PL SQL 101, Database Developer, Database, SQL and PL SQL, DBA

Building with blocks in PL/SQL

September 1, 2020 | 17 minute read

Steven Feuerstein
Developer Advocate for PL/SQL

Part 1 in a series of articles on understanding and using PL/SQL for accessing Oracle Database

PL/SQL is one of the core technologies at Oracle and is essential to leveraging the full potential of Oracle Database. PL/SQL combines the relational data access capabilities of the Structured Query Language with a flexible embedded procedural language, and it executes complex queries and programmatic logic run inside the database engine itself. This enhances the agility, efficiency, and performance of database-driven applications.

Steven Feuerstein, one of the industry's best-respected and most prolific experts in PL/SQL, wrote a 12-part tutorial series on the language. Those articles, first published in 2011, have been among the most popular ever published on the Oracle website and continue to find new readers and enthusiasts in the database community. Beginning with the first installment below, the entire series is being updated and republished; please enjoy!

Oracle PL/SQL celebrates its 31st birthday in 2020. I know this because I am looking at the first Oracle PL/SQL user guide ever published; it is for PL/SQL Release 1.0, and its date of publication is September 1989. I worked for Oracle at that time, building the first sales automation tools ever used by the Oracle US sales force. I had already worked with PL/SQL inside SQL Forms 3.0, but with the release of Oracle 6 Database, PL/SQL was available as a freestanding application development language.

Three years later, I wrote my first book on PL/SQL and since then have done little professionally but study PL/SQL, write lots and lots of PL/SQL code, and write about this best-of-breed database programming language. Of course, I wasn't the only one. Thousands of developers around the world have been building a multitude of applications based on Oracle PL/SQL in the many decades since it was released.

To help newcomers to PL/SQL make the most of this language, *Oracle Connect* is republishing a series of articles that ran in *Oracle Magazine* beginning in 2011—this is the first in the series. These articles are primarily intended for PL/SQL beginners, but if you are an experienced PL/SQL developer, you may also find these articles a handy refresher on PL/SQL fundamentals.

I will assume for this series that the reader has some programming experience and is somewhat familiar with SQL itself. My approach throughout, in addition, will be on getting developers productive with PL/SQL as quickly as possible.

What is PL/SQL?

What is PL/SQL?

Every sophisticated website you visit, and every application you run, is constructed from a *stack* of software technologies. At the top of the stack is the presentation layer, the screens or interactive devices with which the user directly interacts. (A popular language for implementing presentation layers is Java.) At the very bottom of the stack is the operating system, that is, the machine code that communicates with the hardware.

Somewhere in the middle of the technology stack, you will find the *database* software that enables us to store and manipulate large volumes of complex data. Relational database technology, built around SQL, is the dominant database technology in the world today.

SQL, which is an acronym for *Structured Query Language*, is a very powerful, set-oriented language whose sole purpose is to manipulate the contents of relational databases. If you write applications built on Oracle Database, you (or someone writing code at a lower level in the technology stack) *must* be executing SQL statements to retrieve data from or change data in that database. Yet SQL cannot be used to implement all business logic and end user functionality needed in our applications. That brings us to PL/SQL.

PL/SQL stands for *Procedural Language/Structured Query Language*. PL/SQL offers a set of procedural commands (IF statements, loops, assignments), organized within blocks (explained below), that complement and extend the reach of SQL. PL/SQL is Oracle's extension of SQL designed for developers working with the Oracle Database.

It is certainly possible to build applications on top of SQL and Oracle Database *without* using PL/SQL. Using PL/SQL to perform database-specific operations, most notably SQL statement execution, offers several advantages, though, including tight integration with SQL, improved performance through reduced network traffic, and portability. (PL/SQL programs can run on any Oracle Database instance.) Thus, the front-end code of many applications executes both SQL statements and PL/SQL blocks to maximize performance while improving the maintainability of those applications.

Building blocks of PL/SQL programs

PL/SQL is a block-structured language. A PL/SQL block is defined by the keywords DECLARE, BEGIN, EXCEPTION, and END, which break up the block into three sections:

1. Declarative: Statements that declare variables, constants, and other code elements, which can then be used within that block

2. Executable: Statements that are run when the block is executed

3. Exception handling: A specially structured section you can use to "catch," or trap, any exceptions that are raised when the executable section runs

Only the executable section is required. You don't have to declare anything in a block, and you don't have to trap exceptions raised in that block.

