In this chapter, we will discuss the Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts −

|  |  |
| --- | --- |
| **S.No** | **Sections & Description** |
| 1 | **Declarations**  This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program. |
| 2 | **Executable Commands**  This section is enclosed between the keywords **BEGIN** and **END** 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 | **Exception Handling**  This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program. |

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following 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 the 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 the SQL prompt, it produces the following result −

Hello World

PL/SQL procedure successfully completed.

**PL/SQL Program Units**

A PL/SQL unit is any one of the following −

* PL/SQL block
* Function
* Package
* Package body
* Procedure
* Trigger
* Type
* Type body

Each of these units will be discussed in the following chapters.

In this chapter, we will discuss the Data Types in PL/SQL. The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and a valid range of values. We will focus on the **SCALAR** and the **LOB** data types in this chapter. The other two data types will be covered in other chapters.

|  |  |
| --- | --- |
| **S.No** | **Category & Description** |
| 1 | **Scalar**  Single values with no internal components, such as a **NUMBER, DATE,** or **BOOLEAN**. |
| 2 | **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. |
| 3 | **Composite**  Data items that have internal components that can be accessed individually. For example, collections and records. |
| 4 | **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 −

|  |  |
| --- | --- |
| **S.No** | **Date Type & Description** |
| 1 | **Numeric**  Numeric values on which arithmetic operations are performed. |
| 2 | **Character**  Alphanumeric values that represent single characters or strings of characters. |
| 3 | **Boolean**  Logical values on which logical operations are performed. |
| 4 | **Datetime**  Dates and times. |

In this chapter, we will discuss Variables in Pl/SQL. A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and the layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The name of a PL/SQL variable consists 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 keyword as a variable name.

PL/SQL programming language allows to define various types of variables, such as date time data types, records, collections, etc. which we will cover in subsequent chapters. For this chapter, let us study only basic variable types.

In this chapter, we will discuss conditions in PL/SQL. 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.

Following is the general form of a typical conditional (i.e., decision making) structure found in most of the programming languages −



PL/SQL programming language provides following types of decision-making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **S.No** | **Statement & Description** |
| 1 | [IF - THEN statement](https://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. |
| 2 | [IF-THEN-ELSE statement](https://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. |
| 3 | [IF-THEN-ELSIF statement](https://www.tutorialspoint.com/plsql/plsql_if_then_elsif.htm)  It allows you to choose between several alternatives. |
| 4 | [Case statement](https://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. |
| 5 | [Searched CASE statement](https://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. |
| 6 | [nested IF-THEN-ELSE](https://www.tutorialspoint.com/plsql/plsql_nested_if.htm)  You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s). |

In this chapter, we will discuss Loops in PL/SQL. 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 form of a loop statement in most of the programming languages −



PL/SQL provides the following types of loop to handle the looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **S.No** | **Loop Type & Description** |
| 1 | [PL/SQL Basic LOOP](https://www.tutorialspoint.com/plsql/plsql_basic_loop.htm)  In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop. |
| 2 | [PL/SQL WHILE LOOP](https://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. |
| 3 | [PL/SQL FOR LOOP](https://www.tutorialspoint.com/plsql/plsql_for_loop.htm)  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 4 | [Nested loops in PL/SQL](https://www.tutorialspoint.com/plsql/plsql_nested_loops.htm)  You can use one or more loop inside any another basic loop, while, or for loop. |

**Subprogram, procedure, function**

In this chapter, we will discuss Procedures in PL/SQL. 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 the schema level**
* **Inside a package**
* **Inside a PL/SQL block**

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the 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**. We will discuss **PL/SQL function** in the next chapter.

## Parts of a PL/SQL Subprogram

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

|  |  |
| --- | --- |
| **S.No** | **Parts & Description** |
| 1 | **Declarative Part**  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 | **Executable Part**  This is a **mandatory** part and contains statements that perform the designated action. |
| 3 | **Exception-handling**  This is again an **optiona**l 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>**

**ENDprocedure\_name;**

Where,

* *procedure-name* specifies the name of the procedure.
* [OR REPLACE] option allows the modification of an existing procedure.
* The **optional parameter** list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that 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.**

### Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

**CREATE OR REPLACE PROCEDURE greetings**

**AS**

**BEGIN**

**dbms\_output.put\_line('Hello World!');**

**END;**

**/**

When the above code is executed using the SQL prompt, it will produce the following result −

Procedure created.

