of PL/SQL

**These are the Advantages of PL/SQL :**

* ***Block Structures*:** PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.
* ***Procedural Language Capability*:** PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).
* ***Better Performance*:** PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.
* ***Error Handling*:** PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

## PL/SQL Delimiters and Comments

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Delimiter | Description | | **+, -, \*, /** | Addition, subtraction/negation, multiplication, division | | **%** | Attribute indicator | | **'** | Character string delimiter | | **.** | Component selector | | **(,)** | Expression or list delimiter | | **:** | Host variable indicator | | **,** | Item separator | | **"** | Quoted identifier delimiter | | **=** | Relational operator | | **@** | Remote access indicator | | **;** | Statement terminator | | |  | | --- | | Delimiter Description  **:=** Assignment operator  **=>** Association operator  **||** Concatenation operator  **\*\*** Exponentiation operator  **<<, >>** Label delimiter (begin and end)  Multi-line comment delimiter (begin  **/\*, \*/**  and end)  **--** Single-line comment indicator  **..** Range operator  **<, >, <=,**  Relational operators  **>=**  **<>, '=,**  Different versions of NOT EQUAL **~=, ^=** | |

-- (Single Line), /\* … \*/ (Multi line) are the Comments **identifiers are not case-sensitive**

## PL/SQL Datatypes

PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and valid range of values.

|  |  |
| --- | --- |
| 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 and LOB Datatypes

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 LOB Data Types and Subtypes come under the following categories:

|  |  |
| --- | --- |
| Date Type |  |
| BFILE | Used to store large binary objects in operating system files outside the database. |
| BLOB | Used to store large binary objects in the database. |
| CLOB | Used to store large blocks of character data in the database. |
| NCLOB | Used to store large blocks of NCHAR data in the database. |

## PL/SQL User Defined Datatypes

* 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 Eg : PL/SQL predefines the subtypes CHARACTER and INTEGER as follows:

SUBTYPE CHARACTER IS CHAR;

SUBTYPE INTEGER IS NUMBER(38,0);

Now can define User-Defined subtype in the following program

|  |
| --- |
| 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; |

## PL/SQL Variables and Declarations

These are placeholders that store the values that can change through the PL/SQL Block.

**General Syntax to declare a variable(local/global):**

variable\_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial value]

* *variable\_name* is the name of the variable.
* *datatype* is a valid PL/SQL datatype.
* **NOT NULL** is an optional specification on the variable.
* *value* or **DEFAULT** *value* is also an optional specification, where you can initialize a variable.
* Each variable declaration is a separate statement and must be terminated by **‘;’**

|  |
| --- |
| DECLARE  salary number (6);  dept varchar2(10) NOT NULL := “HR Dept”; var\_emp\_id number(6) = 1116; |

**PL/SQL Variables and Declarations**

### 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;

## Assigning SQL Query Results to PL/SQL Variables

**CREATE TABLE CUSTOMERS**( ***ID INT NOT NULL, NAME VARCHAR (20) NOT NULL, AGE INT NOT***

***NULL, ADDRESS CHAR (25), SALARY DECIMAL (18, 2), PRIMARY KEY (ID***) );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 ); I

NSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (6, 'Komal', 22, 'MP', 4500.00 );

|  |
| --- |
| DECLARE  c\_id customers.id%type := 1; c\_name customers.name%type; c\_addr customers.address%type;  c\_sal customers.salary%type;  BEGIN  SELECT name, address, salary INTO c\_name, c\_addr, c\_sal FROM customers WHERE id = c\_id; dbms\_output.put\_line ('Customer ' ||c\_name || ' from ' || c\_addr || ' earns ' || c\_sal); END; |

## PL/SQL Constants and Literals

**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 Eg:

|  |
| --- |
| DECLARE  pi constant number := 3.141592654; radius number(5,2); dia number(5,2); circumference number(7, 2); area number (10, 2);  BEGIN  radius := 9.5; dia := radius \* 2; circumference := 2.0 \* pi \* radius; area := pi \* radius \* radius; 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; |

## PL/SQL Constants and Literals

**The PL/SQL Literals :** A literal is an explicit Numeric, Character, String, Date and Time or Boolean value not represented by an identifier.