A block itself is an executable statement, so you can nest blocks within other blocks.

Here are some examples:

The classic "Hello World!" block contains an executable section that calls the DBMS_OUTPUT.PUT_LINE procedure to display text on the screen:

Copy code snippet

```
BEGIN
  DBMS_OUTPUT.put_line ('Hello World!');
END;
```

Functions and procedures—types of named blocks—are discussed later in this article in more detail, as are packages. Briefly, however, a package is a container for multiple functions and procedures. Oracle extends PL/SQL with many supplied or built-in packages.

This next example block declares a variable of type VARCHAR2 (string) with a maximum length of 100 bytes to hold the string 'Hello World!'. DBMS_OUTPUT.PUT_LINE then accepts the variable, rather than the literal string, for display:

Copy code snippet

```
DECLARE
  l_message
  VARCHAR2 (100) := 'Hello World!';
BEGIN
  DBMS_OUTPUT.put_line (l_message);
END;
```

Note that I named the variable l_message. I generally use the l_ prefix for *local variables*—variables defined within a block of code—and the g_ prefix for *global variables* defined in a package.

This next example block adds an exception section that traps *any* exception (WHEN OTHERS) that might be raised and displays the error message, which is returned by the SQLERRM function (provided by Oracle):

Copy code snippet

```
DECLARE
  l_message
  VARCHAR2 (100) := 'Hello World!';
BEGIN
  DBMS_OUTPUT.put_line (l_message);
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.put_line (SQLERRM);
END;
```

The following example block demonstrates the PL/SQL ability to nest blocks within blocks as well as the use of the *concatenation* operator (||) to join together multiple strings:

Copy code snippet

```
DECLARE
  l_message
  VARCHAR2 (100) := 'Hello';
BEGIN
  DECLARE
    l_message2     VARCHAR2 (100) :=
      l_message || ' World!';
  BEGIN
    DBMS_OUTPUT.put_line (l_message2);
  END;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.put_line
   (DBMS_UTILITY.format_error_stack);
END;
```

Running PL/SQL blocks

Once you have written a block of PL/SQL code, you can execute (run) it. There are many different tools for executing PL/SQL code. The most basic is SQL*Plus, a command-line interface for executing SQL statements as well as PL/SQL blocks.

The first thing I do after connecting to the database through SQL*Plus is turn on server output, so that calls to DBMS_OUTPUT.PUT_LINE will result in the display of text on my screen. I then type in the code that constitutes my block. Finally, I enter a slash (/) to tell SQL*Plus to execute this block.

SQL*Plus then runs the block and displays “Hello World!” on the screen. SQL*Plus is provided by Oracle as a sort of lowest-common-denominator environment in which to execute SQL statements and PL/SQL blocks. Some developers continue to rely solely on SQL*Plus, but most use an integrated development environment (IDE).

Each tool offers slightly different windows and steps for creating, saving, and running PL/SQL blocks as well as for enabling and disabling server output. In this article series, I will assume only that you have access to SQL*Plus and that you will run all my statements in a SQL*Plus command window.

Name those blocks!

All the blocks I have shown you so far are “anonymous”—they have no names. If using anonymous blocks were the only way you could organize your statements, it would be very hard to use PL/SQL to build a large, complex application. Instead, PL/SQL supports the definition of *named* blocks of code, also known as *subprograms*. Subprograms can be procedures or functions. Generally, a procedure is used to perform an action and a function is used to calculate and return a value. I will explore subprograms in much greater detail in an upcoming article in this series. For now, let’s make sure you are comfortable with the basic concepts behind subprogram creation.

Suppose I need to display “Hello World!” from multiple places in my application. I very much want to avoid

repeating the same logic in all those places. For example, what happens when I need to change the message, perhaps to “Hello Universe!”? I would have to find all the locations in my code where this logic appears.

