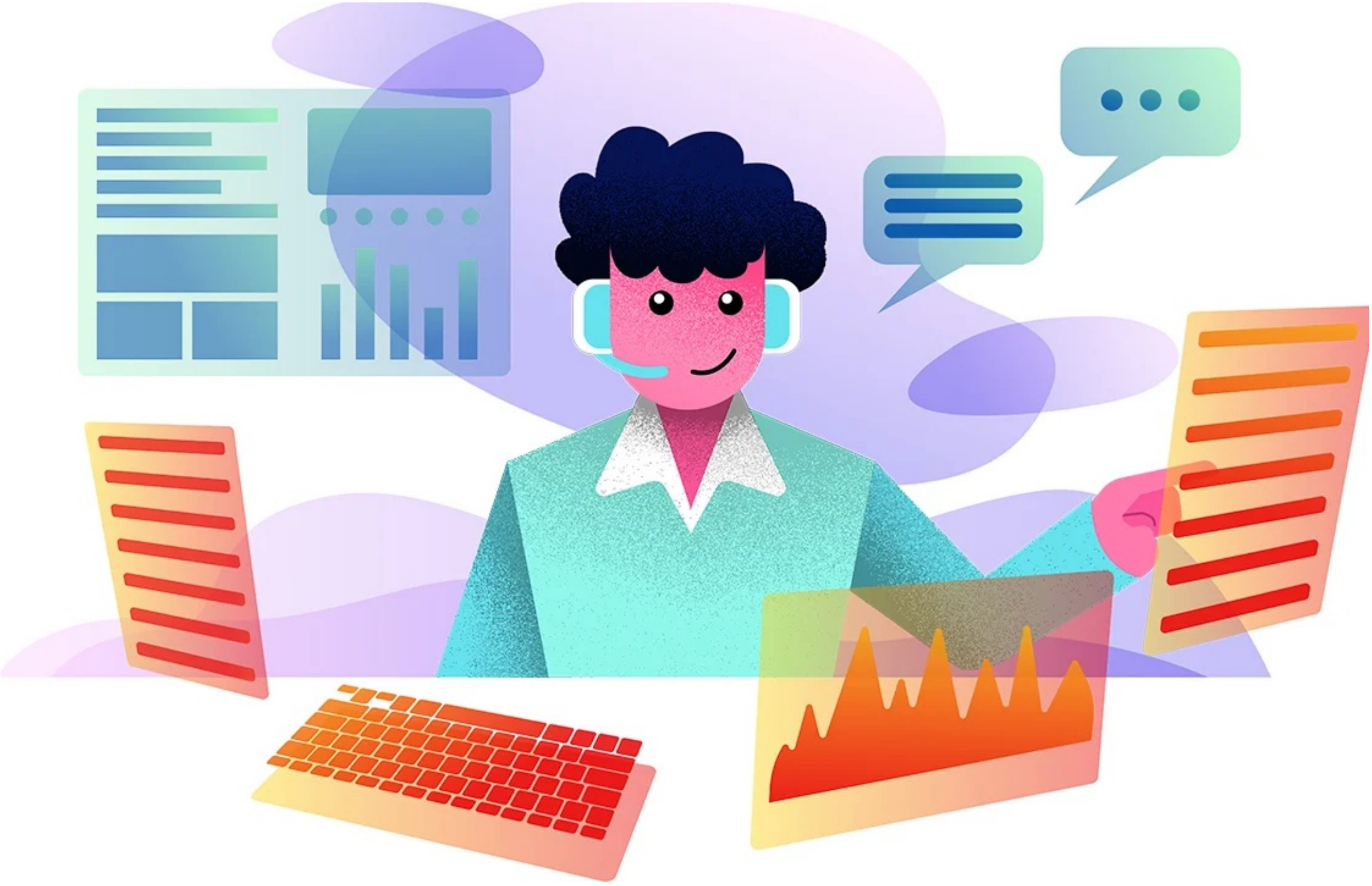
PL/SQL & SQL Coding Guidelines



Tips for Development & Operation

Document Version 4.0-SNAPSHOT

trivadis

Table of Contents

About

Foreword

In the I.T. world of today, robust and secure applications are becoming more and more important. Many business processes no longer work without I.T. and the dependence of businesses on their I.T. has grown tremendously, meaning we need robust and maintainable applications. An important requirement is to have standards and guidelines, which make it possible to maintain source code created by a number of people quickly and easily. This forms the basis of well functioning off- or on-shoring strategy, as it allows quality assurance to be carried out efficiently at the source.

Good standards and guidelines are based on the wealth of experience and knowledge gained from past (and future?) problems, such as those, which can arise in a cloud environment, for example.

Urban Lankes
Chairman of the Board of Directors
Trivadis

The Oracle Database Developer community is made stronger by resources freely shared by experts around the world, such as the Trivadis Coding Guidelines. If you have not yet adopted standards for writing SQL and PL/SQL in your applications, this is a great place to start.

Steven Feuerstein

Steven Feuerstein
Team Lead, Oracle Developer Advocates
Oracle

Coding Guidelines are a crucial part of software development. It is a matter of fact, that code is more often read than written – therefore we should take efforts to ease the work of the reader, which is not necessarily the author.

I am convinced that this standard may be a good starting point for your own guidelines.

Roger Troller
Senior Consultant
finnova AG Bankware

License

Trademarks

All terms that are known trademarks or service marks have been capitalized. All trademarks are the property of their respective owners.

Disclaimer

The authors and publisher shall have neither liability nor responsibility to any person or entity with respect to the loss or damages arising from the information contained in this work. This work may include inaccuracies or typographical errors and solely represent the opinions of the authors. Changes are periodically made to this document without notice. The authors reserve the right to revise this document at any time without notice.

Revision History

The first version of these guidelines was compiled by Roger Troller on March 17, 2009. Jörn Kulessa, Daniela Reiner, Richard Bushnell, Andreas Flubacher and Thomas Mauch helped Roger complete version 1.2 until August 21, 2009. This was the first GA version. The handy printed version in A5 format was distributed free of charge at the DOAG Annual Conference and on other occasions. Since then Roger updated the guidelines regularly. Philipp Salvisberg was involved in the review process for version 3.0 which was a major update. Philipp took the lead, after Roger left Trivadis in 2016. In 2020 Kim Berg Hansen started handling guidelines maintenance, letting Philipp concentrate on the related Trivadis PL/SQL Cop tool.

Since July, 7 2018 these guidelines are hosted on GitHub. Ready to be enhanced by the community and forked to fit specific needs.

On https://github.com/Trivadis/plsql-and-sql-coding-guidelines/releases you find the release information for every version since 1.2.

Introduction

This document describes rules and recommendations for developing applications using the PL/SQL & SQL Language.

Scope

This document applies to the PL/SQL and SQL language as used within ORACLE databases and tools, which access ORACLE databases.

Document Conventions

SQALE (Software Quality Assessment based on Lifecycle Expectations) is a method to support the evaluation of a software application source code. It is a generic method, independent of the language and source code analysis tools.

SQALE characteristics and subcharacteristics

Characteristic	Description and Subcharacteristics
Changeability	The capability of the software product to enable a specified modification to be implemented. • Architecture related changeability • Logic related changeability • Data related changeability
Efficiency	The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions. • Memory use • Processor use • Network use
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. • Understandability • Readability
Portability	The capability of the software product to be transferred from one environment to another. • Compiler related portability • Hardware related portability • Language related portability • OS related portability • Software related portability • Time zone related portability.
Reliability	The capability of the software product to maintain a specified level of performance when used under specified conditions. • Architecture related reliability • Data related reliability

	• Exception handling • Fault tolerance • Instruction related reliability • Logic related reliability • Resource related reliability • Synchronization related reliability • Unit tests coverage.
Reusability	The capability of the software product to be reused within the development process. • Modularity • Transportability.
Security	The capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them. • API abuse • Errors (e.g. leaving a system in a vulnerable state) • Input validatation and representation • Security features.
Testability	The capability of the software product to enable modified software to be validated. • Integration level testability • Unit level testability.

Severity of the rule

Blocker

Will or may result in a bug.

Critical

Will have a high/direct impact on the maintenance cost.

Major

Will have a medium/potential impact on the maintenance cost.

Minor

Will have a low impact on the maintenance cost.

Info

Very low impact; it is just a remediation cost report.

Keywords used

Keyword	Meaning
Always	Emphasizes this rule must be enforced.
Never	Emphasizes this action must not happen.
Avoid	Emphasizes that the action should be prevented, but some exceptions may exist.
Try	Emphasizes that the rule should be attempted whenever possible and appropriate.
Example	Precedes text used to illustrate a rule or a recommendation.
Reason	Explains the thoughts and purpose behind a rule or a recommendation.
Restriction	Describes the circumstances to be fulfilled to make use of a rule.

Why are standards important

For a machine executing a program, code formatting is of no importance. However, for the human eye, well-formatted code is much easier to read. Modern tools can help to implement format and coding rules.

Implementing formatting and coding standards has the following advantages for PL/SQL development:

- Well-formatted code is easier to read, analyze and maintain (not only for the author but also for other developers).
- The developers do not have to define their own guidelines - it is already defined.
- The code has a structure that makes it easier to avoid making errors.
- The code is more efficient concerning performance and organization of the whole application.
- The code is more modular and thus easier to use for other applications.

We have other standards

This document only defines possible standards. These standards are not written in stone, but are meant as guidelines. If standards already exist, and they are different from those in this document, it makes no sense to change them.

We do not agree with all your standards

There are basically two types of standards.

1. Non-controversial

   These standards make sense. There is no reason not to follow them. An example of this category is G-2150: Avoid comparisons with NULL value, consider using IS [NOT] NULL.

2. Controversial

   Almost every rule/guideline falls into this category. An example of this category is 3 space indention. - Why not 2 or 4 or even 8? Why not use tabs? You can argue in favor of all these options. In most cases it does not really matter which option you choose. Being consistent is more important. In this case it will make the code easier to read.

For very controversial rules, we have started to include the reasoning either as a footnote or directly in the text.

Usually it is not helpful to open an issue on GitHub to request to change a highly controversial rule such as the one mentioned. For example, use 2 spaces instead of 3 spaces for an indentation. This leads to a discussion where the people in favor of 4 spaces start to argument as well. There is no right or wrong here. You just have to agree on a standard.

More effective is to fork this repository and amend the standards to fit your needs/expectations.

Naming Conventions

General Guidelines

1. Never use names with a leading numeric character.
2. Always choose meaningful and specific names.
3. Avoid using abbreviations unless the full name is excessively long.
4. Avoid long abbreviations. Abbreviations should be shorter than 5 characters.
5. Any abbreviations must be widely known and accepted.
6. Create a glossary with all accepted abbreviations.
7. Never use ORACLE keywords as names. A list of ORACLEs keywords may be found in the dictionary view `v$reserved_words`.
8. Avoid adding redundant or meaningless prefixes and suffixes to identifiers.
   Example: `create table emp_table`.
9. Always use one spoken language (e.g. English, German, French) for all objects in your application.
10. Always use the same names for elements with the same meaning.

Naming Conventions for PL/SQL

In general, ORACLE is not case sensitive with names. A variable named personname is equal to one named PersonName, as well as to one named PERSONNAME. Some products (e.g. TMDA by Trivadis, APEX, OWB) put each name within double quotes (") so ORACLE will treat these names to be case sensitive. Using case sensitive variable names force developers to use double quotes for each reference to the variable. Our recommendation is to write all names in lowercase and to avoid double quoted identifiers.

A widely used convention is to follow a `{prefix}variablecontent{suffix}` pattern.

The following table shows a possible set of naming conventions.

Identifier	Prefix	Suffix	Example
Global Variable	`g_`		`g_version`
Local Variable	`l_`		`l_version`
Cursor	`c_`		`c_employees`
Record	`r_`		`r_employee`
Array / Table	`t_`		`t_employees`
Object	`o_`		`o_employee`
Cursor Parameter	`p_`		`p_empno`
In Parameter	`in_`		`in_empno`
Out Parameter	`out_`		`out_ename`
In/Out Parameter	`io_`		`io_employee`
Record Type Definitions	`r_`	`_type`	`r_employee_type`
Array/Table Type Definitions	`t_`	`_type`	`t_employees_type`
Exception	`e_`		`e_employee_exists`
Constants	`co_`		`co_empno`
Subtypes		`_type`	`big_string_type`

Database Object Naming Conventions

Never enclose object names (table names, column names, etc.) in double quotes to enforce mixed case or lower case object names in the data dictionary.

Collection Type

A collection type should include the name of the collected objects in their name. Furthermore, they should have the suffix `_ct` to identify it as a collection.

Optionally prefixed by a project abbreviation.

Examples:

- `employees_ct`
- `orders_ct`

Column

Singular name of what is stored in the column (unless the column data type is a collection, in this case you use plural names)[1]

Add a comment to the database dictionary for every column.

Check Constraint

Table name or table abbreviation followed by the column and/or role of the check constraint, a `_ck` and an optional number suffix.

Examples:

- `employees_salary_min_ck`
- `orders_mode_ck`

DML / Instead of Trigger

Choose a naming convention that includes:

either

- the name of the object the trigger is added to,
- any of the triggering events:
  - `_br_iud` for Before Row on Insert, Update and Delete
  - `_io_id` for Instead of Insert and Delete

or

- the name of the object the trigger is added to,
- the activity done by the trigger,
- the suffix `_trg`

Examples:

- `employees_br_iud`
- `orders_audit_trg`
- `orders_journal_trg`

Foreign Key Constraint

Table abbreviation followed by referenced table abbreviation followed by a `_fk` and an optional number suffix.

Examples:

- `empl_dept_fk`
- `sct_icmd_ic_fk1`

Function

Name is built from a verb followed by a noun in general. Nevertheless, it is not sensible to call a function `get_...` as a function always gets something.

The name of the function should answer the question “What is the outcome of the function?”

Optionally prefixed by a project abbreviation.

Example: `employee_by_id`

If more than one function provides the same outcome, you have to be more specific with the name.

Index

Indexes serving a constraint (primary, unique or foreign key) are named accordingly.

Other indexes should have the name of the table and columns (or their purpose) in their name and should also have `_idx` as a suffix.

Object Type

The name of an object type is built by its content (singular) followed by a `_ot` suffix.

Optionally prefixed by a project abbreviation.

Example: `employee_ot`

Package

Name is built from the content that is contained within the package.

Optionally prefixed by a project abbreviation.

Examples:

- `employees_api` - API for the employee table
- `logging_up` - Utilities including logging support

Primary Key Constraint

Table name or table abbreviation followed by the suffix `_pk` .

Examples:

- `employees_pk`
- `departments_pk`
- `sct_contracts_pk`

Procedure

Name is built from a verb followed by a noun. The name of the procedure should answer the question “What is done?”

Procedures and functions are often named with underscores between words because some editors write all letters in uppercase in the object tree, so it is difficult to read them.

Optionally prefixed by a project abbreviation.

Examples:

- `calculate_salary`
- `set_hiredate`
- `check_order_state`

Sequence

Name is built from the table name (or its abbreviation) the sequence serves as primary key generator and the suffix `_seq` or the purpose of the sequence followed by a `_seq` .

Optionally prefixed by a project abbreviation.

Examples:

- `employees_seq`
- `order_number_seq`

Synonym

Synonyms should be used to address an object in a foreign schema rather than to rename an object. Therefore, synonyms should share the name with the referenced object.

System Trigger

Name of the event the trigger is based on.

- Activity done by the trigger
- Suffix `_trg`

Examples:

- `ddl_audit_trg`
- `logon_trg`

Table

Plural[1] name of what is contained in the table (unless the table is designed to always hold one row only – then you should use a singular name).

Suffixed by `_eb` when protected by an editioning view.

Add a comment to the database dictionary for every table and every column in the table.

Optionally prefixed by a project abbreviation.

Examples:

- `employees`
- `departments`
- `countries_eb` - table interfaced by an editioning view named `countries`
- `sct_contracts`
- `sct_contract_lines`
- `sct_incentive_modules`

Temporary Table (Global Temporary Table)

Naming as described for tables.

Optionally suffixed by `_tmp`

Optionally prefixed by a project abbreviation.

