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 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.

Declaring String Variables

- Various string datatypes:
  - CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an **'N'** are **'national character set'** datatypes, that store Unicode character data.

```
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 the SQL prompt, it produces the following result –

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

PL/SQL procedure successfully completed
```

PL/SQL String Functions and Operators

- PL/SQL offers the concatenation operator **(||)** for joining two strings. The following table provides the string functions provided by PL/SQL –

S.No	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 returns the appended string.
4	**INITCAP(x);** Converts the initial letter of each word in x to uppercase and returns that string.
5	**INSTR(x, find_string [, start] [, occurrence]);** Searches for **find_string** in x and returns the position at which it occurs.
6	**INSTRB(x);** Returns the location of a string within another string, but returns the value in bytes.
7	**LENGTH(x);** Returns the number of characters in x.
8	**LENGTHB(x);** Returns the length of a character string in bytes for single byte character set.

PL/SQL String Functions and Operators

9	**LOWER(x);** Converts the letters in x to lowercase and returns that string.
10	**LPAD(x, width [, pad_string]) ;** Pads **x** with spaces to the left, to bring the total length of the string up to width characters.
11	**LTRIM(x [, trim_string]);** Trims characters from the left of **x**.
12	**NANVL(x, value);** Returns value if x matches the NaN special value (not a number), otherwise **x** is returned.
13	**NLS_INITCAP(x);** Same as the INITCAP function except that it can use a different sort method as specified by NLSSORT.
14	**NLS_LOWER(x) ;** Same as the LOWER function except that it can use a different sort method as specified by NLSSORT.
15	**NLS_UPPER(x);** Same as the UPPER function except that it can use a different sort method as specified by NLSSORT.

PL/SQL String Functions and Operators

16	**NLSSORT(x);** Changes the method of sorting the characters. Must be specified before any NLS function; otherwise, the default sort will be used.
17	**NVL(x, value);** Returns value if **x** is null; otherwise, x is returned.
18	**NVL2(x, value1, value2);** Returns value1 if x is not null; if x is null, value2 is returned.
19	**REPLACE(x, search_string, replace_string);** Searches **x** for search_string and replaces it with replace_string.
20	**RPAD(x, width [, pad_string]);** Pads **x** to the right.
21	**RTRIM(x [, trim_string]);** Trims **x** from the right.
22	**SOUNDEX(x) ;** Returns a string containing the phonetic representation of **x**.
23	**SUBSTR(x, start [, length]);** Returns a substring of **x** that begins at the position specified by start. An optional length for the substring may be supplied.

PL/SQL String Functions and Operators

23	**SUBSTR(x, start [, length]);** Returns a substring of **x** that begins at the position specified by start. An optional length for the substring may be supplied.
24	**SUBSTRB(x);** Same as SUBSTR except that the parameters are expressed in bytes instead of characters for the single-byte character systems.
25	**TRIM([trim_char FROM) x);** Trims characters from the left and right of **x**.
26	**UPPER(x);** Converts the letters in x to uppercase and returns that string.

PL/SQL String Functions and Operators EXAMPLES

Example 1

```
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, 1));

    /* 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 String Functions and Operators EXAMPLES

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

```
HELLO WORLD
hello world
Hello World
h
d
World
ello World
2

PL/SQL procedure successfully completed.
```

PL/SQL String Functions and Operators EXAMPLES

- Example 2

```
DECLARE
   greetings varchar2(30) := '......Hello World.....';
BEGIN
   dbms_output.put_line(RTRIM(greetings,'.'));
   dbms_output.put_line(LTRIM(greetings, '.'));
   dbms_output.put_line(TRIM( '.' from greetings));
END;
```

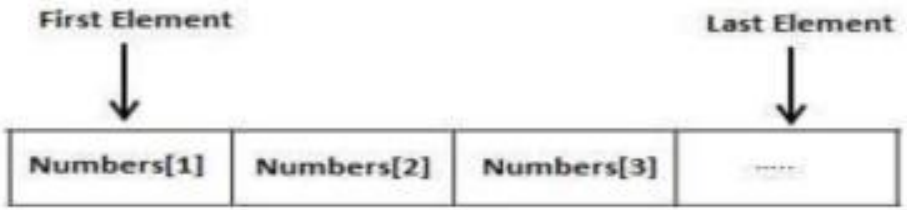
- Output

```
......Hello World
Hello World.....
Hello World

PL/SQL procedure successfully completed.
```

Arrays

- The 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, however it is often better 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.
- Each element in a **varray** has an index associated with it. It also has a maximum size that can be changed dynamically.

Arrays

Creating a Varray Type

- Syntax –

```
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.