For Eg:, TRUE, 786, NULL, 'tutorialspoint' are all literals of type Boolean, Number, or String. PL/SQL, Literals are Case-Sensitive.

|  |  |  |  |
| --- | --- | --- | --- |
| Literal Type | | Example: | |
| Numeric Literals  Character Literals  String Literals  BOOLEAN Literals | |  | |
| 'A' '%' '9' ' ' 'z' '(' | |
| TRUE, FALSE, and NULL. | |
| Date and Time Literals | | DATE '1978-12-25'; TIMESTAMP '2012-10-29 12:01:01'; | |
| DECLARE  message varchar2(30):= 'That''s PL/SQL!';  BEGIN dbms\_output.put\_line(message); END; | |

## PL/SQL Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation.

PL/SQL language is rich in built-in operators and provides the following types of operators:

|  |  |  |
| --- | --- | --- |
| • | Arithmetic operators | : +, - , \* , / , \*\* |
| • | Relational operators | : >, <, >=, <=, !=, ==, <>, ~= |
| • | Comparison operators | : LIKE, BETWEEN, IN, ISNULL |
| • | Logical operators | : AND, OR, NOT |
| • | String Operator | : || |

## PL/SQL Operators Precedence

|  |  |
| --- | --- |
| Operator | Operation |
| \*\* | exponentiation |
| +, - | identity, negation |
| \*, / | multiplication, division |
| +, -, ||  comparison  NOT | addition, subtraction, concatenation |
| logical negation |
| AND | conjunction |
| OR | inclusion |

## PL/SQL Conditions

|  |  |
| --- | --- |
| Statement Description | |
| [IF -](http://www.tutorialspoint.com/plsql/plsql_if_then.htm) [THEN statement](http://www.tutorialspoint.com/plsql/plsql_if_then.htm) | 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. |
| [IF-THEN-ELSE](http://www.tutorialspoint.com/plsql/plsql_if_then_else.htm) [statement](http://www.tutorialspoint.com/plsql/plsql_if_then_else.htm) | **IF statement** adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL , then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed. |
| [IF-THENELSIF statement](http://www.tutorialspoint.com/plsql/plsql_if_then_elsif.htm) | It allows you to choose between several alternatives. |
| [Case statement](http://www.tutorialspoint.com/plsql/plsql_case_statement.htm) | 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. |
| [Searched CASE statement](http://www.tutorialspoint.com/plsql/plsql_searched_case.htm) | The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values. |
| [nested IF-](http://www.tutorialspoint.com/plsql/plsql_nested_if.htm)  [THEN-ELSE](http://www.tutorialspoint.com/plsql/plsql_nested_if.htm) | You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IFTHEN-ELSIF** statement(s). |

## PL/SQL Loops

PL/SQL provides the following types of loop to handle the looping requirements

|  |  |
| --- | --- |
| Loop Type  [Basic LOOP](http://www.tutorialspoint.com/plsql/plsql_basic_loop.htm) | Description |
| In this loop structure, sequence of statements is enclosed between the LOOP and END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop. |
| [WHILE LOOP](http://www.tutorialspoint.com/plsql/plsql_while_loop.htm) | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| [FOR LOOP](http://www.tutorialspoint.com/plsql/plsql_for_loop.htm) | Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| [Nested loops](http://www.tutorialspoint.com/plsql/plsql_nested_loops.htm) | You can use one or more loop inside any another basic loop, while or for loop. |
| Control Statement  [EXIT statement](http://www.tutorialspoint.com/plsql/plsql_exit_statement.htm) | Description |
| The Exit statement completes the loop and control passes to the statement immediately after END  LOOP |
| [CONTINUE statement](http://www.tutorialspoint.com/plsql/plsql_continue_statement.htm) | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| [GOTO statement](http://www.tutorialspoint.com/plsql/plsql_goto_statement.htm) | Transfers control to the labeled statement. Though it is not advised to use GOTO statement in your program. |