Examples:

- `employees_tmp`
- `contracts_tmp`

Unique Key Constraint

Table name or table abbreviation followed by the role of the unique key constraint, a `_uk` and an optional number suffix.

Examples:

- `employees_name_uk`
- `departments_deptno_uk`
- `sct_contracts_uk`
- `sct_coli_uk`
- `sct_icmd_uk1`

View

Plural[1] name of what is contained in the view. Optionally suffixed by an indicator identifying the object as a view (mostly used, when a 1:1 view layer lies above the table layer)

Editioning views are named like the original underlying table to avoid changing the existing application code when introducing edition based redefinition (EBR).

Add a comment to the database dictionary for every view and every column.

Optionally prefixed by a project abbreviation.

Examples:

- `active_orders`
- `orders_v` - a view to the orders table
- `countries` - an editioning view for table `countries_eb`

Coding Style

Formatting

Rules

Rule	Description
1	Keywords and names are written in lowercase[2].
2	3 space indention[3].
3	One command per line.
4	Keywords `loop`, `else`, `elsif`, `end if`, `when` on a new line.
5	Commas in front of separated elements.
6	Call parameters aligned, operators aligned, values aligned.
7	SQL keywords are right aligned within a SQL command.
8	Within a program unit only line comments `--` are used.
9	Brackets are used when needed or when helpful to clarify a construct.

Example

```
procedure set_salary(in_employee_id in employees.employee_id%type) is
   cursor c_employees(p_employee_id in employees.employee_id%type) is
      select last_name
            ,first_name
            ,salary
        from employees
       where employee_id = p_employee_id
       order by last_name
               ,first_name;

   r_employee     c_employees%rowtype;
   l_new_salary   employees.salary%type;
begin
   open  c_employees(p_employee_id => in_employee_id);
   fetch c_employees into r_employee;
   close c_employees;

   new_salary (in_employee_id => in_employee_id
              ,out_salary     => l_new_salary);

   -- Check whether salary has changed
   if r_employee.salary <> l_new_salary then
      update employees
         set salary = l_new_salary
       where employee_id = in_employee_id;
   end if;
end set_salary;
```

Code Commenting

Conventions

Inside a program unit only use the line commenting technique `--` unless you temporarly deactivate code sections for testing.

To comment the source code for later document generation, comments like `/** ... */` are used. Within these documentation comments, tags may be used to define the documentation structure.

Tools like ORACLE SQL Developer or PL/SQL Developer include documentation functionality based on a javadoc-like tagging.

Commenting Tags

Tag	Meaning	Example
`param`	Description of a parameter.	`@param in_string input string`
`return`	Description of the return value of a function.	`@return result of the calculation`
`throws`	Describe errors that may be raised by the program unit.	`@throws NO_DATA_FOUND`

Example

This is an example using the documentation capabilities of SQL Developer.

```
/**
Check whether we passed a valid sql name

@param   in_name  string to be checked
@return  in_name if the string represents a valid sql name
@throws  ORA-44003: invalid SQL name

<b>Call Example:</b>
<pre>
   select TVDAssert.valid_sql_name('TEST') from dual;
   select TVDAssert.valid_sql_name('123') from dual
</pre>
*/
```

Language Usage

General

G-1010: Try to label your sub blocks.

Minor

Maintainability

Reason

It's a good alternative for comments to indicate the start and end of a named processing.

Example (bad)

```
begin
   begin
      null;
   end;

   begin
      null;
   end;
end;
/
```

Example (good)

```
begin
   <<prepare_data>>
   begin
      null;
   end prepare_data;

   <<process_data>>
   begin
      null;
   end process_data;
end good;
/
```

G-1020: Always have a matching loop or block label.

Minor

Maintainability

Reason

Use a label directly in front of loops and nested anonymous blocks:

- To give a name to that portion of code and thereby self-document what it is doing.
- So that you can repeat that name with the `end` statement of that block or loop.

Example (bad)

```
declare
   i integer;
   co_min_value constant integer := 1;
   co_max_value constant integer := 10;
   co_increment constant integer := 1;
begin
   <<prepare_data>>
   begin
      null;
   end;

   <<process_data>>
   begin
      null;
   end;

   i := co_min_value;
   <<while_loop>>
   while (i <= co_max_value)
   loop
      i := i + co_increment;
   end loop;

   <<basic_loop>>
   loop
      exit basic_loop;
   end loop;

   <<for_loop>>
   for i in co_min_value..co_max_value
   loop
      sys.dbms_output.put_line(i);
   end loop;
end;
/
```

Example (good)

```
declare
   i integer;
   co_min_value constant integer := 1;
   co_max_value constant integer := 10;
   co_increment constant integer := 1;
begin
   <<prepare_data>>
   begin
      null;
   end prepare_data;

   <<process_data>>
   begin
      null;
   end process_data;

   i := co_min_value;
   <<while_loop>>
   while (i <= co_max_value)
   loop
      i := i + co_increment;
   end loop while_loop;

   <<basic_loop>>
   loop
      exit basic_loop;
   end loop basic_loop;

   <<for_loop>>
   for i in co_min_value..co_max_value
   loop
     sys.dbms_output.put_line(i);
   end loop for_loop;
end;
/
```

G-1030: Avoid defining variables that are not used.

Minor

Efficiency, Maintainability

Reason

Unused variables decrease the maintainability and readability of your code.

Example (bad)

```
create or replace package body my_package is
   procedure my_proc is
      l_last_name  employees.last_name%type;
      l_first_name employees.first_name%type;
      co_department_id constant departments.department_id%type := 10;
      e_good exception;
   begin
      select e.last_name
        into l_last_name
        from employees e
       where e.department_id = co_department_id;
   exception
      when no_data_found then null; -- handle_no_data_found;
      when too_many_rows then null; -- handle_too_many_rows;
   end my_proc;
end my_package;
/
```

Example (good)

```
create or replace package body my_package is
   procedure my_proc is
      l_last_name  employees.last_name%type;
      co_department_id constant departments.department_id%type := 10;
      e_good exception;
   begin
      select e.last_name
        into l_last_name
        from employees e
       where e.department_id = co_department_id;

      raise e_good;
   exception
      when no_data_found then null; -- handle_no_data_found;
      when too_many_rows then null; -- handle_too_many_rows;
   end my_proc;
end my_package;
/
```

G-1040: Avoid dead code.

Minor

Maintainability

Reason

Any part of your code, which is no longer used or cannot be reached, should be eliminated from your programs to simplify the code.

Example (bad)

```
declare
   co_dept_purchasing constant departments.department_id%type := 30;
begin
   if 2=3 then
      null; -- some dead code here
   end if;

   null; -- some enabled code here

   <<my_loop>>
   loop
      exit my_loop;
      null; -- some dead code here
   end loop my_loop;

   null; -- some other enabled code here

   case
      when 1 = 1 and 'x' = 'y' then
         null; -- some dead code here
      else
         null; -- some further enabled code here
   end case;

   <<my_loop2>>
   for r_emp in (select last_name
                   from employees
                  where department_id = co_dept_purchasing
                     or commission_pct is not null
                    and 5=6)
                -- "or commission_pct is not null" is dead code
   loop
      sys.dbms_output.put_line(r_emp.last_name);
   end loop my_loop2;

   return;
   null; -- some dead code here
end;
/
```

Example (good)

```
declare
   co_dept_admin constant dept.deptno%type := 10;
begin
   null; -- some enabled code here
   null; -- some other enabled code here
   null; -- some further enabled code here

   <<my_loop2>>
   for r_emp in (select last_name
                   from employees
                  where department_id = co_dept_admin
                     or commission_pct is not null)
   loop
      sys.dbms_output.put_line(r_emp.last_name);
   end loop my_loop2;
end;
/
```

G-1050: Avoid using literals in your code.

Minor

Changeability

Reason

Literals are often used more than once in your code. Having them defined as a constant reduces typos in your code and improves the maintainability.

All constants should be collated in just one package used as a library. If these constants should be used in SQL too it is good practice to write a deterministic package function for every constant.

In specific situations this rule could lead to an extreme plethora of constants, for example if you use Logger like `logger.append_param(p_params =>l_params, p_name => 'p_param1_todo', p_val => p_param1_todo);` , where the value for `p_name` always should be the name of the variable that is passed to `p_val` . For such cases it would be overkill to add constants for every single variable name you are logging, so if you use Logger or similar, consider making that an exception to the rule, just document exactly which exceptions you will allow and stick to them.

Example (bad)

```
declare
   l_job employees.job_id%type;
begin
   select e.job_id
     into l_job
     from employees e
    where e.manager_id is null;

   if l_job = 'AD_PRES' then
      null;
   end if;
exception
   when no_data_found then
      null; -- handle_no_data_found;
   when too_many_rows then
      null; -- handle_too_many_rows;
end;
/
```

Example (good)

```
create or replace package constants_up is
   co_president constant employees.job_id%type := 'AD_PRES';
end constants_up;
/

declare
   l_job employees.job_id%type;
begin
   select e.job_id
     into l_job
     from employees e
    where e.manager_id is null;

   if l_job = constants_up.co_president then
      null;
   end if;
exception
   when no_data_found then
      null; -- handle_no_data_found;
   when too_many_rows then
      null; -- handle_too_many_rows;
end;
/
```

G-1060: Avoid storing ROWIDs or UROWIDs in database tables.

⚠ **Major**

Reliability

Reason

It is an extremely dangerous practice to store `rowid`'s in a table, except for some very limited scenarios of runtime duration. Any manually explicit or system generated implicit table reorganization will reassign the row's `rowid` and break the data consistency.

Instead of using `rowid` for later reference to the original row one should use the primary key column(s).

Example (bad)

```
begin
   insert into employees_log (employee_id
                             ,last_name
                             ,first_name
                             ,rid)
   select employee_id
         ,last_name
         ,first_name
         ,rowid
     from employees;
end;
/
```

Example (good)

```
begin
   insert into employees_log (employee_id
                             ,last_name
                             ,first_name)
   select employee_id
         ,last_name
         ,first_name
     from employees;
end;
/
```

G-1070: Avoid nesting comment blocks.

Minor

Maintainability

Reason

Having an end-of-comment within a block comment will end that block-comment. This does not only influence your code but is also very hard to read.

Example (bad)

```
begin
   /* comment one -- nested comment two */
   null;
   -- comment three /* nested comment four */
   null;
end;
/
```

Example (good)

```
begin
   /* comment one, comment two */
   null;
   -- comment three, comment four
   null;
end;
/
```

Variables & Types

General

G-2110: Try to use anchored declarations for variables, constants and types.

⚠ **Major**

Maintainability, Reliability

REASON

Changing the size of the database column last_name in the employees table from `varchar2(20)` to `varchar2(30)` will result in an error within your code whenever a value larger than the hard coded size is read from the table. This can be avoided using anchored declarations.

EXAMPLE (BAD)

```
create or replace package body my_package is
   procedure my_proc is
      l_last_name  varchar2(20 char);
      co_first_row constant integer := 1;
   begin
      select e.last_name
        into l_last_name
        from employees e
       where rownum = co_first_row;
   exception
      when no_data_found then null; -- handle no_data_found
      when too_many_rows then null; -- handle too_many_rows (impossible)
   end my_proc;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package body my_package is
   procedure my_proc is
      l_last_name  employees.last_name%type;
      co_first_row constant integer := 1;
   begin
      select e.last_name
        into l_last_name
        from employees e
       where rownum = co_first_row;
   exception
      when no_data_found then null; -- handle no_data_found
      when too_many_rows then null; -- handle too_many_rows (impossible)
   end my_proc;
end my_package;
/
```

G-2120: Try to have a single location to define your types.

Minor

Changeability

REASON

Single point of change when changing the data type. No need to argue where to define types or where to look for existing definitions.

A single location could be either a type specification package or the database (database-defined types).

EXAMPLE (BAD)

```
create or replace package body my_package is
   procedure my_proc is
      subtype big_string_type is varchar2(1000 char);
      l_note big_string_type;
   begin
      l_note := some_function();
   end my_proc;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package types_up is
   subtype big_string_type is varchar2(1000 char);
end types_up;
/

create or replace package body my_package is
   procedure my_proc is
      l_note types_up.big_string_type;
   begin
      l_note := some_function();
   end my_proc;
end my_package;
/
```

G-2130: Try to use subtypes for constructs used often in your code.

Minor

Changeability

REASON

Single point of change when changing the data type.

Your code will be easier to read as the usage of a variable/constant may be derived from its definition.

EXAMPLES OF POSSIBLE SUBTYPE DEFINITIONS

Type	Usage
`ora_name_type`	Object corresponding to the ORACLE naming conventions (table, variable, column, package, etc.).
`max_vc2_type`	String variable with maximal VARCHAR2 size.
`array_index_type`	Best fitting data type for array navigation.
`id_type`	Data type used for all primary key (id) columns.

EXAMPLE (BAD)

```
create or replace package body my_package is
   procedure my_proc is
      l_note varchar2(1000 char);
   begin
      l_note := some_function();
   end my_proc;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package types_up is
   subtype big_string_type is varchar2(1000 char);
end types_up;
/

create or replace package body my_package is
   procedure my_proc is
      l_note types_up.big_string_type;
   begin
      l_note := some_function();
   end my_proc;
end my_package;
/
```

G-2135: Avoid assigning values to local variables that are not used by a subsequent statement.

⚠ **Major**

Efficiency, Maintainability, Testability

REASON

Expending resources calculating and assigning values to a local variable and never use the value subsequently is at best a waste, at worst indicative of a mistake that leads to a bug.

EXAMPLE (BAD)

```
create or replace package body my_package is
   procedure my_proc is
      co_employee_id constant employees.employee_id%type := 1042;
      l_last_name  employees.last_name%type;
      l_message    varchar2(100 char);
   begin
      select emp.last_name
        into l_last_name
        from employees emp
       where emp.employee_id = co_employee_id;

      l_message := 'Hello, ' || l_last_name;
   exception
      when no_data_found then null; -- handle_no_data_found;
      when too_many_rows then null; -- handle_too_many_rows;
   end my_proc;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package body my_package is
   procedure my_proc is
      co_employee_id constant employees.employee_id%type := 1042;
      l_last_name  employees.last_name%type;
      l_message    varchar2(100 char);
   begin
      select emp.last_name
        into l_last_name
        from employees emp
       where emp.employee_id = co_employee_id;

      l_message := 'Hello, ' || l_last_name;

      message_api.send_message(l_message);
   exception
      when no_data_found then null; -- handle_no_data_found;
      when too_many_rows then null; -- handle_too_many_rows;
   end my_proc;
end my_package;
/
```

G-2140: Never initialize variables with NULL.

Minor

Maintainability

REASON

Variables are initialized to `null` by default.

EXAMPLE (BAD)

```
declare
   l_note big_string_type := null;
begin
   sys.dbms_output.put_line(l_note);
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_note big_string_type;
begin
   sys.dbms_output.put_line(l_note);
end;
/
```

G-2150: Avoid comparisons with NULL value, consider using IS [NOT] NULL.

Blocker

Portability, Reliability

REASON

The `null` value can cause confusion both from the standpoint of code review and code execution. You must always use the `is null` or `is not null` syntax when you need to check if a value is or is not `null`.

EXAMPLE (BAD)

```
declare
   l_value integer;
begin
   if l_value = null then
      null;
   end if;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_value integer;
begin
   if l_value is null then
      null;
   end if;
end;
/
```

G-2160: Avoid initializing variables using functions in the declaration section.

Critical

Reliability

REASON

If your initialization fails, you will not be able to handle the error in your exceptions block.

EXAMPLE (BAD)

```
declare
   co_department_id constant integer := 100;
   l_department_name departments.department_name%type :=
      department_api.name_by_id(in_id => co_department_id);
begin
   sys.dbms_output.put_line(l_department_name);
end;
/
```

EXAMPLE (GOOD)

```
declare
   co_department_id  constant integer := 100;
   co_unkown_name    constant departments.department_name%type := 'unknown';
   l_department_name departments.department_name%type;
begin
   <<init>>
   begin
      l_department_name := department_api.name_by_id(in_id => co_department_id);
   exception
      when value_error then
         l_department_name := co_unkown_name;
   end init;

   sys.dbms_output.put_line(l_department_name);
end;
/
```

G-2170: Never overload variables.