## Executing a Standalone Procedure

A standalone procedure can be called in two ways −

* Using the **EXECUTE** keyword
* Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

**EXECUTE greetings;**

The above call will display −

**Hello World**

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block −

**BEGIN**

**greetings;**

**END;**

/

The above call will display −

Hello World

PL/SQL procedure successfully completed.

## Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

**DROP PROCEDURE procedure-name;**

You can drop the greetings procedure by using the following statement −

DROP PROCEDURE greetings;

## Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −

|  |  |
| --- | --- |
| **S.No** | **Parameter Mode & Description** |
| 1 | **IN**  An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**. |
| 2 | **OUT**  An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**. |
| 3 | **IN OUT**  An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read.  The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.** |

### IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

**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;**

/

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

**Minimum of (23, 45) : 23**

PL/SQL procedure successfully completed.

### IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

**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;**

**/**

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

**Square of (23): 529**

PL/SQL procedure successfully completed.

## Methods for Passing Parameters

Actual parameters can be passed in three ways −

* **Positional notation**
* **Named notation**
* **Mixed notation**

### Positional Notation

In positional notation, you can call the procedure as −

**findMin(a, b, c, d);**

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x, b** is substituted for **y, c** is substituted for **z** and **d** is substituted for **m**.

### Named Notation

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol ( => )**. The procedure call will be like the following −

**findMin(x => a, y => b, z => c, m => d);**

### Mixed Notation

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal −

**findMin(a, b, c, m => d);**

**However, this is not legal:**

**findMin(x => a, b, c, d);**

In this chapter, we will discuss the functions in PL/SQL. A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

## 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 the modification of an existing function.
* The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
* The function must contain a **return** statement.
* The *RETURN* clause specifies the 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.

### Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the [PL/SQL Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) chapter −

**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 |**

**+----+----------+-----+-----------+----------+**

**CREATE OR REPLACE FUNCTION totalCustomers**

**RETURN number IS**

**total number(2):=0;**

**BEGIN**

**SELECT count(\*)into total**

**FROM customers;**

**RETURN total;**

**END;**

/

When the above code is executed using the SQL prompt, it will produce the following result −

Function created.

## Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block −

**DECLARE**

**c number(2);**

**BEGIN**

**c:=totalCustomers();**

**dbms\_output.put\_line('Total no. of Customers: '|| c);**

**END;**

/

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

Total no. of Customers: 6

PL/SQL procedure successfully completed.

### Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

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;

/

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

Maximum of (23,45): 45

PL/SQL procedure successfully completed.

## PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

n! = n\*(n-1)!

= n\*(n-1)\*(n-2)!

...

= n\*(n-1)\*(n-2)\*(n-3)... 1

The following program calculates the factorial of a given number by calling itself recursively −

**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;

/

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

Factorial 6 is 720

PL/SQL procedure successfully completed.

n this chapter, we will discuss the cursors in PL/SQL. Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the 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 needs 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 attributes such as **%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 −

|  |  |
| --- | --- |
| **S.No** | **Attribute & Description** |
| 1 | **%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. |
| 2 | **%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. |
| 3 | **%ISOPEN**  Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| 4 | **%ROWCOUNT**  Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |

Any SQL cursor attribute will be accessed as **sql%attribute\_name** as shown below in the example.

### Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

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 will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

DECLARE

total\_rows number(2);

BEGIN

UPDATE customers

SET salary = salary +500;

IF sql%notfound THEN

dbms\_output.put\_line('no customers selected');

ELSIF sql%found THEN

total\_rows:=sql%rowcount;

dbms\_output.put\_line(total\_rows||' customers selected ');

END IF;

END;

/

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

6 customers selected

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated −

Select \* from customers;

+----+----------+-----+-----------+----------+

**| ID | NAME | AGE | ADDRESS | SALARY |**

**+----+----------+-----+-----------+----------+**

**| 1 | Ramesh | 32 | Ahmedabad | 2500.00 |**

**| 2 | Khilan | 25 | Delhi | 2000.00 |**

**| 3 | kaushik | 23 | Kota | 2500.00 |**

**| 4 | Chaitali | 25 | Mumbai | 7000.00 |**

**| 5 | Hardik | 27 | Bhopal | 9000.00 |**

**| 6 | Komal | 22 | MP | 5000.00 |**

+----+----------+-----+-----------+----------+

## 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 includes the following steps −

* **Declaring the cursor for initializing the memory**
* **Opening the cursor for allocating the memory**
* **Fetching the cursor for retrieving the data**
* **Closing the cursor to release the 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 the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the 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-opened 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 the above-opened cursor as follows −

**CLOSE c\_customers;**

### Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

**DECLARE**

**c\_idcustomers.id%type;**

**c\_namecustomers.Name%type;**

**c\_addrcustomers.address%type;**

**CURSOR c\_customersis**

**SELECT id, name, address FROM customers;**

**BEGIN**

**OPEN c\_customers;**

**LOOP**

**FETCH c\_customers into c\_id,c\_name,c\_addr;**

**EXIT WHEN c\_customers%notfound;**

**dbms\_output.put\_line(c\_id||' '||c\_name||' '||c\_addr);**

**END LOOP;**

**CLOSE c\_customers;**

**END;**

/

When the above code is executed at the 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.