## PL/SQL Strings

The string 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**: 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 up to 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

## PL/SQL Strings

### 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;

## PL/SQL Strings Functions

|  |  |  |
| --- | --- | --- |
| S.N. |  | Function & Purpose |

1. **ASCII(x);** Returns the ASCII value of the character x.
2. **CHR(x);** Returns the character with the ASCII value of x.
3. **CONCAT(x, y);** Concatenates the strings x and y and return the appended string.
4. **INITCAP(x);** Converts the initial letter of each word in x to uppercase and returns that string.

**INSTR(x, find\_string [, start] [, occurrence]);** Searches for find\_string in x and returns the

5

position at which it occurs.

1. **INSTRB(x);** Returns the location of a string within another string, but returns the value in bytes.
2. **LENGTH(x);** Returns the number of characters in x.
3. **LENGTHB(x);** Returns the length of a character string in bytes for single byte character set.
4. **LOWER(x);** Converts the letters in x to lowercase and returns that string.

**LPAD(x, width [, pad\_string]) ;** Pads x with spaces to left, to bring the total length of the string

10

up to width characters.

11 **LTRIM(x [, trim\_string]);** Trims characters from the left of x.

**NANVL(x, value);** Returns value if x matches the NaN special value (not a number), otherwise x is

12

returned.

## PL/SQL Strings Functions

|  |  |
| --- | --- |
| S.N. | Function & Purpose |

**NLS\_INITCAP(x);** Same as the INITCAP function except that it can use a different sort method as

1. specified by NLSSORT.

**NLS\_LOWER(x) ;** Same as the LOWER function except that it can use a different sort method as

1. specified by NLSSORT.

**NLS\_UPPER(x);** Same as the UPPER function except that it can use a different sort method as

1. specified by NLSSORT.

**NLSSORT(x);** Changes the method of sorting the characters. Must be specified before any NLS

16

function; otherwise, the default sort will be used.

1. **NVL(x, value);** Returns value if x is null; otherwise, x is returned.
2. **NVL2(x, value1, value2);** Returns value1 if x is not null; if x is null, value2 is returned.

**REPLACE(x, search\_string, replace\_string);** Searches x for search\_string and replaces it with

19

replace\_string.

1. **RPAD(x, width [, pad\_string]);** Pads x to the right.
2. **RTRIM(x [, trim\_string]);** Trims x from the right.
3. **SOUNDEX(x) ;** Returns a string containing the phonetic representation of x.

**SUBSTR(x, start [, length]);** Returns a substring of x that begins at the position specified by start. An

23

optional length for the substring may be supplied.

**SUBSTRB(x);** Same as SUBSTR except the parameters are expressed in bytes instead of characters for

24

the single-byte character systems

1. **TRIM([trim\_char FROM) x);** Trims characters from the left and right of x.
2. **UPPER(x);** Converts the letters in x to uppercase and returns that string.

## PL/SQL String Example

DECLARE

greetings varchar2(11) := 'hello world';

### BEGIN

dbms\_output.put\_line(UPPER(greetings)); dbms\_output.put\_line(LOWER(greetings)); dbms\_output.put\_line(INITCAP(greetings));

/\* retrieve the first character in the string \*/ dbms\_output.put\_line ( SUBSTR (greetings, 1, 4));

/\* retrieve the last character in the string \*/ dbms\_output.put\_line ( SUBSTR (greetings, -1, 1));

/\* retrieve five characters, starting from the seventh position. \*/ dbms\_output.put\_line ( SUBSTR (greetings, 7, 5));

/\* retrieve the remainder of the string, starting from the second position. \*/ dbms\_output.put\_line ( SUBSTR (greetings, 2));

/\* find the location of the first "e" \*/

dbms\_output.put\_line ( INSTR (greetings, 'e')); END;

## PL/SQL Arrays using varray

It 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.