Major

Reliability

REASON

The readability of your code will be higher when you do not overload variables.

EXAMPLE (BAD)

```
begin
   <<main>>
   declare
      co_main constant user_objects.object_name%type := 'test_main';
      co_sub constant user_objects.object_name%type := 'test_sub';
      co_sep constant user_objects.object_name%type := ' - ';
      l_variable user_objects.object_name%type := co_main;
   begin
      <<sub>>
      declare
         l_variable user_objects.object_name%type := co_sub;
      begin
         sys.dbms_output.put_line(l_variable || co_sep || main.l_variable);
      end sub;
   end main;
end;
/
```

EXAMPLE (GOOD)

```
begin
   <<main>>
   declare
      co_main constant user_objects.object_name%type := 'test_main';
      co_sub constant user_objects.object_name%type := 'test_sub';
      co_sep constant user_objects.object_name%type := ' - ';
      l_main_variable user_objects.object_name%type := co_main;
   begin
      <<sub>>
      declare
         l_sub_variable user_objects.object_name%type := co_sub;
      begin
         sys.dbms_output.put_line(l_sub_variable || co_sep || l_main_variable);
      end sub;
   end main;
end;
/
```

G-2180: Never use quoted identifiers.

⚠ **Major**

Maintainability

REASON

Quoted identifiers make your code hard to read and maintain.

EXAMPLE (BAD)

```
declare
   "sal+comm" integer;
   "my constant" constant integer := 1;
   "my exception" exception;
begin
   "sal+comm" := "my constant";
exception
   when "my exception" then
      null;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_sal_comm     integer;
   co_my_constant constant integer := 1;
   e_my_exception exception;
begin
   l_sal_comm := co_my_constant;
exception
   when e_my_exception then
      null;
end;
/
```

G-2185: Avoid using overly short names for explicitly or implicitly declared identifiers.

Minor

Maintainability

REASON

You should ensure that the name you have chosen well defines its purpose and usage. While you can save a few keystrokes typing very short names, the resulting code is obscure and hard for anyone besides the author to understand.

EXAMPLE (BAD)

```
declare
   i integer;
   c constant integer := 1;
   e exception;
begin
   i := c;
exception
   when e then
      null;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_sal_comm     integer;
   co_my_constant constant integer := 1;
   e_my_exception exception;
begin
   l_sal_comm := co_my_constant;
exception
   when e_my_exception then
      null;
end;
/
```

G-2190: Avoid using ROWID or UROWID.

⚠ **Major**

Portability, Reliability

REASON

Be careful about your use of Oracle-specific data types like `rowid` and `urowid`. They might offer a slight improvement in performance over other means of identifying a single row (primary key or unique index value), but that is by no means guaranteed.

Use of `rowid` or `urowid` means that your SQL statement will not be portable to other SQL databases. Many developers are also not familiar with these data types, which can make the code harder to maintain.

EXAMPLE (BAD)

```
declare
   l_department_name departments.department_name%type;
   l_rowid rowid;
begin
   update departments
      set department_name = l_department_name
    where rowid = l_rowid;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_department_name  departments.department_name%type;
   l_department_id    departments.department_id%type;
begin
   update departments
      set department_name = l_department_name
    where department_id = l_department_id;
end;
/
```

Numeric Data Types

G-2210: Avoid declaring NUMBER variables, constants or subtypes with no precision.

Minor

Efficiency

REASON

If you do not specify precision `number` is defaulted to 38 or the maximum supported by your system, whichever is less. You may well need all this precision, but if you know you do not, you should specify whatever matches your needs.

EXAMPLE (BAD)

```
create or replace package body constants_up is
   co_small_increase constant number := 0.1;

   function small_increase return number is
   begin
      return co_small_increase;
   end small_increase;
end constants_up;
/
```

EXAMPLE (GOOD)

```
create or replace package body constants_up is
   co_small_increase constant number(5,1) := 0.1;

   function small_increase return number is
   begin
      return co_small_increase;
   end small_increase;
end constants_up;
/
```

G-2220: Try to use PLS_INTEGER instead of NUMBER for arithmetic operations with integer values.

Minor

Efficiency

REASON

`pls_integer` having a length of -2,147,483,648 to 2,147,483,647, on a 32bit system.

There are many reasons to use `pls_integer` instead of `number`:

- `pls_integer` uses less memory
- `pls_integer` uses machine arithmetic, which is up to three times faster than library arithmetic, which is used by `number`.

EXAMPLE (BAD)

```
create or replace package body constants_up is
   co_big_increase constant number(5,0) := 1;

   function big_increase return number is
   begin
      return co_big_increase;
   end big_increase;
end constants_up;
/
```

EXAMPLE (GOOD)

```
create or replace package body constants_up is
   co_big_increase constant pls_integer := 1;

   function big_increase return pls_integer is
   begin
      return co_big_increase;
   end big_increase;
end constants_up;
/
```

G-2230: Try to use SIMPLE_INTEGER datatype when appropriate.

Minor

Efficiency

RESTRICTION

ORACLE 11g or later

REASON

`simple_integer` does no checks on numeric overflow, which results in better performance compared to the other numeric datatypes.

With ORACLE 11g, the new data type `simple_integer` has been introduced. It is a sub-type of `pls_integer` and covers the same range. The basic difference is that `simple_integer` is always `not null`. When the value of the declared variable is never going to be null then you can declare it as `simple_integer`. Another major difference is that you will never face a numeric overflow using `simple_integer` as this data type wraps around without giving any error. `simple_integer` data type gives major performance boost over `pls_integer` when code is compiled in `native` mode, because arithmetic operations on `simple_integer` type are performed directly at the hardware level.

EXAMPLE (BAD)

```
create or replace package body constants_up is
   co_big_increase constant number(5,0) := 1;

   function big_increase return number is
   begin
      return co_big_increase;
   end big_increase;
end constants_up;
/
```

EXAMPLE (GOOD)

```
create or replace package body constants_up is
   co_big_increase constant simple_integer := 1;

   function big_increase return simple_integer is
   begin
      return co_big_increase;
   end big_increase;
end constants_up;
/
```

Character Data Types

G-2310: Avoid using CHAR data type.

⚠ **Major**

Reliability

REASON

`char` is a fixed length data type, which should only be used when appropriate. `char` columns/variables are always filled to its specified lengths; this may lead to unwanted side effects and undesired results.

EXAMPLE (BAD)

```
create or replace package types_up
is
   subtype description_type is char(200);
end types_up;
/
```

EXAMPLE (GOOD)

```
create or replace package types_up
is
   subtype description_type is varchar2(200 char);
end types_up;
/
```

G-2320: Never use VARCHAR data type.

Major

Portability

REASON

Do not use the `varchar` data type. Use the `varchar2` data type instead. Although the `varchar` data type is currently synonymous with `varchar2`, the `varchar` data type is scheduled to be redefined as a separate data type used for variable-length character strings compared with different comparison semantics.

EXAMPLE (BAD)

```
create or replace package types_up is
   subtype description_type is varchar(200);
end types_up;
/
```

EXAMPLE (GOOD)

```
create or replace package types_up is
   subtype description_type is varchar2(200 char);
end types_up;
/
```

G-2330: Never use zero-length strings to substitute NULL.

Major

Portability

REASON

Today zero-length strings and `null` are currently handled identical by ORACLE. There is no guarantee that this will still be the case in future releases, therefore if you mean `null` use `null`.

EXAMPLE (BAD)

```
create or replace package body constants_up is
   co_null_string constant varchar2(1) := '';

   function null_string return varchar2 is
   begin
       return co_null_string;
   end null_string;
end constants_up;
/
```

EXAMPLE (GOOD)

```
create or replace package body constants_up is

   function empty_string return varchar2 is
   begin
       return null;
   end empty_string;
end constants_up;
/
```

G-2340: Always define your VARCHAR2 variables using CHAR SEMANTIC (if not defined anchored).

Minor

Reliability

REASON

Changes to the `nls_length_semantic` will only be picked up by your code after a recompilation.

In a multibyte environment a `varchar2(10)` definition may not necessarily hold 10 characters when multibyte characters are part of the value that should be stored, unless the definition was done using the `char` semantic.

EXAMPLE (BAD)

```
create or replace package types_up is
   subtype description_type is varchar2(200);
end types_up;
/
```

EXAMPLE (GOOD)

```
create or replace package types_up is
   subtype description_type is varchar2(200 char);
end types_up;
/
```

Boolean Data Types

G-2410: Try to use boolean data type for values with dual meaning.

Minor

Maintainability

REASON

The use of `true` and `false` clarifies that this is a boolean value and makes the code easier to read.

EXAMPLE (BAD)

```
declare
   co_newfile constant pls_integer := 1000;
   co_oldfile constant pls_integer := 500;
   l_bigger   pls_integer;
begin
   if co_newfile < co_oldfile then
      l_bigger := constants_up.co_numeric_true;
   else
      l_bigger := constants_up.co_numeric_false;
   end if;
end;
/
```

EXAMPLE (BETTER)

```
declare
   co_newfile constant pls_integer := 1000;
   co_oldfile constant pls_integer := 500;
   l_bigger  boolean;
begin
   if co_newfile < co_oldfile then
      l_bigger := true;
   else
      l_bigger := false;
   end if;
end;
/
```

EXAMPLE (GOOD)

```
declare
   co_newfile constant pls_integer := 1000;
   co_oldfile constant pls_integer := 500;
   l_bigger  boolean;
begin
   l_bigger := nvl(co_newfile < co_oldfile, false);
end;
/
```

Large Objects

G-2510: Avoid using the LONG and LONG RAW data types.

⚠ **Major**

Portability

REASON

`long` and `long raw` data types have been deprecated by ORACLE since version 8i - support might be discontinued in future ORACLE releases.

There are many constraints to `long` datatypes in comparison to the `lob` types.

EXAMPLE (BAD)

```
create or replace package example_package is
   g_long long;
   g_raw  long raw;

   procedure do_something;
end example_package;
/

create or replace package body example_package is
   procedure do_something is
   begin
      null;
   end do_something;
end example_package;
/
```

EXAMPLE (GOOD)

```
create or replace package example_package is
   procedure do_something;
end example_package;
/

create or replace package body example_package is
   g_long clob;
   g_raw  blob;

   procedure do_something is
   begin
      null;
   end do_something;
end example_package;
/
```

DML & SQL

General

G-3110: Always specify the target columns when coding an insert statement.

Major

Maintainability, Reliability

REASON

Data structures often change. Having the target columns in your insert statements will lead to change-resistant code.

EXAMPLE (BAD)

```
insert into departments
     values (departments_seq.nextval
            ,'Support'
            ,100
            ,10);
```

EXAMPLE (GOOD)

```
insert into departments (department_id
                        ,department_name
                        ,manager_id
                        ,location_id)
     values (departments_seq.nextval
            ,'Support'
            ,100
            ,10);
```

G-3120: Always use table aliases when your SQL statement involves more than one source.

Major

Maintainability

REASON

It is more human readable to use aliases instead of writing columns with no table information.

Especially when using subqueries the omission of table aliases may end in unexpected behavior and result.

EXAMPLE (BAD)

```sql
select last_name
      ,first_name
      ,department_name
  from      employees
       join departments using (department_id)
 where extract(month from hire_date) = extract(month from sysdate);
```

EXAMPLE (BETTER)

```sql
select e.last_name
      ,e.first_name
      ,d.department_name
  from      employees   e
       join departments d on (e.department_id = d.department_id)
 where extract(month from e.hire_date) = extract(month from sysdate);
```

EXAMPLE (GOOD)

Using meaningful aliases improves the readability of your code.

```sql
select emp.last_name
      ,emp.first_name
      ,dept.department_name
  from      employees   emp
       join departments dept on (emp.department_id = dept.department_id)
 where extract(month from emp.hire_date) = extract(month from sysdate);
```

EXAMPLE SUBQUERY (BAD)

If the `jobs` table has no `employee_id` column and `employees` has one this query will not raise an error but return all rows of the `employees` table as a subquery is allowed to access columns of all its parent tables - this construct is known as correlated subquery.

```sql
select last_name
      ,first_name
  from employees
 where employee_id in (select employee_id
                         from jobs
                        where job_title like '%Manager%');
```

EXAMPLE SUBQUERY (GOOD)

If the `jobs` table has no `employee_id` column this query will return an error due to the directive (given by adding the table alias to the column) to read the `employee_id` column from the `jobs` table.

```sql
select emp.last_name
      ,emp.first_name
  from employees emp
 where emp.employee_id in (select j.employee_id
                             from jobs j
                            where j.job_title like '%Manager%');
```

G-3130: Try to use ANSI SQL-92 join syntax.

Minor

Maintainability, Portability

REASON

ANSI SQL-92 join syntax supports the full outer join. A further advantage of the ANSI SQL-92 join syntax is the separation of the join condition from the query filters.

EXAMPLE (BAD)

```
select e.employee_id
      ,e.last_name
      ,e.first_name
      ,d.department_name
  from employees e
      ,departments d
 where e.department_id = d.department_id
   and extract(month from e.hire_date) = extract(month from sysdate);
```

EXAMPLE (GOOD)

```
select emp.employee_id
      ,emp.last_name
      ,emp.first_name
      ,dept.department_name
  from      employees   emp
       join departments dept on dept.department_id = emp.department_id
 where extract(month from emp.hire_date) = extract(month from sysdate);
```

G-3140: Try to use anchored records as targets for your cursors.

Major

Maintainability, Reliability

REASON

Using cursor-anchored records as targets for your cursors results enables the possibility of changing the structure of the cursor without regard to the target structure.

EXAMPLE (BAD)

```
declare
   cursor c_employees is
      select employee_id, first_name, last_name
        from employees;
   l_employee_id employees.employee_id%type;
   l_first_name  employees.first_name%type;
   l_last_name   employees.last_name%type;
begin
   open c_employees;
   fetch c_employees into l_employee_id, l_first_name, l_last_name;
   <<process_employees>>
   while c_employees%found
   loop
      -- do something with the data
      fetch c_employees into l_employee_id, l_first_name, l_last_name;
   end loop process_employees;
   close c_employees;
end;
/
```

EXAMPLE (GOOD)

```
declare
   cursor c_employees is
      select employee_id, first_name, last_name
        from employees;
   r_employee c_employees%rowtype;
begin
   open c_employees;
   fetch c_employees into r_employee;
   <<process_employees>>
   while c_employees%found
   loop
      -- do something with the data
      fetch c_employees into r_employee;
   end loop process_employees;
   close c_employees;
end;
/
```

G-3150: Try to use identity columns for surrogate keys.

Minor

Maintainability, Reliability

RESTRICTION

ORACLE 12c

REASON

An identity column is a surrogate key by design – there is no reason why we should not take advantage of this natural implementation when the keys are generated on database level. Using identity column (and therefore assigning sequences as default values on columns) has a huge performance advantage over a trigger solution.

EXAMPLE (BAD)

```
create table locations (
   location_id        number(10)        not null
  ,location_name      varchar2(60 char) not null
  ,city               varchar2(30 char) not null
  ,constraint locations_pk primary key (location_id)
  )
/

create sequence location_seq start with 1 cache 20
/

create or replace trigger location_br_i
   before insert on locations
   for each row
begin
   :new.location_id := location_seq.nextval;
end;
/
```

EXAMPLE (GOOD)

```
create table locations (
   location_id        number(10)  generated always as identity
  ,location_name      varchar2(60 char) not null
  ,city               varchar2(30 char) not null
  ,constraint locations_pk primary key (location_id))
/
```

`generated always as identity` ensures that the `location_id` is populated by a sequence. It is not possible to override the behavior in the application.

However, if you use a framework that produces an `insert` statement including the surrogate key column, and you cannot change this behavior, then you have to use the `generated by default on null as identity` option. This has the downside that the application may pass a value, which might lead to an immediate or delayed `ORA-00001: unique constraint violated` error.

G-3160: Avoid visible virtual columns.