**EXCEPTIONS**

In this chapter, we will discuss Exceptions in PL/SQL. An exception is an error condition during a program execution. 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 exceptions as you can 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 a 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 the 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 the **EXCEPTION block**.

## 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 for raising an exception −

**DECLARE**

**exception\_name EXCEPTION;**

**BEGIN**

**IF condition THEN**

**RAISE exception\_name;**

**END IF;**

**EXCEPTION**

**WHEN exception\_name THEN**

**statement;**

**END;**

You can use the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

## User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a **RAISE statement** or the procedure **DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR**.

The syntax for declaring an exception is −

**DECLARE**

**my-exception EXCEPTION;**

### Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid\_id** is raised.

**DECLARE**

**c\_id customers.id%type:=&cc\_id;**

**c\_name customerS.Name%type;**

**c\_addr customers.address%type;**

**--userdefined exception**

**ex\_invalid\_id EXCEPTION;**

**BEGIN**

**IF c\_id<=0 THEN**

**RAISE ex\_invalid\_id;**

**ELSE**

**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);**

**END IF;**

**EXCEPTION**

**WHEN ex\_invalid\_id THEN**

**dbms\_output.put\_line('ID must be greater than zero!');**

**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 the SQL prompt, it produces the following result −

**Enter value for cc\_id: -6 (let's enter a value -6)**

**ID must be greater than zero!**

PL/SQL procedure successfully completed.

## Pre-defined Exceptions

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 Error** | **SQLCODE** | **Description** |
| ACCESS\_INTO\_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 clause of a CASE statement is selected, and there is no ELSE clause. |
| COLLECTION\_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\_ON\_INDEX | 00001 | -1 | It is raised when duplicate values are attempted to be stored in a column with unique index. |
| INVALID\_CURSOR | 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\_NUMBER | 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\_DENIED | 01017 | -1017 | It is raised when a program attempts to log on to the database with an invalid username or password. |
| NO\_DATA\_FOUND | 01403 | +100 | It is raised when a SELECT INTO statement returns no rows. |
| NOT\_LOGGED\_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. |
| 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\_ERROR | 06500 | -6500 | It is raised when PL/SQL ran out of memory or memory was corrupted. |
| TOO\_MANY\_ROWS | 01422 | -1422 | It is raised when a SELECT INTO statement returns more than one row. |
| VALUE\_ERROR | 06502 | -6502 | It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs. |
| ZERO\_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |

**TRIGGERS**In this chapter, we will discuss Triggers in PL/SQL. 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 can be **defined on the table, view, schema, or database** with **which the event is associated.**

### Benefits of Triggers

Triggers can be written for the following purposes −

* **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**

## 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 replaces an existing trigger with the *trigger\_name*.
* {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will 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 will 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**, such as INSERT, UPDATE, and DELETE.
* [FOR EACH ROW] − This specifies **a row-level trigger**, i.e., the trigger will 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 we had created and used in the previous chapters −

**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 the SQL prompt, it produces the following result −

**Trigger created.**

The following points need to be considered here −

* **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.**
* **The 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 the DELETE operation on the table.**

## Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

**INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)**

**VALUES (7,'Kriti',22,'HP',7500.00);**

When a record is created in the CUSTOMERS table, the above create trigger, **display\_salary\_changes** will be fired and it will display the following result −

Old salary:

New salary: 7500

Salary difference:

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The **UPDATE** statement will update an existing record in the table −

**UPDATE customers**

**SET salary = salary +500**

**WHERE id =2;**

When a record is updated in the CUSTOMERS table, the above create trigger, **display\_salary\_changes** will be fired and it will display the following result −

**Old salary: 1500**

**New salary: 2000**

**Salary difference: 500**

**Package**

In this chapter, we will discuss the Packages in 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\_idcustomers.id%type);**

**ENDcust\_sal;**

/

When the above code is executed at the SQL prompt, it produces the 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 the 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 the [PL/SQL - Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) chapter.