**Creating a Varray Type :**

It is created with the CREATE TYPE statement. And 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 <data\_type>**

## PL/SQL Arrays using varray

DECLARE

type namesarray IS VARRAY(5) OF VARCHAR2(10); type grades IS VARRAY(5) OF INTEGER; names namesarray; marks grades; total integer;

BEGIN

names := namesarray('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;

## PL/SQL Subprograms

It is a program unit/module that performs a particular task and combined to form larger programs

It can be invoked by another subprogram or program which is called the calling program.

A subprogram can be created:

* At schema/database level (Standalone subprogram)
* Inside a package (Packaged sub-programmed)
* Inside a PL/SQL block (Functions/Procedures)

### Parts of PL/SQL Subprograms

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 | **Declarative Part** (Optional). 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. |

**Executable Part** (Mandatory contains statements that perform the

2

designated action.

**Exception-handling** (Optional) contains the code that handles run-time

3

errors.

### Creating a Procedure

It created with the CREATE OR REPLACE PROCEDURE statement.

The syntax 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. | | |

## Creating a Procedure ‘greetings’

It created with the CREATE OR REPLACE PROCEDURE statement.

The syntax is as follows:

|  |
| --- |
| **CREATE OR REPLACE PROCEDURE greetings**  **AS**  **BEGIN dbms\_output.put\_line('Hello World!'); END;** |

**Executing a Standalone Procedure**

A standalone procedure can be called in two ways:

* + Using the EXECUTE keyword (for Eg : EXECUTE greetings;)
  + Calling the name of the procedure from a PL/SQL block

(BEGIN greetings; END; )

**Deleting a Standalone Procedure**

DROP PROCEDURE procedure-name;

i.e. for Eg:, DROP PROCEDURE greetings;

### Procedure ‘Example-1 : Minimum of two Numbers’

|  |
| --- |
| **DECLARE**  a number; b number; c number;  PROCEDURE findMin(x IN number, y IN number, z OUT number) IS  **BEGIN**  IF x < y THEN z:= x;  ELSE z:= y;  END IF;  **END;**  **BEGIN**  a:= 23; b:= 45; findMin(a, b, c);  dbms\_output.put\_line(' Minimum of (23, 45) : ' || c); **END;** |

### Procedure ‘Example-1 : Square of a Number’

|  |
| --- |
| **DECLARE**  a number;  PROCEDURE squareNum(x IN OUT number) IS  **BEGIN**  x := x \* x;  **END;**  **BEGIN**  a:= 23; squareNum(a);  dbms\_output.put\_line(' Square of (23): ' || a);  **END;** |

**Methods for Passing Parameters**

Actual parameters could be passed in three ways:

* Positional notation : for Eg : findmin(a,b,c);
* Named notation : for Eg : findmin(x=>a, y=>b, z=>c)
* Mixed notation : for Eg : findmin(x=>a,b,z=>c)

### PL/SQL Functions

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

|  |
| --- |
| **Creating a Function : It** is created using the CREATE FUNCTION statement. The syntax :  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. * The function must contain a **return** statement. * ***RETURN*** clause specifies that data type you are going to return from the function. * ***function-body*** contains the executable part. * The **AS** keyword is used instead of the IS keyword for creating a standalone function. |

### PL/SQL Functions Exampe-1

|  |
| --- |
| CREATE OR REPLACE FUNCTION totalCustomers RETURN number IS total number(2) := 0;  BEGIN  SELECT count(\*) into total FROM customers; RETURN total; END; |

**Calling a Function**

|  |
| --- |
| **DECLARE**  c number(2);  **BEGIN**  c := totalCustomers();  dbms\_output.put\_line('Total no. of Customers: ' || c); **END;** |

### PL/SQL Functions Example-2

DECLARE a number; b number; c number;

FUNCTION findMax(x IN number, y IN number) RETURN number IS z number;

BEGIN

IF x > y THEN z:= x; ELSE Z:= y;

END IF;

RETURN z;

END;

BEGIN

a:= 23; b:= 45; c := findMax(a, b); dbms\_output.put\_line('Maximum of (23,45): ' || c); END;