⚠ **Major**

Maintainability, Reliability

RESTRICTION

ORACLE 12c

REASON

In contrast to visible columns, invisible columns are not part of a record defined using `%rowtype` construct. This is helpful as a virtual column may not be programmatically populated. If your virtual column is visible you have to manually define the record types used in API packages to be able to exclude them from being part of the record definition.

Invisible columns may be accessed by explicitly adding them to the column list in a `select` statement.

EXAMPLE (BAD)

```
alter table employees
   add total_salary generated always as (salary + nvl(commission_pct,0) * salary)
/

declare
   r_employee employees%rowtype;
   l_id employees.employee_id%type := 107;
begin
   r_employee := employee_api.employee_by_id(l_id);
   r_employee.salary := r_employee.salary * constants_up.small_increase();

   update employees
      set row = r_employee
    where employee_id = l_id;
end;
/

Error report -
ORA-54017: update operation disallowed on virtual columns
ORA-06512: at line 9
```

EXAMPLE (GOOD)

```
alter table employees
   add total_salary invisible generated always as
      (salary + nvl(commission_pct,0) * salary)
/

declare
   r_employee employees%rowtype;
   co_id constant employees.employee_id%type := 107;
begin
   r_employee := employee_api.employee_by_id(co_id);
   r_employee.salary := r_employee.salary * constants_up.small_increase();

   update employees
      set row = r_employee
    where employee_id = co_id;
end;
/
```

G-3170: Always use DEFAULT ON NULL declarations to assign default values to table columns if you refuse to store NULL values.

Major

Reliability

RESTRICTION

ORACLE 12c

REASON

Default values have been nullifiable until ORACLE 12c. Meaning any tool sending null as a value for a column having a default value bypassed the default value. Starting with ORACLE 12c default definitions may have an `on null` definition in addition, which will assign the default value in case of a `null` value too.

EXAMPLE (BAD)

```
create table null_test (
   test_case        number(2) not null
  ,column_defaulted varchar2(10 char) default 'Default')
/
insert into null_test(test_case, column_defaulted) values (1,'Value');
insert into null_test(test_case, column_defaulted) values (2,default);
insert into null_test(test_case, column_defaulted) values (3,null);

select * from null_test;

TEST_CASE  COLUMN_DEF
---------  -----------
        1  Value
        2  Default
        3
```

EXAMPLE (GOOD)

```
create table null_test (
   test_case        number(2) not null
  ,column_defaulted varchar2(10 char) default on null 'Default')
/
insert into null_test(test_case, column_defaulted) values (1,'Value');
insert into null_test(test_case, column_defaulted) values (2,default);
insert into null_test(test_case, column_defaulted) values (3,null);

select * from null_test;

 TEST_CASE COLUMN_DEF
---------- ----------
         1 Value
         2 Default
         3 Default
```

G-3180: Always specify column names instead of positional references in ORDER BY clauses.

⚠ **Major**

Changeability, Reliability

REASON

If you change your `select` list afterwards the `order by` will still work but order your rows differently, when not changing the positional number. Furthermore, it is not comfortable to the readers of the code, if they have to count the columns in the `select` list to know the way the result is ordered.

EXAMPLE (BAD)

```
select upper(first_name)
      ,last_name
      ,salary
      ,hire_date
  from employees
 order by 4,1,3;
```

EXAMPLE (GOOD)

```
select upper(first_name) as first_name
      ,last_name
      ,salary
      ,hire_date
  from employees
 order by hire_date
         ,first_name
         ,salary;
```

G-3190: Avoid using NATURAL JOIN.

⚠ **Major**

Changeability, Reliability

REASON

A `natural join` joins tables on equally named columns. This may comfortably fit on first sight, but adding logging columns to a table (`changed_by` , `changed_date`) will result in inappropriate join conditions.

EXAMPLE (BAD)

```
select department_name
      ,last_name
      ,first_name
  from employees natural join departments
 order by department_name
         ,last_name;
DEPARTMENT_NAME                LAST_NAME                 FIRST_NAME
------------------------------ ------------------------- --------------------
Accounting                     Gietz                     William
Executive                      De Haan                   Lex
...

alter table departments add modified_at date default on null sysdate;
alter table employees add modified_at date default on null sysdate;

select department_name
      ,last_name
      ,first_name
  from employees natural join departments
 order by department_name
         ,last_name;

No data found
```

EXAMPLE (GOOD)

```
select d.department_name
      ,e.last_name
      ,e.first_name
  from employees   e
  join departments d on (e.department_id = d.department_id)
 order by d.department_name
         ,e.last_name;

DEPARTMENT_NAME                LAST_NAME                 FIRST_NAME
------------------------------ ------------------------- --------------------
Accounting                     Gietz                     William
Executive                      De Haan                   Lex
...
```

G-3195: Always use wildcards in a LIKE clause.

Minor

Maintainability

REASON

Using `like` without at least one wildcard (`%` or `_`) is unclear to a maintainer whether a wildcard is forgotten or it is meant as equality test. A common antipattern is also to forget that an underscore is a wildcard, so using `like` instead of equal can return unwanted rows. If the `char` datatype is involved, there is also the danger of `like` not using blank padded comparison where equal will. Depending on use case, you should either remember at least one wildcard or use normal equality operator.

EXAMPLE (BAD)

```sql
select e.employee_id
      ,e.last_name
  from employees e
 where e.last_name like 'Smith';
```

EXAMPLE (GOOD)

```sql
select e.employee_id
      ,e.last_name
  from employees e
 where e.last_name like 'Smith%';
```

EXAMPLE (GOOD)

```sql
select e.employee_id
      ,e.last_name
  from employees e
 where e.last_name = 'Smith';
```

Bulk Operations

G-3210: Always use BULK OPERATIONS (BULK COLLECT, FORALL) whenever you have to execute a DML statement for more than 4 times.

Major

Efficiency

REASON

Context switches between PL/SQL and SQL are extremely costly. BULK Operations reduce the number of switches by passing an array to the SQL engine, which is used to execute the given statements repeatedly.

(Depending on the PLSQL_OPTIMIZE_LEVEL parameter a conversion to BULK COLLECT will be done by the PL/SQL compiler automatically.)

EXAMPLE (BAD)

```
declare
   t_employee_ids employee_api.t_employee_ids_type;
   co_increase constant employees.salary%type := 0.1;
   co_department_id constant departments.department_id%type := 10;
begin
   t_employee_ids := employee_api.employee_ids_by_department(
                        id_in => co_department_id
                     );
   <<process_employees>>
   for i in 1..t_employee_ids.count()
   loop
      update employees
         set salary = salary + (salary * co_increase)
       where employee_id = t_employee_ids(i);
   end loop process_employees;
end;
/
```

EXAMPLE (GOOD)

```
declare
   t_employee_ids   employee_api.t_employee_ids_type;
   co_increase      constant employees.salary%type := 0.1;
   co_department_id constant departments.department_id%type := 10;
begin
   t_employee_ids := employee_api.employee_ids_by_department(
                        id_in => co_department_id
                     );
   <<process_employees>>
   forall i in 1..t_employee_ids.count()
      update employees
         set salary = salary + (salary * co_increase)
       where employee_id = t_employee_ids(i);
end;
/
```

Transaction Control

G-3310: Never commit within a cursor loop.

Critical

Efficiency, Reliability

REASON

Doing frequent commits within a cursor loop (all types of loops over cursors, whether implicit cursor for loop or loop with explicit fetch from cursor or cursor variable) risks not being able to complete due to ORA-01555, gives bad performance, and risks that the work is left in an unknown half-finished state and cannot be restarted.

- If the work belongs together (an atomic transaction) the `commit` should be moved to after the loop. Or even better if the logic can be rewritten to a single DML statement on all relevant rows instead of a loop, committing after the single statement.
- If each loop iteration is a self-contained atomic transaction, consider instead to populate a collection of transactions to be done (taking restartability into account by collection population), loop over that collection (instead of looping over a cursor) and call a procedure (that contains the transaction logic and the `commit`) in the loop (see also G-3320).

EXAMPLE (BAD)

```
declare
   l_counter   integer := 0;
   l_discount  discount.percentage%type;
begin
   for r_order in (
      select o.order_id, o.customer_id
        from orders o
       where o.order_status = 'New'
   ) loop
      l_discount := sales_api.calculate_discount(p_customer_id => r_order.customer_id);

      update order_lines ol
         set ol.discount = l_discount
       where ol.order_id = r_order.order_id;

      l_counter := l_counter + 1;
      if l_counter = 100 then
         commit;
         l_counter := 0;
      end if;
   end loop;
   if l_counter > 0 then
      commit;
   end if;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_discount  discount.percentage%type;
begin
   for r_order in (
      select o.order_id, o.customer_id
        from orders o
       where o.order_status = 'New'
   ) loop
      l_discount := sales_api.calculate_discount(p_customer_id => r_order.customer_id);

      update order_lines ol
         set ol.discount = l_discount
       where ol.order_id = r_order.order_id;
   end loop;

   commit;
end;
/
```

G-3320: Try to move transactions within a non-cursor loop into procedures.

⚠ **Major**

Maintainability, Reusability, Testability

REASON

Commit inside a non-cursor loop (other loop types than loops over cursors - see also G-3310) is either a self-contained atomic transaction, or it is a chunk (with suitable restartability handling) of very large data manipulations. In either case encapsulating the transaction in a procedure is good modularity, enabling reuse and testing of a single call.

EXAMPLE (BAD)

```
begin
   for l_counter in 1..5 loop
      insert into headers (id, text) values (l_counter, 'Number '||l_counter);

      insert into lines (header_id, line_no, text)
      select l_counter, rownum, 'Line '||rownum
        from dual
      connect by level <= 3;

      commit;
   end loop;
end;
/
```

EXAMPLE (GOOD)

```
declare
   procedure create_rows (
      p_header_id    in    headers.id%type
   ) is
   begin
      insert into headers (id, text) values (p_header_id, 'Number '||p_header_id);

      insert into lines (header_id, line_no, text)
      select p_header_id, rownum, 'Line '||rownum
        from dual
      connect by level <= 3;

      commit;
   end;
begin
   for l_counter in 1..5 loop
      create_rows(l_counter);
   end loop;
end;
/
```

Control Structures

CURSOR

G-4110: Always use %NOTFOUND instead of NOT %FOUND to check whether a cursor returned data.

Minor

Maintainability

REASON

The readability of your code will be higher when you avoid negative sentences.

EXAMPLE (BAD)

```
declare
   cursor c_employees is
      select last_name
            ,first_name
        from employees
       where commission_pct is not null;

   r_employee  c_employees%rowtype;
begin
   open c_employees;

   <<read_employees>>
   loop
      fetch c_employees into r_employee;
      exit read_employees when not c_employees%found;
   end loop read_employees;

   close c_employees;
end;
/
```

EXAMPLE (GOOD)

```
declare
   cursor c_employees is
      select last_name
            ,first_name
        from employees
       where commission_pct is not null;

   r_employee  c_employees%rowtype;
begin
   open c_employees;

   <<read_employees>>
   loop
      fetch c_employees into r_employee;
      exit read_employees when c_employees%notfound;
   end loop read_employees;

   close c_employees;
end;
/
```

G-4120: Avoid using %NOTFOUND directly after the FETCH when working with BULK OPERATIONS and LIMIT clause.

Critical

Reliability

REASON

`%notfound` is set to `true` as soon as less than the number of rows defined by the `limit` clause has been read.

EXAMPLE (BAD)

The employees table holds 107 rows. The example below will only show 100 rows as the cursor attribute `notfound` is set to true as soon as the number of rows to be fetched defined by the `limit` clause is not fulfilled anymore.

```
declare
   cursor c_employees is
      select *
        from employees
       order by employee_id;

   type t_employees_type is table of c_employees%rowtype;
   t_employees t_employees_type;
   co_bulk_size constant simple_integer := 10;
begin
   open c_employees;

   <<process_employees>>
   loop
      fetch c_employees bulk collect into t_employees limit co_bulk_size;
      exit process_employees when c_employees%notfound;

      <<display_employees>>
      for i in 1..t_employees.count()
      loop
         sys.dbms_output.put_line(t_employees(i).last_name);
      end loop display_employees;
   end loop process_employees;

   close c_employees;
end;
/
```

EXAMPLE (BETTER)

This example will show all 107 rows but execute one fetch too much (12 instead of 11).

```
declare
   cursor c_employees is
      select *
        from employees
       order by employee_id;

   type t_employees_type is table of c_employees%rowtype;
   t_employees t_employees_type;
   co_bulk_size constant simple_integer := 10;
begin
   open c_employees;

   <<process_employees>>
   loop
      fetch c_employees bulk collect into t_employees limit co_bulk_size;
      exit process_employees when t_employees.count() = 0;
      <<display_employees>>
      for i in 1..t_employees.count()
      loop
         sys.dbms_output.put_line(t_employees(i).last_name);
      end loop display_employees;
   end loop process_employees;

   close c_employees;
end;
/
```

EXAMPLE (GOOD)

This example does the trick (11 fetches only to process all rows)

```
declare
   cursor c_employees is
      select *
        from employees
       order by employee_id;

   type t_employees_type is table of c_employees%rowtype;
   t_employees t_employees_type;
   co_bulk_size constant simple_integer := 10;
begin
   open c_employees;

   <<process_employees>>
   loop
      fetch c_employees bulk collect into t_employees limit co_bulk_size;
      <<display_employees>>
      for i in 1..t_employees.count()
      loop
         sys.dbms_output.put_line(t_employees(i).last_name);
      end loop display_employees;
      exit process_employees when t_employees.count() <> co_bulk_size;
   end loop process_employees;

   close c_employees;
end;
/
```

G-4130: Always close locally opened cursors.

⚠ **Major**

Efficiency, Reliability

REASON

Any cursors left open can consume additional memory space (i.e. SGA) within the database instance, potentially in both the shared and private SQL pools. Furthermore, failure to explicitly close cursors may also cause the owning session to exceed its maximum limit of open cursors (as specified by the `open_cursors` database initialization parameter), potentially resulting in the Oracle error of “ORA-01000: maximum open cursors exceeded”.

EXAMPLE (BAD)

```
create or replace package body employee_api as
   function department_salary (in_dept_id in departments.department_id%type)
      return number is
      cursor c_department_salary(p_dept_id in departments.department_id%type) is
         select sum(salary) as sum_salary
           from employees
          where department_id = p_dept_id;
      r_department_salary c_department_salary%rowtype;
   begin
      open c_department_salary(p_dept_id => in_dept_id);
      fetch c_department_salary into r_department_salary;

      return r_department_salary.sum_salary;
   end department_salary;
end employee_api;
/
```

EXAMPLE (GOOD)

```
create or replace package body employee_api as
   function department_salary (in_dept_id in departments.department_id%type)
      return number is
      cursor c_department_salary(p_dept_id in departments.department_id%type) is
         select sum(salary) as sum_salary
           from employees
          where department_id = p_dept_id;
      r_department_salary c_department_salary%rowtype;
   begin
      open c_department_salary(p_dept_id => in_dept_id);
      fetch c_department_salary into r_department_salary;
      close c_department_salary;
      return r_department_salary.sum_salary;
   end department_salary;
end employee_api;
/
```

G-4140: Avoid executing any statements between a SQL operation and the usage of an implicit cursor attribute.

⚠ **Major**

Reliability

REASON

Oracle provides a variety of cursor attributes (like `%found` and `%rowcount`) that can be used to obtain information about the status of a cursor, either implicit or explicit.

You should avoid inserting any statements between the cursor operation and the use of an attribute against that cursor. Interposing such a statement can affect the value returned by the attribute, thereby potentially corrupting the logic of your program.

In the following example, a procedure call is inserted between the `delete` statement and a check for the value of `sql%rowcount` , which returns the number of rows modified by that last SQL statement executed in the session. If this procedure includes a `commit` / `rollback` or another implicit cursor the value of `sql%rowcount` is affected.

