



Share



DATABASE, SQL AND PL/SQL

Building with Blocks

Part 1 in a series of articles on understanding and using PL/SQL

By Steven Feuerstein 🌟

March/April 2011

Oracle PL/SQL celebrates its 22nd birthday in 2011. 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 was working for Oracle at that time, building the first sales automation tools ever used by the Oracle USA. 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 free-standing 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 decades since it was released.

Best of all, there continues to be a steady influx of new PL/SQL developers. In fact, with the relatively recent emergence of India, China, and other nations as technology powerhouses, I have seen a whole new generation of developers discover and work to master PL/SQL.

To help newcomers to PL/SQL make the most of this language, *Oracle Magazine* has asked me to write a series of articles for PL/SQL beginners, of which this is the first. 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 although my readers are new to PL/SQL, they have had some programming experience and are familiar with SQL. My approach throughout, in addition, will be on getting developers productive in PL/SQL as quickly as possible.

What Is PL/SQL?

To answer this question, it is important to remember that every Website you visit, 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. (These days the most popular languages for implementing presentation layers are Java and .NET.) At the very bottom of the stack 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 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.

It is certainly possible to build applications on top of SQL and Oracle Database *without* using PL/SQL. Utilizing 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:

```
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:

```
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).

```
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.

```
DECLARE
  l_message
```

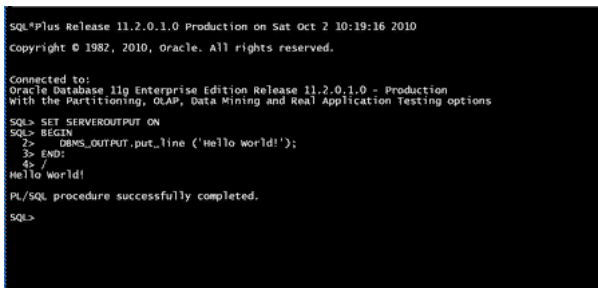
```

    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. Figure 1 shows an example of executing the simplest of my “Hello World!” example blocks in SQL*Plus.



```

SQL*Plus Release 11.2.0.1.0 Production on Sat Oct 2 10:19:16 2010
Copyright © 1982, 2010, Oracle. All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> SET SERVEROUTPUT ON
SQL> BEGIN
2>   DBMS_OUTPUT.put_line ('Hello world!');
3> END;
4> /
Hello World!
PL/SQL procedure successfully completed.
SQL>

```

Figure 1: Executing Hello World! in SQL*Plus

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. Whereas some developers continue to rely solely on SQL*Plus, most use an integrated development environment (IDE). Among the most popular of these IDEs (based on informal surveys I take in my training sessions) are

- Oracle SQL Developer, from Oracle
- Toad and SQL Navigator, from Quest Software
- PL/SQL Developer, from Allround Automations

Each tool offers slightly different windows and steps for creating, saving, and running PL/SQL blocks as well as 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 will 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:

```

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:

```
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:

```
CREATE OR REPLACE PROCEDURE
hello_universe
IS
    l_message
        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 will 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`:

```
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:

parameter_name parameter_mode datatype

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:

```
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.

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:

```
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:

```
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:

```
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:

```
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:

```

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:

```

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:

```

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 your subprogram names to be stored as uppercase, precede and follow that name with double quotation marks when you create the subprogram:

```

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:

```

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:

```

DECLARE
    l_name employees.last_name%TYPE;
BEGIN
    SELECT last_name
    INTO l_name
    FROM employees
    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 datatype 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:

```

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 percent salary increase.

```

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.

```

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.