### PL/SQL Functions Example-3 (Recursive)

DECLARE num number;

factorial number;

**FUNCTION** fact(x number) RETURN number IS f number;

BEGIN

IF x=0 THEN f := 1;

ELSE f := x \* fact(x-1);

END IF;

RETURN f;

END;

BEGIN

num:= 6; factorial := fact(num);

dbms\_output.put\_line(' Factorial '|| num || ' is ' || factorial); END;

### 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 used to control it.
* 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 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 needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.
* In PL/SQL, can refer to the most recent implicit cursor as the **SQL cursor**, which always has the attributes like **%FOUND, %ISOPEN, %NOTFOUND, and**

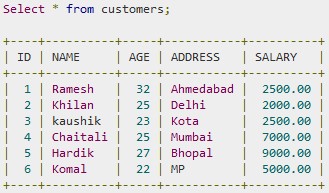
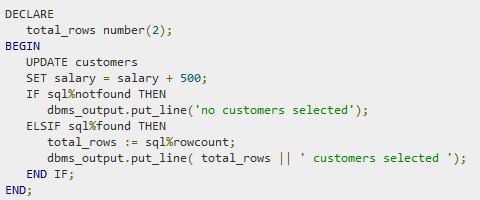
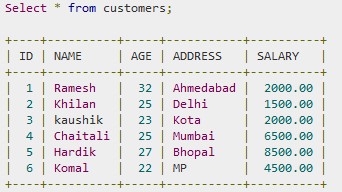
**%ROWCOUNT**.

* The SQL cursor has additional attributes, %BULK\_ROWCOUNT and

%BULK\_EXCEPTIONS, designed for use with the FORALL statement.

The following table provides the description of the most used attributes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | | **Description** | |
| %FOUND | Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE. | |
| %NOTFOUND | The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE. | |
| %ISOPEN | Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. | |
| %ROWCOUNT | Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. | |



* They are programmer defined cursors for gaining more control over the **context area**.
* It 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 curser\_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

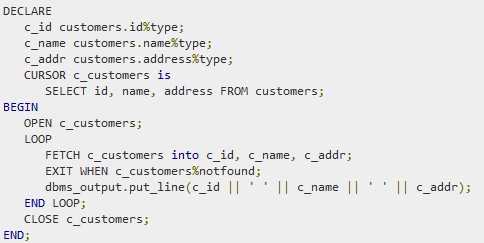
**Declaring the Cursor :** It defines the cursor with a name and the associated SELECT statement.

For eg: CURSOR c\_customers IS SELECT id, name, address FROM customers;

**Opening the Cursor :** It allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For eg: OPEN c\_customers;

**Fetching the Cursor :** It involves accessing one row at a time. For eg: FETCH c\_customers INTO c\_id, c\_name, c\_addr;

**Closing the Cursor :** It releasing the allocated memory. For eg:CLOSE c\_customers;

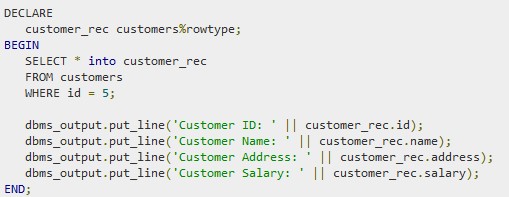


### PL/SQL Records

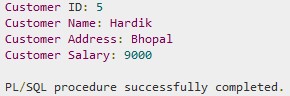
* It is a data structure that can hold data items of different kinds.
* It consist of different fields, similar to a row of a database table.
* For e.g., if we want to keep track of our books in a library, then the following attributes need to be tracked for each book like, Title, Author, Subject, Book ID.
* It containing a field for each of these items allows treating a BOOK as a logical unit and allows us to organize and represent its information in a better way.
* PL/SQL can handle the following types of records:
  + Table-based
  + Cursor-based records
  + User-defined records

### PL/SQL Table Based Records

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

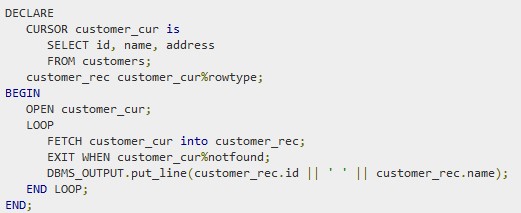


When the above code is executed, The output is

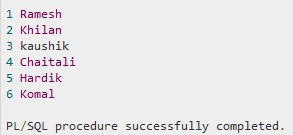


### PL/SQL Cursor Based Records

The following example would illustrate the concept of **cursor-based** records.