EXAMPLE (BAD)

```
create or replace package body employee_api as
   co_one constant simple_integer := 1;

   procedure process_dept(in_dept_id in departments.department_id%type) is
   begin
      null;
   end process_dept;

   procedure remove_employee (in_employee_id in employees.employee_id%type) is
      l_dept_id      employees.department_id%type;
   begin
      delete from employees
       where employee_id = in_employee_id
       returning department_id into l_dept_id;

      process_dept(in_dept_id => l_dept_id);

      if sql%rowcount > co_one then
         -- too many rows deleted.
         rollback;
      end if;
   end remove_employee;
end employee_api;
/
```

EXAMPLE (GOOD)

```
create or replace package body employee_api as
   co_one constant simple_integer := 1;

   procedure process_dept(in_dept_id in departments.department_id%type) is
   begin
      null;
   end process_dept;

   procedure remove_employee (in_employee_id in employees.employee_id%type) is
      l_dept_id      employees.department_id%type;
      l_deleted_emps simple_integer;
   begin
      delete from employees
       where employee_id = in_employee_id
       returning department_id into l_dept_id;

      l_deleted_emps := sql%rowcount;

      process_dept(in_dept_id => l_dept_id);

      if l_deleted_emps > co_one then
         -- too many rows deleted.
         rollback;
      end if;
   end remove_employee;
end employee_api;
/
```

CASE / IF / DECODE / NVL / NVL2 / COALESCE

G-4210: Try to use CASE rather than an IF statement with multiple ELSIF paths.

⚠ **Major**

Maintainability, Testability

REASON

`if` statements containing multiple `elsif` tend to become complex quickly.

EXAMPLE (BAD)

```
declare
   l_color varchar2(7 char);
begin
   if l_color = constants_up.co_red then
      my_package.do_red();
   elsif l_color = constants_up.co_blue then
      my_package.do_blue();
   elsif l_color = constants_up.co_black then
      my_package.do_black();
   end if;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_color types_up.color_code_type;
begin
   case l_color
      when constants_up.co_red   then
         my_package.do_red();
      when constants_up.co_blue  then
         my_package.do_blue();
      when constants_up.co_black then
         my_package.do_black();
      else null;
   end case;
end;
/
```

G-4220: Try to use CASE rather than DECODE.

Minor

Maintainability, Portability

REASON

`decode` is an ORACLE specific function hard to understand and restricted to SQL only. The “newer” `case` function is much more common, has a better readability and may be used within PL/SQL too. Be careful that `decode` can handle `null` values, which the simple `case` cannot - for such cases you must use the searched `case` and `is null` instead.

EXAMPLE (BAD)

```sql
select decode(dummy, 'X', 1
                   , 'Y', 2
                   , 'Z', 3
                        , 0)
  from dual;
```

EXAMPLE (GOOD)

```sql
select case dummy
          when 'X' then 1
          when 'Y' then 2
          when 'Z' then 3
          else 0
       end
  from dual;
```

EXAMPLE (BAD)

```sql
select decode(dummy,  'X',  1
                   ,  'Y',  2
                   , null, -1
                         ,  0)
  from dual;
```

EXAMPLE (GOOD)

```sql
select case
          when dummy = 'X'    then  1
          when dummy = 'Y'    then  2
          when dummy is null then -1
          else 0
       end
  from dual;
```

G-4230: Always use a COALESCE instead of a NVL command, if parameter 2 of the NVL function is a function call or a SELECT statement.

Critical

Efficiency, Reliability

REASON

The nvl function always evaluates both parameters before deciding which one to use. This can be harmful if parameter 2 is either a function call or a select statement, as it will be executed regardless of whether parameter 1 contains a null value or not.

The coalesce function does not have this drawback.

EXAMPLE (BAD)

```
select nvl(dummy, my_package.expensive_null(value_in => dummy))
  from dual;
```

EXAMPLE (GOOD)

```
select coalesce(dummy, my_package.expensive_null(value_in => dummy))
  from dual;
```

G-4240: Always use a CASE instead of a NVL2 command if parameter 2 or 3 of NVL2 is either a function call or a SELECT statement.

Critical

Efficiency, Reliability

REASON

The `nvl2` function always evaluates all parameters before deciding which one to use. This can be harmful, if parameter 2 or 3 is either a function call or a select statement, as they will be executed regardless of whether parameter 1 contains a `null` value or not.

EXAMPLE (BAD)

```sql
select nvl2(dummy, my_package.expensive_nn(value_in => dummy),
                   my_package.expensive_null(value_in => dummy))
  from dual;
```

EXAMPLE (GOOD)

```sql
select case
          when dummy is null then
             my_package.expensive_null(value_in => dummy)
          else
             my_package.expensive_nn(value_in => dummy)
       end
from dual;
```

G-4250: Avoid using identical conditions in different branches of the same IF or CASE statement.

⚠ **Major**

Maintainability, Reliability, Testability

REASON

Conditions are evaluated top to bottom in branches of a `case` statement or chain of `if` / `elsif` statements. The first condition to evaluate as true leads to that branch being executed, the rest will never execute. Having an identical duplicated condition in another branch will never be reached and will be dead code.

EXAMPLE (BAD)

```
declare
   l_color types_up.color_code_type;
begin
   case l_color
      when constants_up.co_red   then
         my_package.do_red();
      when constants_up.co_blue  then
         my_package.do_blue();
      when constants_up.co_red then  -- never reached
         my_package.do_black();      -- dead code
      else null;
   end case;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_color types_up.color_code_type;
begin
   case l_color
      when constants_up.co_red   then
         my_package.do_red();
      when constants_up.co_blue  then
         my_package.do_blue();
      when constants_up.co_black then
         my_package.do_black();
      else null;
   end case;
end;
/
```

G-4260: Avoid inverting boolean conditions.

Minor

Maintainability, Testability

REASON

It is more readable to use the opposite comparison operator instead of inverting the comparison with `not`.

EXAMPLE (BAD)

```
declare
   l_color varchar2(7 char);
begin
   if not l_color != constants_up.co_red then
      my_package.do_red();
   end if;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_color types_up.color_code_type;
begin
   if l_color = constants_up.co_red then
      my_package.do_red();
   end if;
end;
/
```

G-4270: Avoid comparing boolean values to boolean literals.

Minor

Maintainability, Testability

REASON

It is more readable to simply use the boolean value as a condition itself, rather than use a comparison condition comparing the boolean value to the literals `true` or `false`.

EXAMPLE (BAD)

```
declare
   l_string    varchar2(10 char) := '42';
   l_is_valid  boolean;
begin
   l_is_valid := my_package.is_valid_number(l_string);
   if l_is_valid = true then
      my_package.convert_number(l_string);
   end if;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_string    varchar2(10 char) := '42';
   l_is_valid  boolean;
begin
   l_is_valid := my_package.is_valid_number(l_string);
   if l_is_valid then
      my_package.convert_number(l_string);
   end if;
end;
/
```

Flow Control

G-4310: Never use GOTO statements in your code.

⚠ **Major**

Maintainability, Testability

REASON

> Code containing gotos is hard to format. Indentation should be used to show logical structure, and gotos have an effect on logical structure. Using indentation to show the logical structure of a goto and its target, however, is difficult or impossible. (...)
>
> Use of gotos is a matter of religion. My dogma is that in modern languages, you can easily replace nine out of ten gotos with equivalent sequential constructs. In these simple cases, you should replace gotos out of habit. In the hard cases, you can still exorcise the goto in nine out of ten cases: You can break the code into smaller routines, use try-finally, use nested ifs, test and retest a status variable, or restructure a conditional. Eliminating the goto is harder in these cases, but it’s good mental exercise (...).
>
> -- McConnell, Steve C. (2004). *Code Complete. Second Edition*. Microsoft Press.

EXAMPLE (BAD)

```
create or replace package body my_package is
   procedure password_check (in_password in varchar2) is
      co_digitarray  constant string(10 char)   := '0123456789';
      co_lower_bound constant simple_integer := 1;
      co_errno       constant simple_integer := -20501;
      co_errmsg      constant string(100 char)  := 'Password must contain a digit.';
      l_isdigit      boolean     := false;
      l_len_pw       pls_integer;
      l_len_array    pls_integer;
   begin
      l_len_pw    := length(in_password);
      l_len_array := length(co_digitarray);

      <<check_digit>>
      for i in co_lower_bound .. l_len_array
      loop
         <<check_pw_char>>
         for j in co_lower_bound .. l_len_pw
         loop
            if substr(in_password, j, 1) = substr(co_digitarray, i, 1) then
               l_isdigit := true;
               goto check_other_things;
            end if;
         end loop check_pw_char;
      end loop check_digit;

      <<check_other_things>>
      null;

      if not l_isdigit then
         raise_application_error(co_errno, co_errmsg);
      end if;
   end password_check;
end my_package;
/
```

EXAMPLE (BETTER)

```
create or replace package body my_package is
   procedure password_check (in_password in varchar2) is
      co_digitarray  constant string(10 char)   := '0123456789';
      co_lower_bound constant simple_integer := 1;
      co_errno       constant simple_integer := -20501;
      co_errmsg      constant string(100 char)  := 'Password must contain a digit.';
      l_isdigit      boolean     := false;
      l_len_pw       pls_integer;
      l_len_array    pls_integer;
   begin
      l_len_pw    := length(in_password);
      l_len_array := length(co_digitarray);

      <<check_digit>>
      for i in co_lower_bound .. l_len_array
      loop
         <<check_pw_char>>
         for j in co_lower_bound .. l_len_pw
         loop
            if substr(in_password, j, 1) = substr(co_digitarray, i, 1) then
               l_isdigit := true;
               exit check_digit; -- early exit condition
            end if;
         end loop check_pw_char;
      end loop check_digit;

      <<check_other_things>>
      null;

      if not l_isdigit then
         raise_application_error(co_errno, co_errmsg);
      end if;
   end password_check;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package body my_package is
   procedure password_check (in_password in varchar2) is
      co_digitpattern constant string(10 char)   := '\d';
      co_errno        constant simple_integer := -20501;
      co_errmsg       constant string(100 char)  := 'Password must contain a digit.';
   begin
      if not regexp_like(in_password, co_digitpattern)
      then
         raise_application_error(co_errno, co_errmsg);
      end if;
   end password_check;
end my_package;
/
```

G-4320: Always label your loops.

Minor

Maintainability

REASON

It's a good alternative for comments to indicate the start and end of a named loop processing.

EXAMPLE (BAD)

```
declare
   i integer;
   co_min_value constant simple_integer := 1;
   co_max_value constant simple_integer := 10;
   co_increment constant simple_integer := 1;
begin
   i := co_min_value;
   while (i <= co_max_value)
   loop
      i := i + co_increment;
   end loop;

   loop
      exit;
   end loop;

   for i in co_min_value..co_max_value
   loop
      sys.dbms_output.put_line(i);
   end loop;

   for r_employee in (select last_name from employees)
   loop
      sys.dbms_output.put_line(r_employee.last_name);
   end loop;
end;
/
```

EXAMPLE (GOOD)

```
declare
   i integer;
   co_min_value constant simple_integer := 1;
   co_max_value constant simple_integer := 10;
   co_increment constant simple_integer := 1;
begin
   i := co_min_value;
   <<while_loop>>
   while (i <= co_max_value)
   loop
      i := i + co_increment;
   end loop while_loop;

   <<basic_loop>>
   loop
      exit basic_loop;
   end loop basic_loop;

   <<for_loop>>
   for i in co_min_value..co_max_value
   loop
      sys.dbms_output.put_line(i);
   end loop for_loop;

   <<process_employees>>
   for r_employee in (select last_name
                        from employees)
   loop
      sys.dbms_output.put_line(r_employee.last_name);
   end loop process_employees;
end;
/
```

G-4330: Always use a CURSOR FOR loop to process the complete cursor results unless you are using bulk operations.

Minor

Maintainability

REASON

It is easier for the reader to see, that the complete data set is processed. Using SQL to define the data to be processed is easier to maintain and typically faster than using conditional processing within the loop.

Since an `exit` statement is similar to a `goto` statement, it should be avoided, whenever possible.

EXAMPLE (BAD)

```
declare
   cursor c_employees is
      select employee_id, last_name
        from employees;
   r_employee c_employees%rowtype;
begin
   open c_employees;

   <<read_employees>>
   loop
      fetch c_employees into r_employee;
      exit read_employees when c_employees%notfound;
      sys.dbms_output.put_line(r_employee.last_name);
   end loop read_employees;

   close c_employees;
end;
/
```

EXAMPLE (GOOD)

```
declare
   cursor c_employees is
      select employee_id, last_name
        from employees;
begin
   <<read_employees>>
   for r_employee in c_employees
   loop
      sys.dbms_output.put_line(r_employee.last_name);
   end loop read_employees;
end;
/
```

G-4340: Always use a NUMERIC FOR loop to process a dense array.

Minor

Maintainability

REASON

It is easier for the reader to see, that the complete array is processed.

Since an `exit` statement is similar to a `goto` statement, it should be avoided, whenever possible.

EXAMPLE (BAD)

```
declare
   type t_employee_type is varray(10) of employees.employee_id%type;
   t_employees t_employee_type;
   co_himuro     constant integer := 118;
   co_livingston constant integer := 177;
   co_min_value  constant simple_integer := 1;
   co_increment  constant simple_integer := 1;
   i pls_integer;
begin
   t_employees := t_employee_type(co_himuro, co_livingston);
   i           := co_min_value;

   <<process_employees>>
   loop
      exit process_employees when i > t_employees.count();
      sys.dbms_output.put_line(t_employees(i));
      i := i + co_increment;
   end loop process_employees;
end;
/
```

EXAMPLE (GOOD)

```
declare
   type t_employee_type is varray(10) of employees.employee_id%type;
   t_employees t_employee_type;
   co_himuro     constant integer := 118;
   co_livingston constant integer := 177;
begin
   t_employees := t_employee_type(co_himuro, co_livingston);

   <<process_employees>>
   for i in 1..t_employees.count()
   loop
     sys.dbms_output.put_line(t_employees(i));
   end loop process_employees;
end;
/
```

G-4350: Always use 1 as lower and COUNT() as upper bound when looping through a dense array.

Major

Reliability

REASON

Doing so will not raise a `value_error` if the array you are looping through is empty. If you want to use `first()..last()` you need to check the array for emptiness beforehand to avoid the raise of `value_error`.

EXAMPLE (BAD)

```
declare
   type t_employee_type is table of employees.employee_id%type;
   t_employees t_employee_type := t_employee_type();
begin
   <<process_employees>>
   for i in t_employees.first()..t_employees.last()
   loop
      sys.dbms_output.put_line(t_employees(i)); -- some processing
   end loop process_employees;
end;
/
```

EXAMPLE (BETTER)

Raise an unitialized collection error if `t_employees` is not initialized.

```
declare
   type t_employee_type is table of employees.employee_id%type;
   t_employees t_employee_type := t_employee_type();
begin
   <<process_employees>>
   for i in 1..t_employees.count()
   loop
      sys.dbms_output.put_line(t_employees(i)); -- some processing
   end loop process_employees;
end;
/
```

EXAMPLE (GOOD)

Raises neither an error nor checking whether the array is empty. `t_employees.count()` always returns a `number` (unless the array is not initialized). If the array is empty `count()` returns 0 and therefore the loop will not be entered.

```
declare
   type t_employee_type is table of employees.employee_id%type;
   t_employees t_employee_type := t_employee_type();
begin
   if t_employees is not null then
      <<process_employees>>
      for i in 1..t_employees.count()
      loop
         sys.dbms_output.put_line(t_employees(i)); -- some processing
      end loop process_employees;
   end if;
end;
/
```

G-4360: Always use a WHILE loop to process a loose array.

Minor

Efficiency

REASON

When a loose (also called sparse) array is processed using a *numeric* `for loop` we have to check with all iterations whether the element exist to avoid a `no_data_found` exception. In addition, the number of iterations is not driven by the number of elements in the array but by the number of the lowest/highest element. The more gaps we have, the more superfluous iterations will be done.

