Birla Institute of Technology and Science, Pilani.
Database Systems
Lab No #8

Today's Topics

- Strings
- Arrays
- Procedures
- Functions
- Cursors
- Records

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.

PL/SQL strings could be either variables or literals. A string literal is enclosed within quotation marks. For example, 'This is a string literal.' Or 'hello world'

To include a single quote inside a string literal, you need to type two single quotes next to one another, like: 'this isn''t what it looks like'

Declaring String Variables

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

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

```
DECLARE
name varchar2(20);
company varchar2(30);
introduction clob;
choice char(1);
BEGIN
 name := 'John Smith';
```

```
company := 'Infotech';
introduction := ' Hello! I''m John Smith from Infotech.';    choice := 'y';
IF choice = 'y' THEN
dbms_output.put_line(name);       dbms_output.put_line(company);
dbms_output.put_line(introduction);
END IF;
 END;
```

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

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

PL/SQL 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.N	Functions	Purpose
1	**ASCII(x);**	Returns the ASCII value of the character x.
2	**CHR(x);**	Returns the character with the ASCII value of x.
3	**CONCAT(x, y);**	Concatenates the strings x and y and return the appended string.
4	**INITCAP(x);**	Converts the initial letter of each word in x to uppercase and returns that string.
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.
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 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.
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	**SOUNDEX(x) ;**	Returns a string containing the phonetic representation of x.
21	**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.
22	**SUBSTRB(x);**	Same as SUBSTR except the parameters are expressed in bytes instead of characters for the single-byte character systems.
23	**TRIM([trim_char FROM) x);**	Trims characters from the left and right of x.
24	**UPPER(x);**	Converts the letters in x to uppercase and returns that string.

The following examples illustrate some of the above-mentioned functions and their use:

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;
```

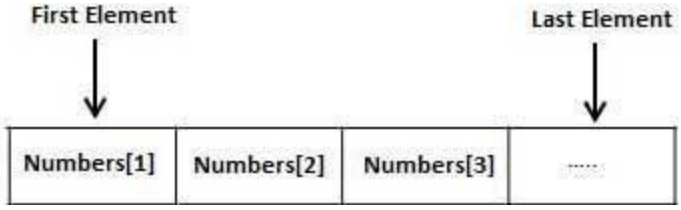
Example 2

```
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;
```

Arrays

PL/SQL programming language provides a data structure called the VARRAY, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



An array is a part of collection type data and it stands for variable-size arrays.
Each element in a varray has an index associated with it. It also has a maximum size that can be changed dynamically.

Creating a Varray Type

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

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

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

Where,

- varray_type_name is a valid attribute name,
- n is the number of elements (maximum) in the varray,
- element_type is the data type of the elements of the array.
- Maximum size of a varray can be changed using the ALTER TYPE statement.

Example:

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

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

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

For example:

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

Example 1

The following program illustrates using varrays:

```
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;
```

Please 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. The following example illustrates the concept:

Use the CUSTOMERS table as:

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

Following example makes use of cursor

```
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;
/
```

Procedures

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

A subprogram can be created:

- At schema level
- Inside a package
- Inside a PL/SQL block

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

A subprogram created inside a package* is a packaged subprogram. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement.

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.

** PL/SQL packages will be covered in next lab*

Parts of a PL/SQL Subprogram

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

S.N.	Parts	Description
1	**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 optional part. It contains the code that handles run-time errors.

Creating a Procedure

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

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

Where,

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

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;
```

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;
```

Deleting a Standalone Procedure

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

```
DROP PROCEDURE procedure-name;
```

So you can drop greetings procedure by using the following statement:

```
BEGIN
DROP PROCEDURE greetings;
END;
```

Parameter Modes in PL/SQL Subprograms

S.N.	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.**
2	**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 its 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 procedure takes two numbers using IN mode and returns their minimum using 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;
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use 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;
```

Methods for Passing Parameters

Actual parameters could 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 (=>). So the procedure call would look like:

```
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);
```

But this is not legal:

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

Functions

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

Creating a Function

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

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

Where,

- function-name specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- The function must contain a return statement.

- RETURN clause specifies that data type you are going to return from the function.

- function-body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example:

The following example illustrates creating and calling a standalone function. This function returns the total number of CUSTOMERS in the customers table. Use the CUSTOMERS table created earlier:

```
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;
```

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

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store 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;
```

Example:

The following is one more example which 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;
```

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;
/
```

Cursors

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

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

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

- Implicit cursors
- Explicit cursors

Implicit Cursors

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

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that 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 the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes:

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

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

Example:

Use the below CUSTOMERS table :

```
Select * from customers;
```

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

The following program would update the table and increase 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;
/
```

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

Explicit Cursors

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

The syntax for creating an explicit cursor is

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor involves four steps:

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

Declaring the Cursor

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

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

Opening the Cursor

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

```
OPEN c_customers;
```

Fetching the Cursor

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

```
CLOSE c_customers;
```

Example:

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

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

Records

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

For example, you want to keep track of your books in a library. You might want to track the following attributes about each book like, Title, Author, Subject, Book ID. A record containing a field for each of these items allows treating a BOOK as a logical unit and allows you to organize and represent its information in a better way.

PL/SQL can handle the following types of records:

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

Table-Based Records

The %ROWTYPE attribute enables a programmer to create table-based and cursor-based records. The following example would illustrate the concept of table-based records.

```
DECLARE
   customer_rec customers%rowtype;
BEGIN
   SELECT * into customer_rec FROM customers
   WHERE id = 5;
      dbms_output.put_line('CustomerID:'||customer_rec.id);
      dbms_output.put_line('CustomerName:'||customer_rec.name);
      dbms_output.put_line('CustomerAddress:'||customer_rec.address);
      dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
```

Cursor-Based Records

The following example would illustrate the concept of 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;
/
```

User-Defined Records

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

- Title
- Author
- Subject
- Book ID

Defining a Record

The record type is defined as:

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

Here is the way you would declare the Book record:

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

Accessing Fields

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

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

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

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

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

Records as Subprogram Parameters

You can pass a record as a subprogram parameter in very similar way as you pass any other variable. You would access the record fields in the similar way as you have accessed in the above example:

```
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;
```