Arrays Examples

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>
```

Example

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

Example 1

- Use of Arrays in PL/SQL

```
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;
```

Output Example 1

- When the above code is executed at the SQL prompt, it produces the 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.
```

Note −

- In Oracle environment, the starting index for varrays is always 1.
- You can initialize the varray elements using the constructor method of the varray type, which has the same name as the varray.
- Varrays are one-dimensional arrays.
- A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

Example 2

- Elements of a varray could also be a %ROWTYPE of any database table or %TYPE of any database table field.
- Consider 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 |
+----+----------+-----+-----------+----------+
```

Example 2

```
DECLARE
   CURSOR c_customers is
   SELECT  name FROM customers;
   type c_list is varray (6) of customers.name%type;
   name_list c_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;
END;
```

Example 2

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

PL/SQL procedure successfully completed.
```

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

Procedures in PL/SQL

- At the schema level, subprogram is a **standalone subprogram**.
- A subprogram created inside a package is a **packaged subprogram**.
- 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.

Procedures in PL/SQL

- Creating a Procedure Syntax-

```
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 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;
```

- Output

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

```
BEGIN
   greetings;
END;
```

- Output

```
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;
```

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.**

Example 1 IN/OUT mode

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;
```

Output

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

PL/SQL procedure successfully completed.
```

Example 2 IN/OUT mode

- 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;
```

- Output

```
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

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

Named notation

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

Mixed Notation

The following call is legal −

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

However, this is not legal:

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

Functions in PL/SQL

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

Creating a Function

```
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

Consider 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 |
+----+----------+-----+-----------+----------+
```

Example

CREATE FUNCTION

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

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

   RETURN total;
END;
```

Calling Functions

```
DECLARE
   c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
```

Output

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

Example 2

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;
```

- Output

```
Maximum of (23,45): 45

PL/SQL procedure successfully completed.
```

PL/SQL recursive functions example

- Factorial of a number

```
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;
```

- Output

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

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**.
- 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.

In PL/SQL, the most recent implicit cursor are the **SQL cursors**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**.

Implicit Cursors

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**

Implicit cursors

- 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;
```

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**.

Syntax-

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

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

Opening the Cursor

OPEN c_customers;

Fetching the Cursor

Fetching the cursor involves accessing one row at a time.

FETCH c_customers INTO c_id, c_name, c_addr;

Closing the Cursor

CLOSE c_customers;

Example

```
DECLARE
   c_id customers.id%type;
   c_name customer.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;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
```

Output

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

PL/SQL procedure successfully completed.
```

Records

A **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 the 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 **cursorbased** records.

```
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;
```

Cursor-Based Records

```
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;
```

Output

```
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 the different record structures. These records consist of different fields.

Defining a Record

```
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;
```

Declaring a Record

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

Accessing fields in the 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;
```

- Output

```
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.
```

Records as Subprogram Parameters

```
DECLARE
    type books is record
        (title  varchar(50),
        author  varchar(50),
        subject varchar(100),
        book_id    number);
    book1 books;
    book2 books;
PROCEDURE printbook (book books) IS
BEGIN
    dbms_output.put_line ('Book  title :  ' || book.title);
    dbms_output.put_line('Book  author :  ' || book.author);
    dbms_output.put_line( 'Book  subject :  ' || book.subject);
    dbms_output.put_line( 'Book book_id :  ' || book.book_id);
END;

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;

    -- Use procedure to print book info
    printbook(book1);
    printbook(book2);
END;
```

- Output

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

PL/SQL procedure successfully completed.
```

Exceptions

An exception is an error condition during a program execution.

There are two types of exceptions −

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

```
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

```
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;
```

Output

```
No such customer!

PL/SQL procedure successfully completed.
```

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**.

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   IF condition THEN
      RAISE exception_name;
   END IF;
EXCEPTION
   WHEN exception_name THEN
   statement;
END;
```

Example

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;
    -- user defined 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;
```

Output

```
Enter value for cc_id: -6 (let's enter a value -6)
old  2: c_id customers.id%type := &cc_id;
new  2: c_id customers.id%type := -6;
ID must be greater than zero!


PL/SQL procedure successfully completed.
```

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.

Pre-defined Exceptions

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

Triggers are stored programs, which are automatically executed or fired when some events occur.

Triggers can be 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

Creating Triggers

```
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

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;
```

Packages

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 **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package.
- 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.