EXAMPLE (BAD)

```
declare -- raises no_data_found when processing 2nd record
   type t_employee_type is table of employees.employee_id%type;
   t_employees t_employee_type;
   co_rogers      constant integer := 134;
   co_matos       constant integer := 143;
   co_mcewen      constant integer := 158;
   co_index_matos constant integer := 2;
begin
   t_employees := t_employee_type(co_rogers, co_matos, co_mcewen);
   t_employees.delete(co_index_matos);

   if t_employees is not null then
      <<process_employees>>
      for i in 1..t_employees.count()
      loop
         sys.dbms_output.put_line(t_employees(i));
      end loop process_employees;
   end if;
end;
/
```

EXAMPLE (GOOD)

```
declare
   type t_employee_type is table of employees.employee_id%type;
   t_employees t_employee_type;
   co_rogers      constant integer := 134;
   co_matos       constant integer := 143;
   co_mcewen      constant integer := 158;
   co_index_matos constant integer := 2;
   l_index        pls_integer;
begin
   t_employees := t_employee_type(co_rogers, co_matos, co_mcewen);
   t_employees.delete(co_index_matos);

   l_index := t_employees.first();

   <<process_employees>>
   while l_index is not null
   loop
      sys.dbms_output.put_line(t_employees(l_index));
      l_index := t_employees.next(l_index);
   end loop process_employees;
end;
/
```

G-4370: Avoid using EXIT to stop loop processing unless you are in a basic loop.

⚠ **Major**

Maintainability

REASON

A numeric for loop as well as a while loop and a cursor for loop have defined loop boundaries. If you are not able to exit your loop using those loop boundaries, then a basic loop is the right loop to choose.

EXAMPLE (BAD)

```
declare
   i integer;
   co_min_value constant simple_integer := 1;
   co_max_value constant simple_integer := 10;
   co_increment constant simple_integer := 1;
begin
   i := co_min_value;
   <<while_loop>>
   while (i <= co_max_value)
   loop
      i := i + co_increment;
      exit while_loop when i > co_max_value;
   end loop while_loop;

   <<basic_loop>>
   loop
      exit basic_loop;
   end loop basic_loop;

   <<for_loop>>
   for i in co_min_value..co_max_value
   loop
      null;
      exit for_loop when i = co_max_value;
   end loop for_loop;

   <<process_employees>>
   for r_employee in (select last_name
                        from employees)
   loop
      sys.dbms_output.put_line(r_employee.last_name);
      null; -- some processing
      exit process_employees;
   end loop process_employees;
end;
/
```

EXAMPLE (GOOD)

```
declare
   i integer;
   co_min_value constant simple_integer := 1;
   co_max_value constant simple_integer := 10;
   co_increment constant simple_integer := 1;
begin
   i := co_min_value;
   <<while_loop>>
   while (i <= co_max_value)
   loop
      i := i + co_increment;
   end loop while_loop;

   <<basic_loop>>
   loop
      exit basic_loop;
   end loop basic_loop;

   <<for_loop>>
   for i in co_min_value..co_max_value
   loop
      sys.dbms_output.put_line(i);
   end loop for_loop;

   <<process_employees>>
   for r_employee in (select last_name
                        from employees)
   loop
      sys.dbms_output.put_line(r_employee.last_name); -- some processing
   end loop process_employees;
end;
/
```

G-4375: Always use EXIT WHEN instead of an IF statement to exit from a loop.

Minor

Maintainability

REASON

If you need to use an `exit` statement use its full semantic to make the code easier to understand and maintain. There is simply no need for an additional `if` statement.

EXAMPLE (BAD)

```
declare
   co_first_year constant pls_integer := 1900;
begin
   <<process_employees>>
   loop
      my_package.some_processing();

      if extract(year from sysdate) > co_first_year then
         exit process_employees;
      end if;

      my_package.some_further_processing();
   end loop process_employees;
end;
/
```

EXAMPLE (GOOD)

```
declare
   co_first_year constant pls_integer := 1900;
begin
   <<process_employees>>
   loop
      my_package.some_processing();

      exit process_employees when extract(year from sysdate) > co_first_year;

      my_package.some_further_processing();
   end loop process_employees;
end;
/
```

G-4380 Try to label your EXIT WHEN statements.

Minor

Maintainability

REASON

It's a good alternative for comments, especially for nested loops to name the loop to exit.

EXAMPLE (BAD)

```
declare
   co_init_loop  constant simple_integer           := 0;
   co_increment  constant simple_integer           := 1;
   co_exit_value constant simple_integer           := 3;
   co_outer_text constant types_up.short_text_type := 'Outer Loop counter is ';
   co_inner_text constant types_up.short_text_type := ' Inner Loop counter is ';
   l_outerlp pls_integer;
   l_innerlp pls_integer;
begin
   l_outerlp := co_init_loop;
   <<outerloop>>
   loop
      l_innerlp := co_init_loop;
      l_outerlp := nvl(l_outerlp,co_init_loop) + co_increment;
      <<innerloop>>
      loop
         l_innerlp := nvl(l_innerlp, co_init_loop) + co_increment;
         sys.dbms_output.put_line(co_outer_text || l_outerlp ||
                                  co_inner_text || l_innerlp);

         exit when l_innerlp = co_exit_value;
      end loop innerloop;

      exit when l_innerlp = co_exit_value;
   end loop outerloop;
end;
/
```

EXAMPLE (GOOD)

```
declare
   co_init_loop  constant simple_integer           := 0;
   co_increment  constant simple_integer           := 1;
   co_exit_value constant simple_integer           := 3;
   co_outer_text constant types_up.short_text_type := 'Outer Loop counter is ';
   co_inner_text constant types_up.short_text_type := ' Inner Loop counter is ';
   l_outerlp pls_integer;
   l_innerlp pls_integer;
begin
   l_outerlp := co_init_loop;
   <<outerloop>>
   loop
      l_innerlp := co_init_loop;
      l_outerlp := nvl(l_outerlp,co_init_loop) + co_increment;
      <<innerloop>>
      loop
         l_innerlp := nvl(l_innerlp, co_init_loop) + co_increment;
         sys.dbms_output.put_line(co_outer_text || l_outerlp ||
                                  co_inner_text || l_innerlp);

         exit outerloop when l_innerlp = co_exit_value;
      end loop innerloop;
   end loop outerloop;
end;
/
```

G-4385: Never use a cursor for loop to check whether a cursor returns data.

⚠ **Major**

Efficiency

REASON

You might process more data than required, which leads to bad performance.

EXAMPLE (BAD)

```
declare
   l_employee_found boolean := false;
   cursor c_employees is
      select employee_id, last_name
        from employees;
begin
   <<check_employees>>
   for r_employee in c_employees
   loop
      l_employee_found := true;
   end loop check_employees;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_employee_found boolean := false;
   cursor c_employees is
      select employee_id, last_name
        from employees;
   r_employee c_employees%rowtype;
begin
   open c_employees;
   fetch c_employees into r_employee;
   l_employee_found := c_employees%found;
   close c_employees;
end;
/
```

G-4390: Avoid use of unreferenced FOR loop indexes.

Major

Efficiency

REASON

If the loop index is used for anything but traffic control inside the loop, this is one of the indicators that a numeric `for` loop is being used incorrectly. The actual body of executable statements completely ignores the loop index. When that is the case, there is a good chance that you do not need the loop at all.

EXAMPLE (BAD)

```
declare
   l_row   pls_integer;
   l_value pls_integer;
   co_lower_bound constant simple_integer           := 1;
   co_upper_bound constant simple_integer           := 5;
   co_row_incr    constant simple_integer           := 1;
   co_value_incr  constant simple_integer           := 10;
   co_delimiter   constant types_up.short_text_type := ' ';
   co_first_value constant simple_integer           := 100;
begin
   l_row := co_lower_bound;
   l_value := co_first_value;
   <<for_loop>>
   for i in co_lower_bound .. co_upper_bound
   loop
      sys.dbms_output.put_line(l_row || co_delimiter || l_value);
      l_row   := l_row + co_row_incr;
      l_value := l_value + co_value_incr;
   end loop for_loop;
end;
/
```

EXAMPLE (GOOD)

```
declare
   co_lower_bound constant simple_integer           := 1;
   co_upper_bound constant simple_integer           := 5;
   co_value_incr  constant simple_integer           := 10;
   co_delimiter   constant types_up.short_text_type := ' ';
   co_first_value constant simple_integer           := 100;
begin
   <<for_loop>>
   for i in co_lower_bound .. co_upper_bound
   loop
      sys.dbms_output.put_line(i || co_delimiter ||
                               to_char(co_first_value + i * co_value_incr));
   end loop for_loop;
end;
/
```

G-4395: Avoid hard-coded upper or lower bound values with FOR loops.

Minor

Changeability, Maintainability

REASON

Your `loop` statement uses a hard-coded value for either its upper or lower bounds. This creates a "weak link" in your program because it assumes that this value will never change. A better practice is to create a named constant (or function) and reference this named element instead of the hard-coded value.

EXAMPLE (BAD)

```
begin
   <<for_loop>>
   for i in 1..5
   loop
      sys.dbms_output.put_line(i);
   end loop for_loop;
end;
/
```

EXAMPLE (GOOD)

```
declare
   co_lower_bound constant simple_integer := 1;
   co_upper_bound constant simple_integer := 5;
begin
   <<for_loop>>
   for i in co_lower_bound..co_upper_bound
   loop
      sys.dbms_output.put_line(i);
   end loop for_loop;
end;
/
```

Exception Handling

G-5010: Try to use a error/logging framework for your application.

Critical

Reliability, Reusability, Testability

Reason

Having a framework to raise/handle/log your errors allows you to easily avoid duplicate application error numbers and having different error messages for the same type of error.

This kind of framework should include

- Logging (different channels like table, mail, file, etc. if needed)
- Error Raising
- Multilanguage support if needed
- Translate ORACLE error messages to a user friendly error text
- Error repository

Example (bad)

```
begin
   sys.dbms_output.put_line('START');
   -- some processing
   sys.dbms_output.put_line('END');
end;
/
```

Example (good)

```
declare
  -- see https://github.com/OraOpenSource/Logger
  l_scope logger_logs.scope%type := 'DEMO';
begin
  logger.log('START', l_scope);
  -- some processing
  logger.log('END', l_scope);
end;
/
```

G-5020: Never handle unnamed exceptions using the error number.

Critical

Maintainability

Reason

When literals are used for error numbers the reader needs the error message manual to unterstand what is going on. Commenting the code or using constants is an option, but it is better to use named exceptions instead, because it ensures a certain level of consistency which makes maintenance easier.

Example (bad)

```
declare
   co_no_data_found constant integer := -1;
begin
   my_package.some_processing(); -- some code which raises an exception
exception
   when too_many_rows then
      my_package.some_further_processing();
   when others then
      if sqlcode = co_no_data_found then
         null;
      end if;
end;
/
```

Example (good)

```
begin
   my_package.some_processing(); -- some code which raises an exception
exception
   when too_many_rows then
      my_package.some_further_processing();
   when no_data_found then
      null; -- handle no_data_found
end;
/
```

G-5030: Never assign predefined exception names to user defined exceptions.

Blocker

Reliability, Testability

Reason

This is error-prone because your local declaration overrides the global declaration. While it is technically possible to use the same names, it causes confusion for others needing to read and maintain this code. Additionally, you will need to be very careful to use the prefix `standard` in front of any reference that needs to use Oracle’s default exception behavior.

Example (bad)

Using the code below, we are not able to handle the `no_data_found` exception raised by the `select` statement as we have overwritten that exception handler. In addition, our exception handler doesn't have an exception number assigned, which should be raised when the `select` statement does not find any rows.

```
declare
   l_dummy dual.dummy%type;
   no_data_found    exception;
   co_rownum        constant simple_integer           := 0;
   co_no_data_found constant types_up.short_text_type := 'no_data_found';
begin
   select dummy
     into l_dummy
     from dual
    where rownum = co_rownum;

   if l_dummy is null then
      raise no_data_found;
   end if;
exception
   when no_data_found then
      sys.dbms_output.put_line(co_no_data_found);
end;
/

Error report -
ORA-01403: no data found
ORA-06512: at line 5
01403. 00000 -  "no data found"
*Cause:    No data was found from the objects.
*Action:   There was no data from the objects which may be due to end of fetch.
```

Example (good)

```
declare
   l_dummy dual.dummy%type;
   empty_value      exception;
   co_rownum        constant simple_integer           := 0;
   co_empty_value   constant types_up.short_text_type := 'empty_value';
   co_no_data_found constant types_up.short_text_type := 'no_data_found';
begin
   select dummy
     into l_dummy
     from dual
    where rownum = co_rownum;

   if l_dummy is null then
      raise empty_value;
   end if;
exception
   when empty_value then
      sys.dbms_output.put_line(co_empty_value);
   when no_data_found then
      sys.dbms_output.put_line(co_no_data_found);
end;
/
```

G-5040: Avoid use of WHEN OTHERS clause in an exception section without any other specific handlers.

Major

Reliability

Reason

There is not necessarily anything wrong with using `when others`, but it can cause you to "lose" error information unless your handler code is relatively sophisticated. Generally, you should use `when others` to grab any and every error only after you have thought about your executable section and decided that you are not able to trap any specific exceptions. If you know, on the other hand, that a certain exception might be raised, include a handler for that error. By declaring two different exception handlers, the code more clearly states what we expect to have happen and how we want to handle the errors. That makes it easier to maintain and enhance. We also avoid hard-coding error numbers in checks against `sqlcode`.

When using a logging framework like Logger, consider making an exception to this rule and allow a `when others` even without other specific handlers, but *only* if the `when others` exception handler calls a logging procedure that saves the error stack (that otherwise is lost) and the last statement of the handler is `raise`.

Example (bad)

```
begin
   my_package.some_processing();
exception
   when others then
      my_package.some_further_processing();
end;
/
```

Example (good)

```
begin
   my_package.some_processing();
exception
   when dup_val_on_index then
      my_package.some_further_processing();
end;
/
```

Example (exception to the rule)

```
begin
   my_package.some_processing();
exception
   when others then
      logger.log_error('Unhandled Exception');
      raise;
end;
/
```

G-5050: Avoid use of the RAISE_APPLICATION_ERROR built-in procedure with a hard-coded 20nnn error number or hard-coded message.

⚠ **Major**

Changeability, Maintainability

Reason

If you are not very organized in the way you allocate, define and use the error numbers between 20999 and 20000 (those reserved by Oracle for its user community), it is very easy to end up with conflicting usages. You should assign these error numbers to named constants and consolidate all definitions within a single package. When you call `raise_application_error`, you should reference these named elements and error message text stored in a table. Use your own raise procedure in place of explicit calls to `raise_application_error`. If you are raising a "system" exception like `no_data_found`, you must use `raise`. However, when you want to raise an application-specific error, you use `raise_application_error`. If you use the latter, you then have to provide an error number and message. This leads to unnecessary and damaging hard-coded values. A more fail-safe approach is to provide a predefined raise procedure that automatically checks the error number and determines the correct way to raise the error.

Example (bad)

```
begin
   raise_application_error(-20501,'Invalid employee_id');
end;
/
```

Example (good)

```
begin
   err_up.raise(in_error => err.co_invalid_employee_id);
end;
/
```

G-5060: Avoid unhandled exceptions.

⚠ **Major**

Reliability

Reason

This may be your intention, but you should review the code to confirm this behavior.

If you are raising an error in a program, then you are clearly predicting a situation in which that error will occur. You should consider including a handler in your code for predictable errors, allowing for a graceful and informative failure. After all, it is much more difficult for an enclosing block to be aware of the various errors you might raise and more importantly, what should be done in response to the error.

The form that this failure takes does not necessarily need to be an exception. When writing functions, you may well decide that in the case of certain exceptions, you will want to return a value such as `null`, rather than allow an exception to propagate out of the function.

Example (bad)

```
create or replace package body department_api is
   function name_by_id (in_id in departments.department_id%type)
      return departments.department_name%type is
      l_department_name departments.department_name%type;
   begin
      select department_name
        into l_department_name
        from departments
       where department_id = in_id;

      return l_department_name;
   end name_by_id;
end department_api;
/
```

Example (good)

```
create or replace package body department_api is
   function name_by_id (in_id in departments.department_id%type)
      return departments.department_name%type is
      l_department_name departments.department_name%type;
   begin
      select department_name
        into l_department_name
        from departments
       where department_id = in_id;

      return l_department_name;
   exception
      when no_data_found then return null;
      when too_many_rows then raise;
   end name_by_id;
end department_api;
/
```

G-5070: Avoid using Oracle predefined exceptions.