When the above code is executed, the output is

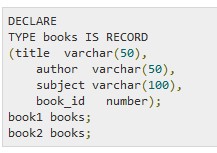
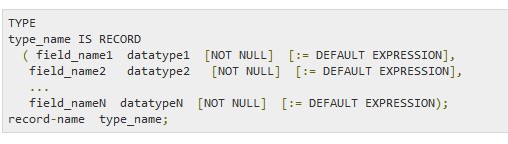


### PL/SQL User Defined Records

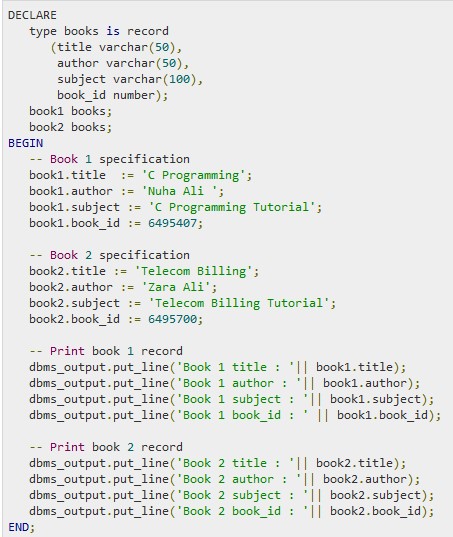
PL/SQL provides a user-defined record type that allows you to define different record structures. Records consist of different fields.

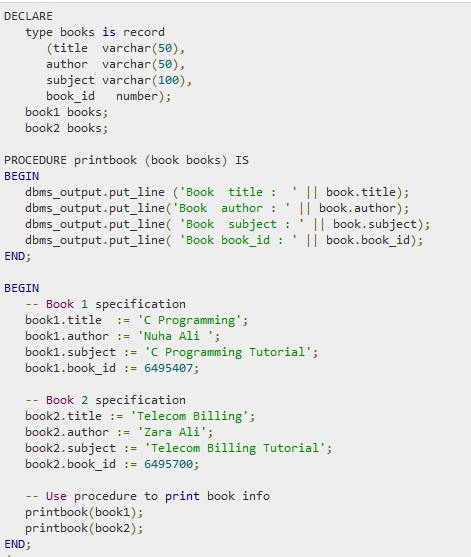
Suppose, if to keep track of books in a library, need to keep track of attributes about each book like, Title, Author, Subject, Book ID

**Defining a Record :** The record type is defined as:



#### PL/SQL Accessing Record 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.

**PL/SQL Record as**

**Subprogram**

**Parameters**

* 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:

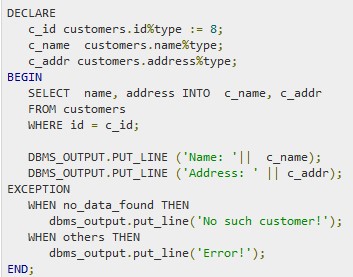
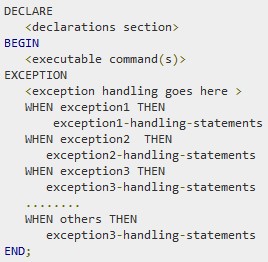
* System-defined exceptions
* User-defined exceptions

**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*:

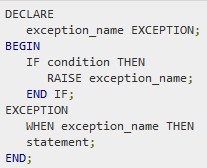
**Syntax for Exception Handling :** The Syntax for exception handling is as follows. Here you can list down as many as exceptions you want to handle.