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

Package Body

```
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;
```

Using the Package Elements

```
package_name.element_name;
```

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

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

PL/SQL procedure successfully completed.
```

Example

Package Specification

```
CREATE OR REPLACE PACKAGE c_package AS
   -- Adds a customer
   PROCEDURE addCustomer(c_id   customers.id%type,
   c_name  customerS.No.ame%type,
   c_age  customers.age%type,
   c_addr customers.address%type,
   c_sal  customers.salary%type);

   -- Removes a customer
   PROCEDURE delCustomer(c_id  customers.id%TYPE);
   --Lists all customers
   PROCEDURE listCustomer;

END c_package;
```

```
CREATE OR REPLACE PACKAGE BODY c_package AS
    PROCEDURE addCustomer(c_id  customers.id%type,
        c_name customerS.No.ame%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);
    END addCustomer;

    PROCEDURE delCustomer(c_id   customers.id%type) IS
    BEGIN
        DELETE FROM customers
        WHERE id = c_id;
    END delCustomer;

    PROCEDURE listCustomer IS
    CURSOR c_customers is
        SELECT  name FROM customers;
    TYPE c_list is TABLE OF customers.Name%type;
    name_list c_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;
    END listCustomer;

END c_package;
```

Using The Package

```
DECLARE
   code customers.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;
```

```
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

PL/SQL procedure successfully completed
```

If recompilation is successful then the package body becomes valid. If any errors are detected, the appropriate error messages are generated and the package body remains invalid.

Example 11:
Create a package comprising of a procedure and a function.
The function will:

- Accept the branch number and calculate the number of employees in that branch and finally return the number of employees

The procedure will:

- Accept the branch number
- Using the function created will get the employee count for the branch number accepted
- Based on the employee count a decision will be taken to delete the employees belonging to that branch followed by deleting the branch

Package Specification

```
CREATE OR REPLACE PACKAGE PCK_DEL IS
    PROCEDURE DEL_EMP_BRANCH(mBRANCH_NO VARCHAR2);
    FUNCTION CNT_EMP_BRANCH(mBRANCH_NO VARCHAR2) RETURN NUMBER;
END PCK_DEL;
```

Output:

```
Package created.
```

Package Body

```
CREATE OR REPLACE PACKAGE BODY PCK_DEL IS
    PROCEDURE DEL_EMP_BRANCH(mBRANCH_NO VARCHAR2) IS noemp NUMBER;
    BEGIN
        noemp := CNT_EMP_BRANCH(mBRANCH_NO);
        IF noemp < 2 AND noemp > 0 THEN
```

```
    DELETE EMP_MSTR WHERE BRANCH_NO = mBRANCH_NO;
    DBMS_OUTPUT.PUT_LINE('All the employees belonging to the branch ' ||
                         mBRANCH_NO || ' deleted sucessfully');
    DELETE BRANCH_MSTR WHERE BRANCH_NO = mBRANCH_NO;
    DBMS_OUTPUT.PUT_LINE('Branch ' || mBRANCH_NO || ' deleted sucessfully');
  END IF;
  IF noemp = 0 THEN
    DBMS_OUTPUT.PUT_LINE('There exist no employees in the branch.');
  END IF;
  IF noemp >= 2 THEN
    DBMS_OUTPUT.PUT_LINE('There exist ' || noemp || ' employees in the branch ' ||
                         mBRANCH_NO || ' Skipping Deletion.');

  END IF;
END;
FUNCTION CNT_EMP_BRANCH(mBRANCH_NO VARCHAR2) RETURN NUMBER IS
  noemp NUMBER;
BEGIN
  SELECT COUNT(*) INTO noemp FROM EMP_MSTR
    WHERE BRANCH_NO = mBRANCH_NO;
  RETURN noemp;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN 0;
END;
D PCK_DEL;
```

put:

