Ideas for Oracle PL/SQL
Naming Conventions and Coding Standards

Author: Steven Feuerstein, steven.feuerstein@oracle.com

Contents

Introduction

I am often asked about the naming conventions and coding standards that I use. My answer is usually a combination of muttering and vague statements and hand-waving.

That's because I have a confession: I don't have a single naming conventions and coding standards document that I use. Why not? Because I am a software developer! That is, I feel (well, I *am*) very busy, overwhelmed by deadlines. I don't feel like I have the time to stop and write down the few rules that I might actually follow more of less consistently. And it seems that a part of me is somewhat resistant to being accountable to a standard. Sound familiar?

Sigh...unfortunately (no, make that *very* fortunately), lots of other PL/SQL developers look to me for advice about how to "do things right." Ah, the pressure, the pressure!

Ah, the hypocrisy. That's what it really comes down to: I am a big hypocrite. I routinely violate just about every best practice I push on others. I sound high and mighty, but when I write code, I feel the same pull to do the quick and dirty.

Yet, I *do* have standards, and I even follow them, more or less. It's just been hard to find the time to write them down. So that's (finally) what you'll find below. It's a combination of the naming conventions I follow, certain naming conventions with which I *disagree*, and a variety of other suggestions. I hope you find it useful. I'm not going to make much of an effort to justify my approach or explain away the inconsistencies. I'll just show you what I do and you can decide if you'd like to use my path as a model for your own journey.

Principles Underlying My Standards

- The most important factor in a name is that it concisely (not that we have any choice about that with a limit of 30 characters!) 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? As my friend, John Beresniewicz, so neatly puts it: "The algorithm or process should be visible in procedure names and invisible in function names." If I had to choose between highly standardized, but obscure names or non-standardized, but highly descriptive names, I would always go for the latter.

- Normalize thy names: I try to 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. Lots of people disagree with me on this, including Bryn Llewellyn, PL/SQL Product Manager. He feels like my code is shouting at him. Sorry, Bryn! I like this style because it gives some *dimension* to the code, makes it easier to pick out *my* stuff as opposed to the built-in Oracle stuff. 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.

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);

General Guidelines I Try to Follow

- I 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.

- I 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.

- I 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!

- I try to 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 (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.

Naming Conventions I Reject

Some fairly 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.

How to Verify Naming Conventions and Standards

What's the point of setting standards if you never check to see if they are followed?

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.

PL/Scope Examples

1. Enable gathering of PL/Scope information:

```
ALTER SESSION SET plscope_settings='IDENTIFIERS:ALL'
/
```

2. 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
```

3. 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-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')
                          AND usage = 'DECLARATION'
                          AND SUBSTR (LOWER (name), 1, 2) <> 'p_'
                     THEN
                        '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 */
                                                                AND
SUBSTR (LOWER (name), 1, 2) <> 'g_'
                           THEN
                              'Violated convention that global
variables should be called g_<name>'
                           WHEN line > global_section.end_line /* local
variable */
                                                                AND
SUBSTR (LOWER (name), 1, 2) <> 'l_'
                           THEN
```

```
                                'Violated convention that local variables
should be called l_<name>'
                           END
                     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
/
```

Things I still wonder about

I haven't sorted it all out yet. Here are the aspects about which I am not yet completely clear. Do you have any suggestions or things to add to the list?

- How to distinguish between all the levels of scope between global (package level) and local (in the current block). What if the current block is a local subprogram within another subprogram?

- Should functions and procedures without any arguments be invoked with "()" after the subprogram name? Specifically for functions, if I do not invoke it this way *function_name*(), then it could be interpreted as a variable. Is this a bad thing? I am not sure. What do you think?

Modification History

Date	Change	By/From
June 16, 2009	Change prefix to suffix for record and add _rec as alternative.	Mike English
June 16, 2009	Fix typo on use of record type.	Layne Robinson
June 1, 2009	Add standards for exception and exception number.	Kevan Gelling
May 27, 2009	Publish on ToadWorld	Steven Feuerstein
Mid-May 2009	Incorporate recommendations from John Beresniewicz and Patrick Barel	John Beresniewicz Patrick Barel
May 17, 2009	Document distributed for review	Steven Feuerstein