Critical

Reliability

Reason

You have raised an exception whose name was defined by Oracle. While it is possible that you have a good reason for "using" one of Oracle's predefined exceptions, you should make sure that you would not be better off declaring your own exception and raising that instead.

If you decide to change the exception you are using, you should apply the same consideration to your own exceptions. Specifically, do not "re-use" exceptions. You should define a separate exception for each error condition, rather than use the same exception for different circumstances.

Being as specific as possible with the errors raised will allow developers to check for, and handle, the different kinds of errors the code might produce.

Example (bad)

```
begin
   raise no_data_found;
end;
/
```

Example (good)

```
declare
   my_exception exception;
begin
   raise my_exception;
end;
/
```

Dynamic SQL

G-6010: Always use a character variable to execute dynamic SQL.

⚠ **Major**

Maintainability, Testability

Reason

Having the executed statement in a variable makes it easier to debug your code (e.g. by logging the statement that failed).

Example (bad)

```
declare
   l_next_val employees.employee_id%type;
begin
   execute immediate 'select employees_seq.nextval from dual' into l_next_val;
end;
/
```

Example (good)

```
declare
   l_next_val employees.employee_id%type;
   co_sql constant types_up.big_string_type :=
       'select employees_seq.nextval from dual';
begin
   execute immediate co_sql into l_next_val;
end;
/
```

G-6020: Try to use output bind arguments in the RETURNING INTO clause of dynamic DML statements rather than the USING clause.

Minor

Maintainability

Reason

When a dynamic `insert`, `update`, or `delete` statement has a `returning` clause, output bind arguments can go in the `returning into` clause or in the `using` clause.

You should use the `returning into` clause for values returned from a DML operation. Reserve `out` and `in out` bind variables for dynamic PL/SQL blocks that return values in PL/SQL variables.

Example (bad)

```
create or replace package body employee_api is
   procedure upd_salary (in_employee_id  in     employees.employee_id%type
                        ,in_increase_pct in     types_up.percentage
                        ,out_new_salary     out employees.salary%type)
   is
      co_sql_stmt constant types_up.big_string_type := '
           update employees set salary = salary + (salary / 100 * :1)
            where employee_id = :2
        returning salary into :3';
   begin
     execute immediate co_sql_stmt
           using in_increase_pct, in_employee_id, out out_new_salary;
   end upd_salary;
end employee_api;
/
```

Example (good)

```
create or replace package body employee_api is
   procedure upd_salary (in_employee_id  in     employees.employee_id%type
                        ,in_increase_pct in     types_up.percentage
                        ,out_new_salary     out employees.salary%type)
   is
      co_sql_stmt constant types_up.big_string_type :=
          'update employees set salary = salary + (salary / 100 * :1)
            where employee_id = :2
        returning salary into :3';
   begin
     execute immediate co_sql_stmt
           using in_increase_pct, in_employee_id
           returning into out_new_salary;
   end upd_salary;
end employee_api;
/
```

Stored Objects

General

G-7110: Try to use named notation when calling program units.

Major

Changeability, Maintainability

REASON

Named notation makes sure that changes to the signature of the called program unit do not affect your call.

This is not needed for standard functions like (`to_char`, `to_date`, `nvl`, `round`, etc.) but should be followed for any other stored object having more than one parameter.

EXAMPLE (BAD)

```
declare
   r_employee employees%rowtype;
   co_id constant employees.employee_id%type := 107;
begin
   employee_api.employee_by_id(r_employee, co_id);
end;
/
```

EXAMPLE (GOOD)

```
declare
   r_employee employees%rowtype;
   co_id constant employees.employee_id%type := 107;
begin
   employee_api.employee_by_id(out_row => r_employee, in_employee_id => co_id);
end;
/
```

G-7120 Always add the name of the program unit to its end keyword.

Minor

Maintainability

REASON

It's a good alternative for comments to indicate the end of program units, especially if they are lengthy or nested.

EXAMPLE (BAD)

```
create or replace package body employee_api is
   function employee_by_id (in_employee_id in employees.employee_id%type)
      return employees%rowtype is
      r_employee employees%rowtype;
   begin
      select *
        into r_employee
        from employees
       where employee_id = in_employee_id;

      return r_employee;
   exception
      when no_data_found then
         null;
      when too_many_rows then
         raise;
   end;
end;
/
```

EXAMPLE (GOOD)

```
create or replace package body employee_api is
   function employee_by_id (in_employee_id in employees.employee_id%type)
      return employees%rowtype is
      r_employee employees%rowtype;
   begin
      select *
        into r_employee
        from employees
       where employee_id = in_employee_id;

      return r_employee;
   exception
      when no_data_found then
         null;
      when too_many_rows then
         raise;
   end employee_by_id;
end employee_api;
/
```

G-7130: Always use parameters or pull in definitions rather than referencing external variables in a local program unit.

⚠ **Major**

Maintainability, Reliability, Testability

REASON

Local procedures and functions offer an excellent way to avoid code redundancy and make your code more readable (and thus more maintainable). Your local program refers, however, an external data structure, i.e., a variable that is declared outside of the local program. Thus, it is acting as a global variable inside the program.

This external dependency is hidden, and may cause problems in the future. You should instead add a parameter to the parameter list of this program and pass the value through the list. This technique makes your program more reusable and avoids scoping problems, i.e. the program unit is less tied to particular variables in the program. In addition, unit encapsulation makes maintenance a lot easier and cheaper.

EXAMPLE (BAD)

```
create or replace package body employee_api is
   procedure calc_salary (in_employee_id in employees.employee_id%type) is
      r_emp employees%rowtype;

      function commission return number is
         l_commission employees.salary%type := 0;
      begin
         if r_emp.commission_pct is not null
         then
            l_commission := r_emp.salary * r_emp.commission_pct;
         end if;

         return l_commission;
      end commission;
   begin
      select *
        into r_emp
        from employees
       where employee_id = in_employee_id;

      sys.dbms_output.put_line(r_emp.salary + commission());
   exception
      when no_data_found then
         null;
      when too_many_rows then
         null;
   end calc_salary;
end employee_api;
/
```

EXAMPLE (GOOD)

```
create or replace package body employee_api is
   procedure calc_salary (in_employee_id in employees.employee_id%type) is
      r_emp employees%rowtype;

      function commission (in_salary   in employees.salary%type
                          ,in_comm_pct in employees.commission_pct%type)
         return number is
         l_commission employees.salary%type := 0;
      begin
         if in_comm_pct is not null then
            l_commission := in_salary * in_comm_pct;
         end if;

         return l_commission;
      end commission;
   begin
      select *
        into r_emp
        from employees
       where employee_id = in_employee_id;

      sys.dbms_output.put_line(
         r_emp.salary + commission(in_salary   => r_emp.salary
                                  ,in_comm_pct => r_emp.commission_pct)
      );
   exception
      when no_data_found then
         null;
      when too_many_rows then
         null;
   end calc_salary;
end employee_api;
/
```

G-7140: Always ensure that locally defined procedures or functions are referenced.

⚠ **Major**

Maintainability, Reliability

REASON

This can occur as the result of changes to code over time, but you should make sure that this situation does not reflect a problem. And you should remove the declaration to avoid maintenance errors in the future.

You should go through your programs and remove any part of your code that is no longer used. This is a relatively straightforward process for variables and named constants. Simply execute searches for a variable's name in that variable's scope. If you find that the only place it appears is in its declaration, delete the declaration.

There is never a better time to review all the steps you took, and to understand the reasons you took them, then immediately upon completion of your program. If you wait, you will find it particularly difficult to remember those parts of the program that were needed at one point, but were rendered unnecessary in the end.

EXAMPLE (BAD)

```
create or replace package body my_package is
   procedure my_procedure is
      function my_func return number is
         co_true constant integer := 1;
      begin
         return co_true;
      end my_func;
   begin
       null;
   end my_procedure;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package body my_package is
   procedure my_procedure is
      function my_func return number is
         co_true constant integer := 1;
      begin
         return co_true;
      end my_func;
   begin
       sys.dbms_output.put_line(my_func());
   end my_procedure;
end my_package;
/
```

G-7150: Try to remove unused parameters.

Minor

Efficiency, Maintainability

REASON

You should go through your programs and remove any parameter that is no longer used.

EXAMPLE (BAD)

```
create or replace package body department_api is
   function name_by_id (in_department_id in departments.department_id%type
                       ,in_manager_id    in departments.manager_id%type)
      return departments.department_name%type is
      l_department_name departments.department_name%type;
   begin
      <<find_department>>
      begin
         select department_name
           into l_department_name
           from departments
          where department_id = in_department_id;
      exception
         when no_data_found or too_many_rows then
            l_department_name := null;
      end find_department;

      return l_department_name;
   end name_by_id;
end department_api;
/
```

EXAMPLE (GOOD)

```
create or replace package body department_api is
   function name_by_id (in_department_id in departments.department_id%type)
      return departments.department_name%type is
      l_department_name departments.department_name%type;
   begin
      <<find_department>>
      begin
         select department_name
           into l_department_name
           from departments
          where department_id = in_department_id;
      exception
         when no_data_found or too_many_rows then
            l_department_name := null;
      end find_department;

      return l_department_name;
   end name_by_id;
end department_api;
/
```

G-7160: Always explicitly state parameter mode.

Major

Maintainability

REASON

By showing the mode of parameters, you help the reader. If you do not specify a parameter mode, the default mode is `in`. Explicitly showing the mode indication of all parameters is a more assertive action than simply taking the default mode. Anyone reviewing the code later will be more confident that you intended the parameter mode to be `in`, `out` or `in out`.

EXAMPLE (BAD)

```
create or replace package employee_api is
   procedure upsert (io_id             in out employees.id%type
                    ,in_first_name            employees.first_name%type
                    ,in_last_name             employees.last_name%type
                    ,in_email                 employees.email%type
                    ,in_department_id         employees.department_id%type
                    ,out_success       out    pls_integer);
end employee_up;
/
```

EXAMPLE (GOOD)

```
create or replace package employee_api is
   procedure upsert (io_id             in out employees.id%type
                    ,in_first_name     in     employees.first_name%type
                    ,in_last_name      in     employees.last_name%type
                    ,in_email          in     employees.email%type
                    ,in_department_id  in     employees.department_id%type
                    ,out_success       out    pls_integer);
end employee_up;
/
```

G-7170: Avoid using an IN OUT parameter as IN or OUT only.

Major

Efficiency, Maintainability

Unsupported in PL/SQL Cop Validators

Rule G-7170 is not expected to be implemented in the static code analysis validators.

REASON

Avoid using parameter mode `in out` unless you actually use the parameter both as input and output. If the code body only reads from the parameter, use `in`; if the code body only assigns to the parameter, use `out`. If at the beginning of a project you expect a parameter to be both input and output and therefore choose `in out` just in case, but later development shows the parameter actually is only `in` or `out`, you should change the parameter mode accordingly.

EXAMPLE (BAD)

```
create or replace package body employee_up is
   procedure rcv_emp (io_first_name     in out employees.first_name%type
                     ,io_last_name      in out employees.last_name%type
                     ,io_email          in out employees.email%type
                     ,io_phone_number   in out employees.phone_number%type
                     ,io_hire_date      in out employees.hire_date%type
                     ,io_job_id         in out employees.job_id%type
                     ,io_salary         in out employees.salary%type
                     ,io_commission_pct in out employees.commission_pct%type
                     ,io_manager_id     in out employees.manager_id%type
                     ,io_department_id  in out employees.department_id%type
                     ,in_wait           in     integer) is
      l_status pls_integer;
      co_dflt_pipe_name constant string(30 char) := 'MyPipe';
      co_ok constant pls_integer := 1;
   begin
      -- Receive next message and unpack for each column.
      l_status := sys.dbms_pipe.receive_message(pipename => co_dflt_pipe_name
                                               ,timeout  => in_wait);
      if l_status = co_ok then
         sys.dbms_pipe.unpack_message (io_first_name);
         sys.dbms_pipe.unpack_message (io_last_name);
         sys.dbms_pipe.unpack_message (io_email);
         sys.dbms_pipe.unpack_message (io_phone_number);
         sys.dbms_pipe.unpack_message (io_hire_date);
         sys.dbms_pipe.unpack_message (io_job_id);
         sys.dbms_pipe.unpack_message (io_salary);
         sys.dbms_pipe.unpack_message (io_commission_pct);
         sys.dbms_pipe.unpack_message (io_manager_id);
         sys.dbms_pipe.unpack_message (io_department_id);
      end if;
   end rcv_emp;
end employee_up;
/
```

EXAMPLE (GOOD)

```
create or replace package body employee_up is
   procedure rcv_emp (out_first_name     out employees.first_name%type
                     ,out_last_name      out employees.last_name%type
                     ,out_email          out employees.email%type
                     ,out_phone_number   out employees.phone_number%type
                     ,out_hire_date      out employees.hire_date%type
                     ,out_job_id         out employees.job_id%type
                     ,out_salary         out employees.salary%type
                     ,out_commission_pct out employees.commission_pct%type
                     ,out_manager_id     out employees.manager_id%type
                     ,out_department_id  out employees.department_id%type
                     ,in_wait            in  integer) is
      l_status pls_integer;
      co_dflt_pipe_name constant string(30 char) := 'MyPipe';
      co_ok constant pls_integer := 1;
   begin
      -- Receive next message and unpack for each column.
      l_status := sys.dbms_pipe.receive_message(pipename => co_dflt_pipe_name
                                               ,timeout  => in_wait);
      if l_status = co_ok then
         sys.dbms_pipe.unpack_message (out_first_name);
         sys.dbms_pipe.unpack_message (out_last_name);
         sys.dbms_pipe.unpack_message (out_email);
         sys.dbms_pipe.unpack_message (out_phone_number);
         sys.dbms_pipe.unpack_message (out_hire_date);
         sys.dbms_pipe.unpack_message (out_job_id);
         sys.dbms_pipe.unpack_message (out_salary);
         sys.dbms_pipe.unpack_message (out_commission_pct);
         sys.dbms_pipe.unpack_message (out_manager_id);
         sys.dbms_pipe.unpack_message (out_department_id);
      end if;
   end rcv_emp;
end employee_up;
/
```

Packages

G-7210: Try to keep your packages small. Include only few procedures and functions that are used in the same context.

Minor

Efficiency, Maintainability

REASON

The entire package is loaded into memory when the package is called the first time. To optimize memory consumption and keep load time small packages should be kept small but include components that are used together.

G-7220: Always use forward declaration for private functions and procedures.

Minor

Changeability

REASON

Having forward declarations allows you to order the functions and procedures of the package in a reasonable way.

EXAMPLE (BAD)

```
create or replace package department_api is
   procedure del (in_department_id in departments.department_id%type);
end department_api;
/

create or replace package body department_api is
   function does_exist (in_department_id in departments.department_id%type)
      return boolean is
      l_return pls_integer;
   begin
      <<check_row_exists>>
      begin
         select 1
           into l_return
           from departments
          where department_id = in_department_id;
      exception
         when no_data_found or too_many_rows then
            l_return := 0;
      end check_row_exists;

      return l_return = 1;
   end does_exist;

   procedure del (in_department_id in departments.department_id%type) is
   begin
      if does_exist(in_department_id) then
        null;
      end if;
   end del;
end department_api;
/
```

EXAMPLE (GOOD)

```
create or replace package department_api is
   procedure del (in_department_id in departments.department_id%type);
end department_api;
/

create or replace package body department_api is
   function does_exist (in_department_id in departments.department_id%type)
      return boolean;

   procedure del (in_department_id in departments.department_id%type) is
   begin
      if does_exist(in_department_id) then
        null;
      end if;
   end del;

   function does_exist (in_department_id in departments.department_id%type)
      return boolean is
      l_return pls_integer;
   begin
      <<check_row_exists>>
      begin
         select 1
           into l_return
           from departments
          where department_id = in_department_id;
      exception
         when no_data_found or too_many_rows then
            l_return := 0;
      end check_row_exists;

      return l_return = 1;
   end does_exist;
end department_api;
/
```

G-7230: Avoid declaring global variables public.