Instead, I will create a procedure named hello_world by executing the following data definition language (DDL) command:

Copy code snippet

```
CREATE OR REPLACE PROCEDURE
hello_world
IS
  l_message
  VARCHAR2 (100) := 'Hello World!';
BEGIN
  DBMS_OUTPUT.put_line (l_message);
END hello_world;
```

I have now, in effect, extended PL/SQL. In addition to calling programs created by Oracle and installed in the database (such as DBMS_OUTPUT.PUT_LINE), I can call my own subprogram inside a PL/SQL block:

Copy code snippet

```
BEGIN
   hello_world;
END;
```

I have hidden all the details of how I say hello to the world inside the *body*, or implementation, of my procedure. I can now call this hello_world procedure and have it display the desired message without having to write the call to DBMS_OUTPUT.PUT_LINE or figure out the correct way to format the string. I can call this procedure from any location in my application. So if I ever need to change that string, I will do so in one place, the *single point of definition* of that string.

The hello_world procedure is very simple. Your procedures will have lots more code inside them, and they will almost always also have *parameters*. Parameters pass information *into* subprograms when they are called, and they enable you to create subprograms that are more flexible and generic. They can be used in many different contexts.

I mentioned earlier that someday I might want to display “Hello Universe!” instead of “Hello World!” I *could* make a copy of my hello_world procedure and change the string it displays:

Copy code snippet

```
CREATE OR REPLACE PROCEDURE
hello_universe
IS
  l_message
  VARCHAR2 (100) := 'Hello Universe!';
```

```
  VARCHAR2 (100) := 'Hello universe.';
BEGIN
  DBMS_OUTPUT.put_line (l_message);
END hello_universe;
```

I could, however, end up with dozens of variations of the "same" hello procedure, which would make it very difficult to maintain my application. A much better approach is to analyze the procedure and identify which parts stay the same (are *static*) when the message needs to change and which parts change. I can then pass the changing parts as parameters and have a single procedure that can be used under different circumstances.

So I will change hello_world (and hello_universe) to a new procedure, hello_place:

Copy code snippet

```
CREATE OR REPLACE PROCEDURE
hello_place (place_in IN VARCHAR2)
IS
  l_message  VARCHAR2 (100);
BEGIN
  l_message  := 'Hello ' || place_in;
  DBMS_OUTPUT.put_line (l_message);
END hello_place;
```

Right after the name of the procedure, I add open and close parentheses, and inside them I provide a single parameter. I can have multiple parameters, but each parameter follows the same basic form:

Copy code snippet

```
parameter_name   parameter_mode   data_type
```

You must, in other words, provide a name for the parameter, the mode or way it will be used (IN = read only), and the type of data that will be passed to the subprogram through this parameter.

In this case, I am going to pass a string for read-only use to the hello_place procedure.

And I can now say hello to my world *and* my universe as follows:

Copy code snippet

```
BEGIN
   hello_place ('World');
   hello_place ('Universe');
END;
```

Later in this series, we will be exploring the concept of reuse and how to avoid repetition, but you should be able to see from this simple example the power of hiding logic behind a named block.

be able to see from this simple example the power of hiding logic behind a named block.

Now suppose I don't just want to display my "Hello" messages. Sometimes I need to save those messages in a database table; at other times, I must pass the string back to the host environment for display in a web browser. In other words, I need to separate the way the "Hello" message is constructed from the way it is used (displayed, saved, sent to another program, and so on). I can achieve this desired level of flexibility by moving the code that constructs the message into its own function:

Copy code snippet

```
CREATE OR REPLACE FUNCTION
hello_message
   (place_in IN VARCHAR2)
    RETURN VARCHAR2
IS
BEGIN
   RETURN 'Hello ' || place_in;
END hello_message;
```

This subprogram differs from the original procedure as follows:

- The type of program is now FUNCTION, not PROCEDURE.
- The subprogram name now describes the data being returned, not the action being taken.
- The body or implementation of the subprogram now contains a RETURN clause that constructs the message and passes it back to the calling block.
- The RETURN clause after the parameter list sets the type of data returned by the function.

With the code needed to construct the message inside the hello_message function, I can use this message in multiple ways. I can, for example, call the function to retrieve the message and assign it to a variable:

Copy code snippet

```
DECLARE
  l_message  VARCHAR2 (100);
BEGIN
  l_message := hello_message ('Universe');
END;
```

Note that I call the hello_message function as *part of* a PL/SQL statement (in this case, an assignment of a string to a variable). The hello_message function returns a string, so it can be used in place of a string in any executable statement.