**CREATE OR REPLACE PACKAGE BODY cust\_sal AS**

**PROCEDURE find\_sal(c\_idcustomers.id%TYPE) IS**

**c\_salcustomers.salary%TYPE;**

**BEGIN**

**SELECT salary INTO c\_sal**

**FROM customers**

**WHERE id =c\_id;**

**dbms\_output.put\_line('Salary: '||c\_sal);**

**ENDfind\_sal;**

**ENDcust\_sal;**

/

When the above code is executed at the SQL prompt, it produces the 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 the above package in our database schema, the following program uses the ***find\_sal*** method of the ***cust\_sal***package −

**DECLARE**

**codecustomers.id%type:=&cc\_id;**

**BEGIN**

**cust\_sal.find\_sal(code);**

**END;**

/

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

**Enter value for cc\_id: 1**

**Salary: 3000**

PL/SQL procedure successfully completed.

### Example

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records −

Select \* from customers;

+----+----------+-----+-----------+----------+

**| ID | NAME | AGE | ADDRESS | SALARY |**

**+----+----------+-----+-----------+----------+**

**| 1 | Ramesh | 32 | Ahmedabad | 3000.00 |**

**| 2 | Khilan | 25 | Delhi | 3000.00 |**

**| 3 | kaushik | 23 | Kota | 3000.00 |**

**| 4 | Chaitali | 25 | Mumbai | 7500.00 |**

**| 5 | Hardik | 27 | Bhopal | 9500.00 |**

**| 6 | Komal | 22 | MP | 5500.00 |**

**+----+----------+-----+-----------+----------+**

### The Package Specification

**CREATE OR REPLACE PACKAGE c\_package AS**

**--Adds a customer**

**PROCEDURE addCustomer(c\_idcustomers.id%type,**

**c\_namecustomerS.No.ame%type,**

**c\_agecustomers.age%type,**

**c\_addrcustomers.address%type,**

**c\_salcustomers.salary%type);**

**--Removes a customer**

**PROCEDURE delCustomer(c\_idcustomers.id%TYPE);**

**--Lists all customers**

**PROCEDURE listCustomer;**

**ENDc\_package;**

**/**

When the above code is executed at the SQL prompt, it creates the above package and displays the following result −

Package created.

### Creating the Package Body

**CREATE OR REPLACE PACKAGE BODY c\_package AS**

**PROCEDURE addCustomer(c\_id customers.id%type,**

**c\_name customers.name%type,**

**c\_age customers.age%type,**

**c\_addr customers.address%type,**

**c\_sal customers.salary%type)**

**IS**

**BEGIN**

**INSERT INTO customers (id,name,age,address,salary)**

**VALUES(c\_id,c\_name,c\_age,c\_addr,c\_sal);**

**ENDaddCustomer;**

**PROCEDURE delCustomer(c\_idcustomers.id%type) IS**

**BEGIN**

**DELETE FROM customers**

**WHERE id =c\_id;**

**ENDdelCustomer;**

**PROCEDURE listCustomer IS**

**CURSOR c\_customersis**

**SELECT name FROM customers;**

**TYPE c\_listis TABLE OF customerS.No.ame%type;**

**name\_listc\_list:=c\_list();**

**counter integer :=0;**

**BEGIN**

**FOR n IN c\_customers LOOP**

**counter:= counter +1;**

**name\_list.extend;**

**name\_list(counter):= n.name;**

**dbms\_output.put\_line('Customer('||counter||')'||name\_list(counter));**

**END LOOP;**

**ENDlistCustomer;**

**ENDc\_package;**

/

The above example makes use of the **nested table**. We will discuss the concept of nested table in the next chapter.

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

Package body created.

### Using The Package

The following program uses the methods declared and defined in the package *c\_package*.

**DECLARE**

**codecustomers.id%type:=8;**

**BEGIN**

**c\_package.addcustomer(7,'Rajnish',25,'Chennai',3500);**

**c\_package.addcustomer(8,'Subham',32,'Delhi',7500);**

**c\_package.listcustomer;**

**c\_package.delcustomer(code);**

**c\_package.listcustomer;**

**END;**

/

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

**Customer(1): Ramesh**

**Customer(2): Khilan**

**Customer(3): kaushik**

**Customer(4): Chaitali**

**Customer(5): Hardik**

**Customer(6): Komal**

**Customer(7): Rajnish**

**Customer(8): Subham**

**Customer(1): Ramesh**

**Customer(2): Khilan**

**Customer(3): kaushik**

**Customer(4): Chaitali**

**Customer(5): Hardik**

**Customer(6): Komal**

**Customer(7): Rajnish**