```
kage body created.
```

Functionality Of The PL/SQL Block Of Code with the Package Will Be As Follows:

PCK_DEL based on the package definition cr

Calling The Package

Situation 1: When there are no employees in that branch.
The procedure defined in the above package can be executed as follows:
EXECUTE PCK_DEL. DEL_EMP_BRANCH ('B5');

Output:
```
There exist no employees in the branch.
PL/ SQL procedure successfully completed.
```

OR

CALL PCK_DEL. DEL_EMP_BRANCH ('B5');

Output:
```
There exist no employees in the branch.
Call completed.
```

Situation 2: When there are more than 2 employees in that branch.
The procedure defined in the above package can be executed as follows:
EXECUTE PCK_DEL. DEL_EMP_BRANCH ('B1');

Output:
```
There exist 2 employees in the branch B1 Skipping Deletion.
PL/ SQL procedure successfully completed.
```

OR

CALL PCK_DEL. DEL_EMP_BRANCH ('B1');

Output:
```
There exist 2 employees in the branch B1 Skipping Deletion.
Call completed.
```

Situation 3: When there are less than 2 employees in that branch.
The procedure defined in the above package can be executed as follows:
EXECUTE PCK_DEL. DEL_EMP_BRANCH ('B6');

Output:
```
All the employees belonging to the branch B6 deleted successfully.
PL/ SQL procedure successfully completed.
```

OR

CALL PCK_DEL. DEL_EMP_BRANCH ('B1');

VERLOADING PROCEDURES AND FUNCTIONS

package is an Oracle object that can hold a number of other objects like procedures and functions. More
an one procedure or function with the **same name** but with **different parameters** can be defined within a
kage or within a PL/SQL declaration block.

ultiple procedures that are declared with the same name are called **Overloaded Procedures**. Similarly,
ultiple Functions that are declared with the same name are called **Overloaded Functions**.

e code in the overloaded functions or overloaded procedures can be same or completely different.

ample 12:
eate a package to check that a numeric value is greater than zero, and a date is less than or equal to
date.

```
REATE OR REPLACE PACKAGE CHECK_FUNC IS
  FUNCTION VALUE_OK(DATE_IN IN DATE) RETURN VARCHAR2;
  FUNCTION VALUE_OK(NUMBER_IN IN NUMBER) RETURN VARCHAR2;
ND;
```

utput:

```
ckage created.
```

```
REATE OR REPLACE PACKAGE BODY CHECK_FUNC IS
  FUNCTION VALUE_OK (DATE_IN IN DATE) RETURN VARCHAR2 IS
  BEGIN
      IF DATE_IN <= SYSDATE THEN
          RETURN 'Output From the First Over loaded Function: TRUE';
      ELSE
          RETURN 'Output From the First Over loaded Function: FALSE';
      END IF;
  END;
  FUNCTION VALUE_OK (NUMBER_IN IN NUMBER) RETURN VARCHAR2 IS
  BEGIN
      IF NUMBER_IN > 0 THEN
          RETURN 'Output From the Second Over loaded Function: TRUE';
      ELSE
          RETURN 'Output From the Second Over loaded Function: FALSE';
      END IF;
  END;
ND;
```

Output:

```
ackage body created.
```

```
    IF NUMBER_IN > 0 THEN
        RETURN 'Output From the Second Over loaded Function: TRUE';
    ELSE
        RETURN ' Output From the Second Over loaded Function: FALSE';
    END IF;
END;
```

Overloading Built-In PL/SQL Functions And Procedures

PL/ SQL itself makes extensive use of overloading. An Example of an overloaded function is the TO_CHAR function. Function overloading allows developers to use a single function to convert numbers and dates to character format.

Example 13:

```
DATE_STRING := TO_CHAR(SYSDATE, 'DD/MM/YY');
NUMBER_STRING := TO_CHAR(10000, '$099,999');
```

The parameter name is replaced by the values sent to the objects when the package is called, so more than name or parameter mode in name do not offer a guide to the overloaded objects that must be used.

Example 16:
A procedure definition will be as:

```
CREATE OR REPLACE PACKAGE BODY CHECK_DATE IS
    FUNCTION VALUE_OK(DATE_IN IN DATE) RETURN BOOLEAN IS
    BEGIN
        RETURN DATE_IN <= SYSDATE;
    END;
    FUNCTION VALUE_OK(DATE_OUT IN DATE)RETURN BOOLEAN IS
    BEGIN
        RETURN DATE_OUT >= SYSDATE;
    END;
END;
```

The call to the function will be:

```
IS_DATE_OK := CHECK_DATA.VALUE_OK(TO_DATE('03-JAN-81'))
```

The name of the parameter is not available in the module call and thus PL/SQL interpreter cannot distinguish objects by name.

Similarly, even if a parameter in the first module is IN and the same parameter is IN OUT in another module, PL/SQL interpreter **cannot** distinguish using the package call.

overloading ... the following error messages:

LS-00307: too many declarations of 'value_check' match this call.

Overloaded functions must differ by more than their return data type.

the time that the overloaded function is called, the PL/SQL interpreter does not know what type of data function will return. The interpreter therefore cannot distinguish between different overloaded nctions based on the return data type.

ample 17:

```
REATE OR REPLACE PACKAGE BODY CHECK_RETURN IS
 FUNCTION VALUE_OK(DATE_IN IN DATE) RETURN BOOLEAN IS
 BEGIN
    RETURN DATE_IN <= SYSDATE ;
 END;
 FUNCTION VALUE_OK(DATE_OUT IN DATE ) RETURN NUMBER IS
 BEGIN
    IF DATE_OUT >= SYSDATE THEN
       RETURN 1;
    ELSE
       RETURN 0;
    END IF;
 END;
ND;
```

All the overloaded modules must be defined within the same PL/SQL scope or block (PL/SQL or package).

so modules cannot be overloaded across two PL/SQL blocks or across two packages.

ample 18:

```
OCEDURE DEVELOP_ANALYSIS (QUARTER_END_IN IN DATE, SALES_IN IN NUMBER) IS
 PROCEDURE REVISE_ESTIMATE (DATE_IN IN DATE) IS
 BEGIN
    PROCEDURE REVISE_ESTIMATE (DOLLAR_IN IN NUMBER) IS
    BEGIN
       ...
    END;
    BEGIN
       REVISE_ESTIMATE(QUARTER_END_IN);
       REVISE_ESTIMATE(DOLLARS_IN);
    END;
 END;
```

When the above code is interpreted, the PL/SQL interpreter displays the following error messa

```
rror in Line 12 / Column 3:
LS-00306: wrong number or type of arguments in call to
'REVISE_ESTIMATE'
```

message because the scope and visibility of both the procedu entire scope of the body DEVELO

Function Or Procedure Overloading

Example 19:
The bank manager decides to activate all those accounts, which were previously marked as inactive for performing no transactions in last 365 days.

Create a package spec and package body named **ACCT_MNTC** that includes two procedures of the same name. The procedure name is **ACT_ACCTS**. The first procedure accepts BRANCH_NO and the second procedure accepts branch name.

Package Specification:

```
CREATE OR REPLACE PACKAGE ACCT_MNTC IS
    PROCEDURE ACT_ACCTS(vBRANCH_NO IN NUMBER);
    PROCEDURE ACT_ACCTS (vNAME IN VARCHAR2);
END;
```

Output:

```
Package created.
```

Package Body:

```
CREATE OR REPLACE PACKAGE BODY ACCT_MNTC IS
    PROCEDURE ACT_ACCTS(vBRANCH_NO IN NUMBER) IS
    BEGIN
        UPDATE ACCT_MSTR SET STATUS = 'A'
            WHERE BRANCH_NO = 'B' || vBRANCH_NO AND STATUS = 'S';
        IF SQL%ROWCOUNT > 0 THEN
            DBMS_OUTPUT.PUT_LINE(TO_CHAR(SQL%ROWCOUNT) || ' Account(s) Activated
                                        Successfully');
        ELSE
            DBMS_OUTPUT.PUT_LINE('Currently there exist no Inactive Accounts in the branch no ' ||
                                        vBRANCH_NO);
        END IF;
    END;
    PROCEDURE ACT_ACCTS(vNAME IN VARCHAR2) IS
    BEGIN
        UPDATE ACCT_MSTR SET STATUS = 'A' WHERE STATUS = 'S'
            AND BRANCH_NO IN(SELECT BRANCH_NO FROM BRANCH_MSTR
                WHERE NAME = vNAME);
        IF SQL%ROWCOUNT > 0 THEN
            DBMS_OUTPUT.PUT_LINE(TO_CHAR(SQL%ROWCOUNT) || ' Account(s) Activated
                                        Successfully');
        ELSE
            DBMS_OUTPUT.PUT_LINE('Currently there exist no Inactive Accounts in the branch
                                        vNAME);
        END IF;
    END;
END ACCT_MNTC;
```

Output:

```
Package body created.
```

1. The data subtype of at least one of the parameters of the overloaded function or procedure must differ.

For example an overloaded procedure distinguished by parameters of different types of numeric data types is not allowed. Similarly, an overloaded procedure distinguished by parameters with varchar2 and char data types is not allowed.

Example 15:

```
CREATE OR REPLACE PACKAGE BODY STRING_FNS IS
    PROCEDURE TRIM_AND_CENTER (STRING_IN IN CHAR, STRING_OUT OUT CHAR)
    BEGIN
        . . .
    END;
    PROCEDURE TRIM_AND_CENTER (STRING_IN IN VARCHAR2, STRING_OUT OUT
                               VARCHAR2)

    BEGIN
        . . .
    END;
END;
```

Caution

Such procedure overloading is not allowed.

2. The parameter list of overloaded functions must differ by more than name or parameter mode.

The parameter name is replaced by the values sent to the objects when the package is called, so different in name do not offer a guide to the overloaded objects that must be used.