**Version History**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver. No.** | **Authors** | **Date** | **Reviewers** | **Review Date** | **Release Date** |
| 1.0 | Application Development Team | 27-Aug-2018 | QMF | 31-Aug-2018 | 03-Sep-2018 |
| 2.0 | Application Development Team | 10-Dec-2019 | QMF | 13-Dec-2019 | 16-Dec-2019 |
| 3.0 | Application Development Team | 02-Nov-2020 | QMF | 06-Nov-2020 | 10-Nov-2020 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Change History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Ver. No.** | **Section** | **Date** | **Change Information** | **RFC No.** |
| 1.0 | All | 03-Sep-2018 | New Release | - |
| 2.0 | All | 16-Dec-2019 | Annual Review | - |
| 3.0 | All | 02-Nov-2020 | Annual Review | - |
|  |  |  |  |  |
|  |  |  |  |  |

# STANDARDS FOR Naming Conventions and Coding Standards Oracle PL SQL

1. **Objective**

The objective of this document is to define the coding standards to be followed while developing application using ‘PL SQL’.

1. **Scope**

This standard applies to all products / projects developed using PL SQL.

1. **References to (checklists, forms, guidelines, lists, standards, Templates, other processes)**

| **Item** | **Description** | **ID** |
| --- | --- | --- |
| **Checklists** | -- | -- |

# 4.0 Principles Underlying Standards

* The most important factor in a name is that it concisely and accurately describes the significance of the identifier.
* If it's a variable, what is the description of the value?
* If it's a procedure, what does it do (and how can I describe it in 30 characters or less)?
* If it's a function, what does it return?
* "The algorithm or process should be visible in procedure names and invisible in function names.
* Normalize names: Avoid following conventions that introduce redundancy. For example, I never put the word "get" in front of a function name. After all every function gets *something* (returns it, anyway). So that prefix adds nothing and takes up valuable real estate.
* Upper-case non-application identifiers (that is, elements of the base PL/SQL language and Oracle built-ins) and lower-case application-specific elements. Camel notation (maxSalary) is something to be avoided in case- insensitive PL/SQL. Auto-formatters will likely destroy all your hard and careful naming work.
* Make plural anything that contains multiple pieces of information: relational tables and collections, for the most part. The table containing orders would be called *orders*, but a variable holding a single row of this table might be named *l\_order*.
* If the value of a variable will not (should not) change within the scope in which it is defined, I declare it to be a constant and use a c\_ or gc\_ prefix to indicate that it is a constant.
* Whenever the variable is based on (with %TYPE or %ROWTYPE) and/or contains data retrieved from a table or column in a table, the "root name" of the identifier (that is the name, absent prefixes and suffixes) should be the same as the element from which it is defined. If, for example, I use employees.first\_name%TYPE to declare my variable, then I will name it l\_first\_name.

# 5.0 Table of Naming Conventions

Key to table:

* + "SC" indicates the scope prefix, either l\_ for local or g\_ for global.
  + [] indicates optional, as in: Sure, I should do it, but I rarely do.
  + <rootname> is the root name of the identifier, the part of the name that describes the meaning of the thing named.

|  |  |  |
| --- | --- | --- |
| **Type of element** | **Naming convention** | **Example** |
| Variable declared in a PL/SQL block (anonymous, nested, subprogram) | l\_<rootname> | l\_total\_sales |
| Constant declared in a PL/SQL block (anonymous, nested, subprogram) | c\_<rootname> | c\_max\_salary |
| Variable declared at the package level | g\_<rootname> | g\_total\_sales |
| Constant declared at the package level | gc\_<rootname> | gc\_max\_salary |
| Explicit cursor | [SC\_]<rootname>\_cur | employees\_cur l\_employees\_cur  g\_employees \_cur |
| Cursor variables | [SC\_]<rootname>\_cv | name\_and\_salary\_cv |
| Record types | [SC\_]<rootname>\_rt | name\_and\_salary\_rt |
| Record variable | Same as local or global variable, singular form.  Sometimes I used a suffix like "\_rec" or "\_info" to indicate it’s a *bunch* of information, not just a single value. | l\_employee l\_employee\_info l\_employee\_rec |
| Collection types | [SC\_]<rootname>\_aat [SC\_]<rootname>\_nt [SC\_]<rootname>\_vat  or sometimes just | employees\_aat employees\_nt employees\_vat |
|  | [SC\_]<rootname>\_t |  |
| Collection variable | Same as local or global variable, plural form | l\_employees |
| Object type | [SC\_]<rootname>\_ot  or [SC\_]<rootname>t | employee\_ot |
| IN parameter | <rootname>\_in or  <rootname>\_i | salary\_in |
| OUT parameter | <rootname>\_out or  <rootname>\_o | salary\_out |
| IN OUT parameter | <rootname>\_inout or  <rootname>\_io | salary\_inout  That's a bit verbose, I know. |
| Exception | e\_<rootname> | e\_balance\_too\_low |
| Exception number (a constant that is assigned the value of an error code) | en\_<rootname> | Here is an example of declaring an exception, the error number that goes with it, and associating the two together with the EXCEPTION\_INIT pragma:  e\_balance\_too\_low EXCEPTION;  en\_balance\_too\_low CONSTANT PLS\_INTEGER :=  -20555;  PRAGMA EXCEPTION\_INIT  (e\_balance\_too\_low, -20555); |