The default exception will be handled using *WHEN others THEN*:



**Example**

**Raising Exceptions :** Exceptions are raised by the database server automatically, whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**.

Following is the simple syntax of raising an exception:

**User-defined Exceptions :** PL/SQL allows you to define your own exceptions according to the need of your program. It must be declared and then raised explicitly,

using either a RAISE statement or the procedure

DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR.

# PL/SQL Exceptions Example

* PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program.
* For example, the predefined exception NO\_DATA\_FOUND is raised when a SELECT INTO statement returns no rows.
* The following table lists few of the important pre-defined exceptions:

|  |  |  |  |
| --- | --- | --- | --- |
| Exception | Oracle  SQLCODE  Error | | Description |
| ACCESS\_IN TO\_NULL | 06530 | -6530 | It is raised when a null object is automatically assigned a value. |
| CASE\_NOT\_ FOUND | 06592 | -6592 | It is raised when none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause. |
| COLLECTIO N\_IS\_NULL | 06531 | -6531 | It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| DUP\_VAL\_O N\_INDEX | 00001 | -1 | It is raised when duplicate values are attempted to be stored in a column with unique index. |

|  |  |  |  |
| --- | --- | --- | --- |
| Exception | Oracle Error | SQLCODE | Description |
| INVALID\_C  URSOR | 01001 | -1001 | It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor. |
| INVALID\_N  UMBER | 01722 | -1722 | It is raised when the conversion of a character string into a number fails because the string does not represent a valid number. |
| LOGIN\_DEN IED | 01017 | -1017 | It is raised when s program attempts to log on to the database with an invalid username or password. |
| NO\_DATA\_F OUND | 01403 | +100 | It is raised when a SELECT INTO statement returns no rows. |
| NOT\_LOGG ED\_ON | 01012 | -1012 | It is raised when a database call is issued without being connected to the database. |
| PROGRAM\_  ERROR | 06501 | -6501 | It is raised when PL/SQL has an internal problem. |
| ROWTYPE\_  MISMATCH | 06504 | -6504 | It is raised when a cursor fetches value in a variable having incompatible data type. |

|  |  |  |  |
| --- | --- | --- | --- |
| Exception | Oracle Error | SQLCOD E | Description |
| SELF\_IS\_NULL | 30625 | -30625 | It is raised when a member method is invoked, but the instance of the object type was not initialized. |
| STORAGE\_ER ROR | 06500 | -6500 | It is raised when PL/SQL ran out of memory or memory was corrupted. |
| TOO\_MANY\_R OWS | 01422 | -1422 | It is raised when s SELECT INTO statement returns more than one row. |
| VALUE\_ERRO R | 06502 | -6502 | It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs. |
| ZERO\_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |

# PL/SQL Triggers

They are stored programs, automatically executed/fired when some events occur.

They are, in fact, written to be executed in response to any of the following events:

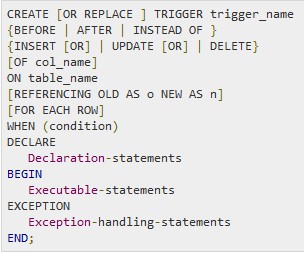
* A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
* A database definition (DDL) statement (CREATE, ALTER, or DROP).
* A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

It is defined on the table, view, schema, or database with which the event is associated.

**Benefits of Triggers**

* Generating some derived column values automatically
* Enforcing referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations
* Preventing invalid transactions

The syntax for creating a trigger is:

Where,

* CREATE [OR REPLACE] TRIGGER

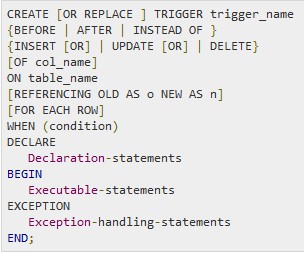
trigger\_name: Creates or replaces an existing trigger with the *trigger\_name*.

* {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.