I can also return to my hello_place procedure and replace the code used to build the string with a call to the function:

Copy code snippet

```
CREATE OR REPLACE PROCEDURE
hello_place (place_in IN VARCHAR2)
IS
BEGIN
  DBMS_OUTPUT.put_line
  (hello_message (place_in));
END hello_place;
```

I can also call the function from within a SQL statement. In the following block, I insert the message into a database table:

Copy code snippet

```
BEGIN
  INSERT INTO message_table (message_date, MESSAGE_TEXT)
      VALUES (SYSDATE, hello_message('Chicago'));
END;
```

Although the “hello place” logic is very simple, it demonstrates the power of assigning names to one or more executable statements (an algorithm) and then referencing that algorithm simply by specifying the name and any required parameters.

Named PL/SQL blocks make it possible to construct complex applications that can be understood and maintained with relative ease.

About names in Oracle Database

Now that you can see the importance of assigning names to logic, it is time to talk about the rules for names (or, to be more precise, identifiers) in both PL/SQL and, more generally, Oracle Database.

Here are the rules for constructing valid identifiers in Oracle Database:

- The maximum length is 30 characters.
- The first character must be a letter, but each character after the first can be either a letter, a numeral (0 through 9), a dollar sign ($), an underscore (_), or a number sign (#). All of the following are valid identifiers: hello_world
  hello$world
  hello#world But these are invalid:
  1hello_world
  hello%world
- PL/SQL is case-*insensitive* with regard to identifiers. PL/SQL treats all of the following as the same identifier: hello_world
  Hello_World
  HELLO_WORLD

To offer you increased flexibility, Oracle Database lets you bypass the restrictions of the second and third

rules by enclosing your identifier within double quotes. A *quoted identifier* can contain any sequence of printable characters excluding double quotes; differences in case will also be preserved. So all of the following strings are valid and distinct identifiers:

"Abc"
"ABC"
"a b c"

You will rarely encounter quoted identifiers in PL/SQL code; some development groups use them to conform to their naming conventions or because they find the mixed-case strings easier to read.

These same rules apply to the names of database objects such as tables, views, and procedures, with one additional rule: Unless you put double quotation marks around the names of those database objects, Oracle Database will store them as uppercase.

So when I create a procedure as follows:

Copy code snippet

```
CREATE OR REPLACE PROCEDURE
hello_world
IS
BEGIN
  DBMS_OUTPUT.put_line
  ('Hello World!');
END hello_world;
```

Oracle Database stores this procedure under the name HELLO_WORLD.

In the following block, I call this procedure three times, and although the name looks different in all the calls, they all execute the same procedure:

Copy code snippet

```
BEGIN
   hello_world;
   HELLO_WORLD;
   "HELLO_WORLD";
END;
```

Oracle Database will not, on the other hand, be able to run my procedure if I call it as follows:

Copy code snippet

```
BEGIN
   "hello_world";
END;
```

It will look inside the database for a procedure named hello_world rather than HELLO_WORLD.

If you don't want a subprogram name to be stored as uppercase, precede and follow that name with double quotation marks when you create the subprogram:

Copy code snippet

```
CREATE OR REPLACE PROCEDURE
"Hello_World"
IS
BEGIN
  DBMS_OUTPUT.put_line
  ('Hello World!');
END "Hello_World";
```

Running SQL inside PL/SQL blocks

PL/SQL is a database programming language. Almost all the programs you will ever write in PL/SQL will read from or write to—or read from and write to—Oracle Database by using SQL. Although this series assumes a working knowledge of SQL, you should be aware of the way you call SQL statements from within a PL/SQL block.

And here's some very good news: Oracle Database makes it very easy to write and run SQL statements in PL/SQL. For the most part, you simply write the SQL statement directly in your PL/SQL block and then add the code needed to interface between the SQL statement and the PL/SQL code.

Suppose, for example, that I have a table named employees, with a primary key column, employee_id, and a last_name column. I can then see the last name of the employee with ID 138, as follows:

Copy code snippet

```
SELECT last_name
  FROM employees
 WHERE employee_id = 138
```

Now I would like to run this same query inside my PL/SQL block and display the name. To do this, I need to "copy" the name from the table into a local variable, which I can do with the INTO clause:

Copy code snippet

```
DECLARE
  l_name  employees.last_name%TYPE;
BEGIN
  SELECT last_name
    INTO l_name
    FROM employees
   WHERE employee_id = 138;
```

```
   WHERE employee_id = 138;

  DBMS_OUTPUT.put_line (l_name);
END;
```

First I declare a local variable and in doing so introduce another elegant feature of PL/SQL: the ability to anchor the data type of my variable back to a table's column. (Anchoring is covered in more detail later in this series.)