Resources for PL/SQL Developers

If you are just getting started with PL/SQL, you should be aware of and take advantage of a multitude of free online resources. Here are some of the most popular and useful ones:

- Oracle PL/SQL home page: <https://oracle.com/plsql>. An excellent starting point for exploring PL/SQL resources for the Oracle Database team.
- Oracle Dev Gym: <https://devgym.oracle.com>. Classes, workouts and quizzes on PL/SQL, SQL, database design and other topics.
- AskTOM Office Hours: <https://asktom.oracle.com>. Monthly live training and how-to sessions on PL/SQL and other database areas, from experts on the Oracle Database team.
- My blog at stevenfeuersteinonplsql.blogspot.com. News you can use about the PL/SQL language
- Practically Perfect PLSQL: <https://www.youtube.com/c/PracticallyPerfectPLSQL>. My YouTube channel, full of hours of recordings on PL/SQL features and recommendations for writing applications built on PL/SQL.
- Oracle LiveSQL: <https://livesql.oracle.com>. 24x7 access to the latest Oracle Database, no installation needed. Write and run SQL and PL/SQL, learn from over 200 PL/SQL scripts I have uploaded there.

Next: 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 to name those blocks so that your application code can be more easily used and maintained. Finally, you were introduced to the execution of SQL statements inside PL/SQL.

Why 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, and (2) constrain the propagation of an exception raised by one of the statements in the nested block.

Consider the following block:

```

DECLARE
    l_message  VARCHAR2 (100) := 'Hello';
    l_message2 VARCHAR2 (100) := ' World!';
BEGIN

```

```

IF SYSDATE >= TO_DATE ('01-JAN-2011')
THEN
    l_message2 := l_message || l_message2;
    DBMS_OUTPUT.put_line (l_message2);
ELSE
    DBMS_OUTPUT.put_line (l_message);
END IF;
END;

```

It displays “Hello World!” when today’s date (returned by SYSDATE) is at least the first day of 2011; otherwise, it displays only the “Hello” message. Yet even when this block is run in 2010, it allocates memory for the l_message2 variable.

If I restructure this block, the memory for l_message2 will be allocated only after 2010:

```

DECLARE
    l_message    VARCHAR2 (100) := 'Hello';
BEGIN
    IF SYSDATE > TO_DATE ('01-JAN-2011')
    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:

```

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);
END;

```

In this case, the nested block will raise a VALUE_ERROR exception, because l_message2 is too small (maximum of 5 bytes) for the “World!” string. The exception section of the nested block will trap and display the error. The outer block will then continue executing.

A future article in this series will focus on how exception handling works in PL/SQL.

Next in this article series, I will show you how to control the flow of execution in your block: conditional logic with IF and CASE; iterative logic with FOR, WHILE, and simple loops; and raising and handling exceptions.

Next Steps

- READ [PL/SQL 101 parts 1-12](#)
- READ [PL/SQL column archive](#)
- DISCUSS [PL/SQL](#)
- DOWNLOAD [Oracle Database](#)
- TEST [your PL/SQL knowledge](#)

Photography by Ricardo Gomez, Unsplash

Steven Feuerstein

Steven Feuerstein is Oracle Corporation's Developer Advocate for PL/SQL, and an expert on the Oracle PL/SQL language, having written ten books on PL/SQL, including Oracle PL/SQL Programming and



Oracle PL/SQL Best Practices (all...



Related



DATABASE, SQL AND PL/SQL

PL/SQL Enhancements

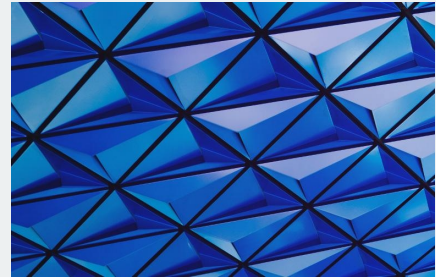
Oracle Database 12c enhances the PL/SQL function result cache, improves PL/SQL...



DATABASE, SQL AND PL/SQL

More on Oracle Database 11g

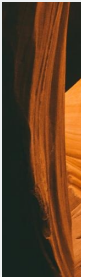
Our technologist takes another look at Oracle Database 11g.



DATABASE, SQL AND PL/SQL

On Emulating FINALLY

Code PL/SQL to function like the Java FINALLY section.



DATABAS

On Inv

Best prac



Contact Us



About Oracle



Cloud



Events



Top Tasks



News



Key Topics



ORACLE

Integrated Cloud & Platform Services

© Oracle | Site Maps | Terms of Use & Privacy | Cookie Preferences | Ad Choices