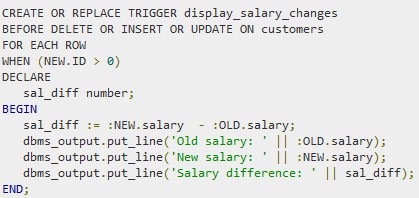
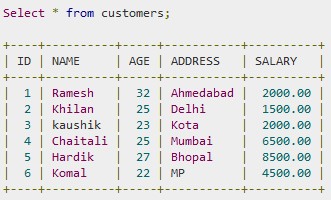
The syntax for creating a trigger is:

* [ON table\_name]: This specifies the name of the table associated with the trigger.
* [REFERENCING OLD AS o NEW AS n]:

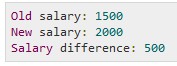
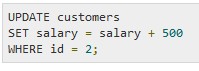
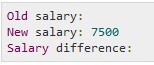
This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.

* [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.

Consider the Following Example The following program creates a **row level** trigger, that

would fire for INSERT/UPDATE/DELETE operations performed on the CUSTOMERS table. It will display the salary difference between the old values and new values:

# PL/SQL Triggering a Trigger

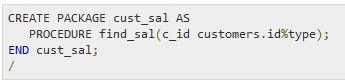


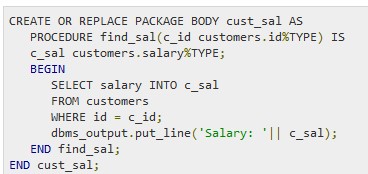
PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.

A package will have two mandatory parts:

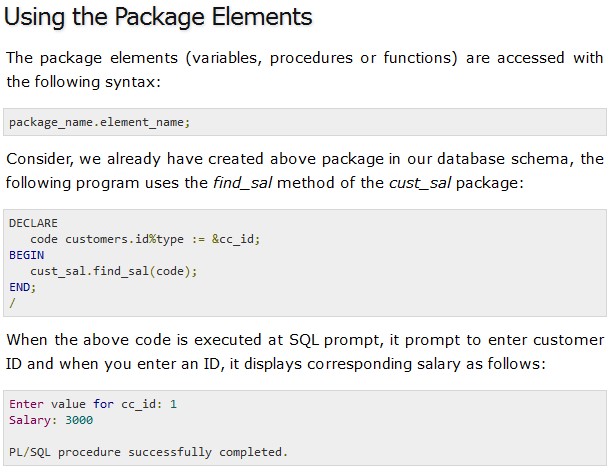
* Package specification
* Package body or definition

**Package Specification :** is the interface to the package.

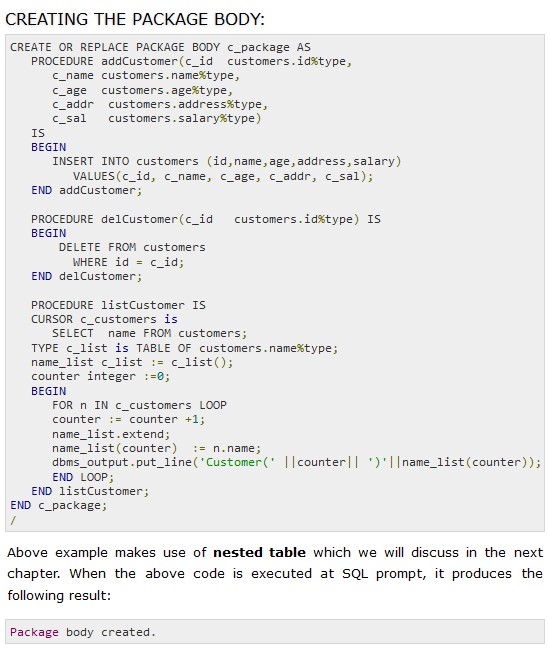
* It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package.
* It contains all information about the content of the package, but excludes the code for the subprograms.
* All objects placed in the specification are called **public** objects.
* Any subprogram not in the package specification but coded in the package body is called a **private** object.
* **Package Body : It** as 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.



# PL/SQL Using Packages



# PL/SQL Using Packages

**Creating the Package Body**

# Use of Packages in PL/SQL