I then execute a query against the database, retrieving the last name for the employee and depositing it directly into the l_name variable.

Of course, you will want to do more than run SELECT statements in PL/SQL—you will want to be able to insert into and update tables and delete from them in PL/SQL as well. Here are examples of each type of data manipulation language (DML) statement:

Delete all employees in department 10 and show how many rows were deleted:

Copy code snippet

```
DECLARE
  l_dept_id
  employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM employees
       WHERE department_id = l_dept_id;

  DBMS_OUTPUT.put_line (SQL%ROWCOUNT);
END;
```

Directly inside the DELETE statement, I reference the PL/SQL variable. When the block is executed, the variable name is replaced with the actual value, 10, and the DELETE is run by the SQL engine. SQL%ROWCOUNT is a special cursor attribute that returns the number of rows modified by the most recently executed DML statement in my session.

Update all employees in department 10 with a 20% salary increase.

Copy code snippet

```
DECLARE
  l_dept_id
  employees.department_id%TYPE := 10;
BEGIN
  UPDATE employees
    SET salary = salary * 1.2
   WHERE department_id = l_dept_id;

  DBMS_OUTPUT.put_line (SQL%ROWCOUNT);
END;
```

Insert a new employee into the table.

Copy code snippet

```
BEGIN
  INSERT INTO employees (employee_id
                       , last_name
                       , department_id
                       , salary)
       VALUES (100
             , 'Feuerstein'
             , 10
             , 200000);

  DBMS_OUTPUT.put_line (SQL%ROWCOUNT);
END;
```

In this block, I supply all the column values as literals, rather than variables, directly inside the SQL statement.

Next topic: Controlling block execution

In this article, you learned about how PL/SQL fits into the wider world of Oracle Database. You also learned how to define blocks of code that will execute PL/SQL statements and how to name those blocks so your application code can be more easily used and maintained. Finally, you were introduced to the execution of SQL statements inside PL/SQL. Now, let's talk about *nested blocks*.

Why would you nest blocks? You can put BEGIN before any set of one or more executable statements and follow it with END, creating a nested block for those statements. There are two key advantages of doing this:

1. Defer allocation of memory for variables needed only within that nested block.

2. Constrain the propagation of an exception raised by one of the statements in the nested block.

Consider the following block:

Copy code snippet

```
DECLARE
  l_message   VARCHAR2 (100) := 'Hello';
  l_message2  VARCHAR2 (100) := ' World!';
BEGIN
  IF SYSDATE >= TO_DATE ('01-JAN-2021')
  THEN
    l_message2 := l_message || l_message2;
    DBMS_OUTPUT.put_line (l_message2);
  ELSE
```

```
      DBMS_OUTPUT.put_line (l_message);
   END IF;
END;
```

This code displays “Hello World!” when today’s date (returned by SYSDATE) is at least the first day of the year 2021; otherwise, it displays only the “Hello” message. Yet even when this block is run in 2020, it allocates memory for the l_message2 variable.

If I restructure this block, the memory for l_message2 will be allocated only after 2020:

Copy code snippet

```
DECLARE
   l_message   VARCHAR2 (100) := 'Hello';
BEGIN
   IF SYSDATE > TO_DATE ('01-JAN-2021')
   THEN
      DECLARE
         l_message2   VARCHAR2 (100) := ' World!';
      BEGIN
         l_message2 := l_message || l_message2;
         DBMS_OUTPUT.put_line (l_message2);
      END;
   ELSE
      DBMS_OUTPUT.put_line (l_message);
   END IF;
END;
```

Similarly, I can add an exception section to that nested block, trapping errors and enabling the outer block to continue executing:

Copy code snippet

```
DECLARE
   l_message   VARCHAR2 (100) := 'Hello';
BEGIN
   DECLARE
      l_message2   VARCHAR2 (5);
   BEGIN
      l_message2 := 'World!';
      DBMS_OUTPUT.put_line (
         l_message || l_message2);
   EXCEPTION
      WHEN OTHERS
      THEN
         DBMS_OUTPUT.put_line (
            DBMS_UTILITY.format_error_stack);
   END;
   DBMS_OUTPUT.put_line (l_message);
```