# 6.0 General Guidelines to Follow

* + Avoid using names like "i" and "j" for variables. Sure, we all pretty much know what they mean (integer iterators, for the most part), but they are still obscure and do not enhance the readability of my code.
  + Avoid using the names of any elements defined in the STANDARD and DBMS\_STANDARD package. These are the two default packages of PL/SQL. Many of the names you might think are *reserved*, are rather simply elements defined in STANDARD, like *integer, sysdate* and *no\_data\_found.* You could declare variables with these same names, but the resulting code will be very confusing.
  + Do not "recycle" names. I might have a program in which I perform a series of calculations, all involving integers. I *could* declare a single variable like this: l\_integer PLS\_INTEGER;

But the code will be hard to understand. Declare distinct variables, exceptions, etc. so that the names are specific and relevant to the task at hand. Most definitely do *not* do what Oracle did with NO\_DATA\_FOUND: raise it for a SELECT INTO that returns no rows, an attempt to read an element in a collection at an index value that is undefined, an attempt to read past the end of a file. How silly!

* + Avoid repeating variable names at different scopes (especially nested and especially parameters): this can easily and often happen with cut and paste, and procedure extraction. It is, I suppose, a violation of my "Don't Recycle" rule, but it's more of an accidental repurposing. The problem with having the same name used for multiple, nested levels is that unless you *qualify* each reference (which is not a common practice), you can easily end up referencing the "wrong" variable or parameter.
  + Qualify all column names and variables names inside SQL statements in PL/SQL. This is valuable not only for making the code more readable and guarding against really hard to pin down bugs (when, for example, the DBA adds a new column to the table called "employee\_id\_in," which just so happens to match the naming convention you use for parameters), but also to take advantage of Oracle Database 11g's fine-grained dependency feature to minimize the invalidation of program units when referenced objects are changed.
  + Add labels to the END statements of all your packages, procedures, functions, etc. It is a small thing that greatly aids in readability.
  + Rely on a single, generic error manager utility to raise, handle and log errors. Individual developers should *never* waste their time calling DBMS\_UTILITY.FORMAT\_ERROR\_BACKTRACE (though you should *definitely* know what that is!) or writing inserts into log tables. You end with total

chaos and inconsistent information for both users and support. The best way to avoid this is to rely on a single utility. If you don't already have one, check out the freeware Quest Error Manager, currently available at [http://www.ToadWorld.com](http://www.ToadWorld.com/) (press Download button and then "Exclusive ToadWorld Downloads).

* + Stop writing so much SQL! Every SQL statement is a hard-coding of the current data structures. Don't repeat the same logical SQL statement. Keep SQL *out* of application-level code entirely. Instead, generate/build and rely on APIs (table- level, transaction-level) that *hide* SQL statements behind procedures and functions. One way to accomplish this: download the Quest CodeGen Utility from ToadWorld.com (press Download button and then "Exclusive ToadWorld Downloads). CodeGen generates API packages for you – and it's free!
  + Write tiny, little chunks of code (). Use top-down design, combined with reusable code and local subprograms (procedures and functions declared within *another* procedure or function), to make sure that your executable sections have no more than 50 lines of code in them. Define your subprograms at the package level if they need to be used by more than one program in that package or outside of that package.

# 7.0 Naming Conventions I Reject

Some common naming conventions rub me the wrong way.

* + Start your parameters with "p\_". Redundant. If you follow my approach of using a suffix to indicate the parameter mode (\_i, \_o, \_io) then you can see at a glance that it is a parameter *and* you know how it can be used in your program.
  + Start your functions with "get\_". Redundant. I mentioned this earlier. I consider it redundant information that uses up valuable real estate. The whole point of a function is return (that is, *get*) a piece of information – that's why it has a RETURN clause. Side note: I also recommend that you avoid any OUT or IN OUT parameters for a function – it should return information *only* through the RETURN clause. If the function needs to return multiple values, group those values together as a record type and return a record based on that type. If such a transformation seems unnatural then maybe you should be writing a procedure.
  + Start your procedures with "p\_" (or "sp\_" for stored procedure), start your functions with "f\_" (or "sf\_" for stored function),. Again, redundant. The way in which you invoke your subprogram in your code will unambiguously reveal if it is a procedure or a function.
  + Include an indicator of the datatype in the name, as in "g\_i\_counter", in which the "i" indicates that the datatype is an integer. I suppose that this information might sometimes come in handy, but I have these concerns: it uses up some more of our precious real estate; what if I change the datatype? Then I have to change the name; surely you should be comfortable enough with the code to have a basic understanding of the types of things with which you working.

# 8.0 How to Verify Naming Conventions and Standards

Here are some ideas for analyzing code to check with standards compliance:

* + Automate the process whenever possible. Many of the better PL/SQL editors include features that automatically analyze your code for compliance with best practices (though none yet validate compliance with naming conventions). Check out CodeXpert in Toad and SQL Navigator for an example.
  + Write queries against the ALL\_SOURCE data dictionary view. This view contains all the source code of the programs on which you have execute authority (USER\_SOURCE contains all the source code of the programs you own). It won't be easy to validate naming standards with a simply query against source, but you can certainly check for violations of some rules (for example, "In our application, we call the Quest Error Manager q$error\_manager package to log and raise exceptions, so RAISE\_APPLICATION\_ERROR should never appear in our code.").
  + Download and install Oracle Database 11g and check out the new PL/Scope feature. When you turn on PL/Scope in your session and compile your code, Oracle loads tons of information about your identifiers (the named elements of your code) into the ALL\_IDENTIFIERS view. You can then write queries against this view, just like against ALL\_SOURCE, but those queries can be so much more intelligent and productive. I offer a few examples below. Hopefully over time, tools like Toad will provide user interfaces to this fantastic feature.

# 9.0 PL/SQL Examples

1. Enable gathering of PL/Scope information:

ALTER SESSION SET plscope\_settings='IDENTIFIERS:ALL'

/

1. Show me all the declarations of variables for the specified program:

SELECT a.name variable\_name, b.name context\_name, a.signature FROM user\_identifiers a, user\_identifiers b

WHERE a.usage\_context\_id = b.usage\_id AND a.TYPE = 'VARIABLE'

AND a.usage = 'DECLARATION'

AND a.object\_name = '&1'

AND a.object\_name = b.object\_name ORDER BY a.object\_type, a.usage\_id

1. Finally, to show you what *really* is possible with PL/Scope, I offer a query to validate the following naming convention violations (and, yes, I realize that they do not match mine):
   * Type definitions should be named starting with t\_
   * Names of global (package level) variables should start with g\_
   * Parameters are named p\_<parameter description>
   * Local variables have names starting with l\_
   * Variable and parameter names should be written in lowercase
   * No variables should be declared in the package specification

This was written by Lucas Jellema of the AMIS consulting firm. His complete explanation is available on AMIS's fantastic blog:

[**http://technology.amis.nl/blog/2584/enforcing-plsql-naming-conventions-**](http://technology.amis.nl/blog/2584/enforcing-plsql-naming-conventions-) **through-a-simple-sql-query-using-oracle-11g-plscope**

WITH identifiers

AS (SELECT i.name

, i.TYPE

, i.usage

, s.line

, i.object\_type

, i.object\_name

, s.text source

FROM user\_identifiers i JOIN

user\_source s

ON ( s.name = i.object\_name AND s.TYPE = i.object\_type AND s.line = i.line)

WHERE object\_name = '&1'), global\_section

AS ( SELECT MIN (line) end\_line, object\_name FROM identifiers

WHERE object\_type = 'PACKAGE BODY'

AND TYPE IN ('PROCEDURE', 'FUNCTION')

GROUP BY object\_name), naming\_convention\_violations

AS (SELECT name identifier

, 'line ' || line || ': ' || source sourceline

, CASE

WHEN TYPE = 'RECORD'

AND usage = 'DECLARATION'

AND SUBSTR (LOWER (name), 1, 2) <> 't\_'

THEN

'Violated convention that type definitions should be called t\_<name>'

WHEN TYPE IN

('FORMAL IN', 'FORMAL IN OUT', 'FORMAL

OUT')

THEN

AND usage = 'DECLARATION'

AND SUBSTR (LOWER (name), 1, 2) <> 'p\_'

'Violated convention that (input and output) parameters should be called p\_<name>'

WHEN TYPE = 'VARIABLE' AND usage = 'DECLARATION' THEN

CASE

WHEN line < global\_section.end\_line /\*

global variable \*/

SUBSTR (LOWER (name), 1, 2) <> 'g\_'

THEN

AND

'Violated convention that global variables should be called g\_<name>'

WHEN line > global\_section.end\_line /\* local

variable \*/

SUBSTR (LOWER (name), 1, 2) <> 'l\_'

THEN

AND

should be called l\_<name>'

END

'Violated convention that local variables

END

MESSAGE

FROM identifiers JOIN

global\_section USING (object\_name)),

global\_violations

AS (SELECT name identifier

, 'line ' || line || ': ' || source sourceline

, CASE

WHEN TYPE = 'VARIABLE'

AND usage = 'DECLARATION'

AND object\_type = 'PACKAGE'

THEN

'Violated convention that there should not be any Global Variables in a Package Specification'

END

MESSAGE

FROM identifiers), casing\_violations

AS (SELECT name identifier

, 'line ' || line || ': ' || source sourceline

, CASE

WHEN TYPE = 'VARIABLE'

AND usage = 'DECLARATION'

AND INSTR (source, LOWER (name)) = 0

THEN

'Violated convention that variable names should spelled in lowercase only'

END

MESSAGE

FROM identifiers), convention\_violations AS (SELECT \*

FROM naming\_convention\_violations UNION ALL

SELECT \*

FROM global\_violations UNION ALL

SELECT \*

FROM casing\_violations)

SELECT \*

FROM convention\_violations WHERE MESSAGE IS NOT NULL

/