⚠ **Major**

Reliability

REASON

You should always declare package-level data (non-constants) inside the package body. You can then define "get and set" methods (functions and procedures, respectively) in the package specification to provide controlled access to that data. By doing so you can guarantee data integrity, you can change your data structure implementation, and also track access to those data structures.

Data structures (scalar variables, collections, cursors) declared in the package specification (not within any specific program) can be referenced directly by any program running in a session with `execute` rights to the package.

Instead, declare all package-level data in the package body and provide "get and set" methods - a function to get the value and a procedure to set the value - in the package specification. Developers then can access the data using these methods - and will automatically follow all rules you set upon data modification.

For package-level constants, consider whether the constant should be public and usable from other code, or if only relevant for code within the package. If the latter, declare the constant in the package body. If the former, it is typically good practice to place the constants in a package specification that only holds constants.

EXAMPLE (BAD)

```
create or replace package employee_api as
   co_min_increase constant types_up.sal_increase_type := 0.01;
   co_max_increase constant types_up.sal_increase_type := 0.5;
   g_salary_increase types_up.sal_increase_type := co_min_increase;

   procedure set_salary_increase (in_increase in types_up.sal_increase_type);
   function salary_increase return types_up.sal_increase_type;
end employee_api;
/

create or replace package body employee_api as
   procedure set_salary_increase (in_increase in types_up.sal_increase_type) is
   begin
      g_salary_increase := greatest(least(in_increase,co_max_increase)
                                   ,co_min_increase);
   end set_salary_increase;

   function salary_increase return types_up.sal_increase_type is
   begin
      return g_salary_increase;
   end salary_increase;
end employee_api;
/
```

EXAMPLE (GOOD)

```
create or replace package constants_up as
   co_min_increase constant types_up.sal_increase_type := 0.01;
   co_max_increase constant types_up.sal_increase_type := 0.5;
end constants_up;
/

create or replace package employee_api as
   procedure set_salary_increase (in_increase in types_up.sal_increase_type);
   function salary_increase return types_up.sal_increase_type;
end employee_api;
/

create or replace package body employee_api as
   g_salary_increase types_up.sal_increase_type(4,2);

   procedure init;

   procedure set_salary_increase (in_increase in types_up.sal_increase_type) is
   begin
      g_salary_increase := greatest(least(in_increase
                                         ,constants_up.co_max_increase)
                                   ,constants_up.co_min_increase);
   end set_salary_increase;

   function salary_increase return types_up.sal_increase_type is
   begin
      return g_salary_increase;
   end salary_increase;

   procedure init
   is
   begin
      g_salary_increase := constants_up.co_min_increase;
   end init;
begin
   init();
end employee_api;
/
```

G-7240: Never use RETURN in package initialization block.

Minor

Maintainability

REASON

The purpose of the initialization block of a package body is to set initial values of the global variables of the package (initialize the package state). Although `return` is syntactically allowed in this block, it makes no sense. If it is the last keyword of the block, it is superfluous. If it is not the last keyword, then all code after the `return` is unreachable and thus dead code.

EXAMPLE (BAD)

```
create or replace package body employee_api as
   g_salary_increase types_up.sal_increase_type(4,2);

   procedure set_salary_increase (in_increase in types_up.sal_increase_type) is
   begin
      g_salary_increase := greatest(least(in_increase
                                         ,constants_up.max_salary_increase())
                                   ,constants_up.min_salary_increase());
   end set_salary_increase;

   function salary_increase return types_up.sal_increase_type is
   begin
      return g_salary_increase;
   end salary_increase;

begin
   g_salary_increase := constants_up.min_salary_increase();

   return;

   set_salary_increase(constants_up.min_salary_increase()); -- dead code
end employee_api;
/
```

EXAMPLE (GOOD)

```
create or replace package body employee_api as
   g_salary_increase types_up.sal_increase_type(4,2);

   procedure set_salary_increase (in_increase in types_up.sal_increase_type) is
   begin
      g_salary_increase := greatest(least(in_increase
                                         ,constants_up.max_salary_increase())
                                   ,constants_up.min_salary_increase());
   end set_salary_increase;

   function salary_increase return types_up.sal_increase_type is
   begin
      return g_salary_increase;
   end salary_increase;

begin
   g_salary_increase := constants_up.min_salary_increase();
end employee_api;
/
```

Procedures

G-7310: Avoid standalone procedures – put your procedures in packages.

Minor

Maintainability

REASON

Use packages to structure your code, combine procedures and functions which belong together.

Package bodies may be changed and compiled without invalidating other packages. This is major advantage compared to standalone procedures and functions.

EXAMPLE (BAD)

```
create or replace procedure my_procedure is
begin
   null;
end my_procedure;
/
```

EXAMPLE (GOOD)

```
create or replace package my_package is
   procedure my_procedure;
end my_package;
/

create or replace package body my_package is
   procedure my_procedure is
   begin
      null;
   end my_procedure;
end my_package;
/
```

G-7320: Avoid using RETURN statements in a PROCEDURE.

Major

Maintainability, Testability

REASON

Use of the `return` statement is legal within a procedure in PL/SQL, but it is very similar to a `goto`, which means you end up with poorly structured code that is hard to debug and maintain.

A good general rule to follow as you write your PL/SQL programs is "one way in and one way out". In other words, there should be just one way to enter or call a program, and there should be one way out, one exit path from a program (or loop) on successful termination. By following this rule, you end up with code that is much easier to trace, debug, and maintain.

EXAMPLE (BAD)

```
create or replace package body my_package is
   procedure my_procedure is
      l_idx simple_integer := 1;
      co_modulo constant simple_integer := 7;
   begin
      <<mod7_loop>>
      loop
        if mod(l_idx,co_modulo) = 0 then
           return;
        end if;

        l_idx := l_idx + 1;
      end loop mod7_loop;
   end my_procedure;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package body my_package is
   procedure my_procedure is
      l_idx simple_integer := 1;
      co_modulo constant simple_integer := 7;
   begin
      <<mod7_loop>>
      loop
        exit mod7_loop when mod(l_idx,co_modulo) = 0;

        l_idx := l_idx + 1;
      end loop mod7_loop;
   end my_procedure;
end my_package;
/
```

G-7330: Always assign values to OUT parameters.

Major

Maintainability, Testability

REASON

Marking a parameter for output means that callers will expect its value to be updated with a result from the execution of the procedure. Failing to update the parameter before the procedure returns is surely an error.

EXAMPLE (BAD)

```
create or replace package body my_package is
   procedure greet(
      in_name      in  varchar2
    , out_greeting out varchar2
   ) is
      l_message varchar2(100 char);
   begin
      l_message := 'Hello, ' || in_name;
   end my_procedure;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package body my_package is
   procedure greet(
      in_name      in  varchar2
    , out_greeting out varchar2
   ) is
   begin
      out_greeting := 'Hello, ' || in_name;
   end my_procedure;
end my_package;
/
```

Functions

G-7410: Avoid standalone functions – put your functions in packages.

Minor

Maintainability

REASON

Use packages to structure your code, combine procedures and functions which belong together.

Package bodies may be changed and compiled without invalidating other packages. This is major advantage compared to standalone procedures and functions.

EXAMPLE (BAD)

```
create or replace function my_function return varchar2 is
begin
   return null;
end my_function;
/
```

EXAMPLE (GOOD)

```
create or replace package body my_package is
   function my_function return varchar2 is
   begin
      return null;
   end my_function;
end my_package;
/
```

G-7420: Always make the RETURN statement the last statement of your function.

Major

Maintainability

REASON

The reader expects the `return` statement to be the last statement of a function.

EXAMPLE (BAD)

```
create or replace package body my_package is
   function my_function (in_from in pls_integer
                       , in_to   in pls_integer) return pls_integer is
      l_ret pls_integer;
   begin
      l_ret := in_from;
      <<for_loop>>
      for i in in_from .. in_to
      loop
         l_ret := l_ret + i;
         if i = in_to then
            return l_ret;
         end if;
      end loop for_loop;
   end my_function;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package body my_package is
   function my_function (in_from in pls_integer
                       , in_to   in pls_integer) return pls_integer is
      l_ret pls_integer;
   begin
      l_ret := in_from;
      <<for_loop>>
      for i in in_from .. in_to
      loop
         l_ret := l_ret + i;
      end loop for_loop;
      return l_ret;
   end my_function;
end my_package;
/
```

G-7430: Try to use no more than one RETURN statement within a function.

> ⚠ **Major**
>
> Will have a medium/potential impact on the maintenance cost. Maintainability, Testability

REASON

A function should have a single point of entry as well as a single exit-point.

EXAMPLE (BAD)

```sql
create or replace package body my_package is
   function my_function (in_value in pls_integer) return boolean is
      co_yes constant pls_integer := 1;
   begin
      if in_value = co_yes then
         return true;
      else
         return false;
      end if;
   end my_function;
end my_package;
/
```

EXAMPLE (BETTER)

```sql
create or replace package body my_package is
   function my_function (in_value in pls_integer) return boolean is
      co_yes constant pls_integer := 1;
      l_ret boolean;
   begin
      if in_value = co_yes then
         l_ret := true;
      else
         l_ret := false;
      end if;

      return l_ret;
   end my_function;
end my_package;
/
```

EXAMPLE (GOOD)

```sql
create or replace package body my_package is
   function my_function (in_value in pls_integer) return boolean is
      co_yes constant pls_integer := 1;
   begin
      return in_value = co_yes;
   end my_function;
end my_package;
/
```

G-7440: Never use OUT parameters to return values from a function.

⚠ **Major**

Reusability

REASON

A function should return all its data through the `return` clause. Having an `out` parameter prohibits usage of a function within SQL statements.

EXAMPLE (BAD)

```
create or replace package body my_package is
   function my_function (out_date out date) return boolean is
   begin
      out_date := sysdate;
      return true;
   end my_function;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package body my_package is
   function my_function return date is
   begin
      return sysdate;
   end my_function;
end my_package;
/
```

G-7450: Never return a NULL value from a BOOLEAN function.

⚠ **Major**

Reliability, Testability

REASON

If a boolean function returns `null`, the caller has do deal with it. This makes the usage cumbersome and more error-prone.

EXAMPLE (BAD)

```
create or replace package body my_package is
   function my_function return boolean is
   begin
      return null;
   end my_function;
end my_package;
/
```

EXAMPLE (GOOD)

```
create or replace package body my_package is
   function my_function return boolean is
   begin
      return true;
   end my_function;
end my_package;
/
```

G-7460: Try to define your packaged/standalone function deterministic if appropriate.

⚠ **Major**

Efficiency

REASON

A deterministic function (always return same result for identical parameters) which is defined to be deterministic will be executed once per different parameter within a SQL statement whereas if the function is not defined to be deterministic it is executed once per result row.

EXAMPLE (BAD)

```
create or replace package department_api is
   function name_by_id (in_department_id in departments.department_id%type)
      return departments.department_name%type;
end department_api;
/
```

EXAMPLE (GOOD)

```
create or replace package department_api is
   function name_by_id (in_department_id in departments.department_id%type)
      return departments.department_name%type deterministic;
end department_api;
/
```

Oracle Supplied Packages

G-7510: Always prefix ORACLE supplied packages with owner schema name.

⚠ **Major**

Security

REASON

The signature of oracle-supplied packages is well known and therefore it is quite easy to provide packages with the same name as those from oracle doing something completely different without you noticing it.

EXAMPLE (BAD)

```
declare
   co_hello_world constant string(30 char) := 'Hello World';
begin
   dbms_output.put_line(co_hello_world);
end;
/
```

EXAMPLE (GOOD)

```
declare
   co_hello_world constant string(30 char) := 'Hello World';
begin
   sys.dbms_output.put_line(co_hello_world);
end;
/
```

Object Types

There are no object type-specific recommendations to be defined at the time of writing.

Triggers

G-7710: Avoid cascading triggers.

⚠ **Major**

Maintainability, Testability

REASON

Having triggers that act on other tables in a way that causes triggers on that table to fire lead to obscure behavior.

EXAMPLE (BAD)

```
create or replace trigger dept_br_u
before update on departments for each row
begin
   insert into departments_hist (department_id
                                ,department_name
                                ,manager_id
                                ,location_id
                                ,modification_date)
        values (:old.department_id
               ,:old.department_name
               ,:old.manager_id
               ,:old.location_id
               ,sysdate);
end;
/
create or replace trigger dept_hist_br_i
before insert on departments_hist for each row
begin
   insert into departments_log (department_id
                               ,department_name
                               ,modification_date)
                        values (:new.department_id
                               ,:new.department_name
                               ,sysdate);
end;
/
```

EXAMPLE (GOOD)

```
create or replace trigger dept_br_u
before update on departments for each row
begin
   insert into departments_hist (department_id
                                ,department_name
                                ,manager_id
                                ,location_id
                                ,modification_date)
        values (:old.department_id
               ,:old.department_name
               ,:old.manager_id
               ,:old.location_id
               ,sysdate);

   insert into departments_log (department_id
                               ,department_name
                               ,modification_date)
                        values (:old.department_id
                               ,:old.department_name
                               ,sysdate);

end;
/
```

G-7720: Never use multiple UPDATE OF in trigger event clause.

Blocker

Maintainability, Reliability, Testability

REASON

A DML trigger can have multiple triggering events separated by `or` like `before insert or delete or update of some_column`. If you have multiple `update of` separated by `or`, only one of them (the last one) is actually used and you get no error message, so you have a bug waiting to happen. Instead you always should use a single `update of` with all columns comma-separated, or an `update` without `of` if you wish all columns.

EXAMPLE (BAD)

```
create or replace trigger dept_br_u
before update of department_id or update of department_name
on departments for each row
begin
   -- will only fire on updates of department_name
   insert into departments_log (department_id
                               ,department_name
                               ,modification_date)
                        values (:old.department_id
                               ,:old.department_name
                               ,sysdate);
end;
/
```

EXAMPLE (GOOD)

```
create or replace trigger dept_br_u
before update of department_id, department_name
on departments for each row
begin
   insert into departments_log (department_id
                               ,department_name
                               ,modification_date)
                        values (:old.department_id
                               ,:old.department_name
                               ,sysdate);
end;
/
```

G-7730: Avoid multiple DML events per trigger if primary key is assigned in trigger.

Major

Efficiency, Reliability

REASON

If a trigger makes assignment to the primary key anywhere in the trigger code, that causes the session firing the trigger to take a lock on any child tables with a foreign key to this primary key. Even if the assignment is in for example an `if inserting` block and the trigger is fired by an `update` statement, such locks still happen unnecessarily. The issue is avoided by having one trigger for the insert containing the primary key assignment, and another trigger for the update. Or even better by handling the insert assignment as ´default on null´ clauses, so that only an `on update` trigger is needed.

EXAMPLE (BAD)

```
create or replace trigger dept_br_iu
before insert or update
on departments for each row
begin
   if inserting then
      :new.department_id := department_seq.nextval;
      :new.created_date  := sysdate;
   end if;
   if updating then
      :new.changed_date  := sysdate;
   end if;
end;
/
```

EXAMPLE (BETTER)

```
create or replace trigger dept_br_i
before insert
on departments for each row
begin
   :new.department_id := department_seq.nextval;
   :new.created_date  := sysdate;
end;
/

create or replace trigger dept_br_u
before update
on departments for each row
begin
   :new.changed_date  := sysdate;
end;
/
```

EXAMPLE (GOOD)

```
alter table department modify department_id default on null department_seq.nextval;
alter table department modify created_date  default on null sysdate;

create or replace trigger dept_br_u
before update
on departments for each row
begin
   :new.changed_date  := sysdate;
end;
/
```

Sequences

G-7810: Never use SQL inside PL/SQL to read sequence numbers (or SYSDATE).

⚠ **Major**

Efficiency, Maintainability

REASON

Since ORACLE 11g it is no longer needed to use a `select` statement to read a sequence (which would imply a context switch).

EXAMPLE (BAD)

```
declare
   l_sequence_number employees.emloyee_id%type;
begin
   select employees_seq.nextval
     into l_sequence_number
     from dual;
end;
/
```

EXAMPLE (GOOD)

```
declare
   l_sequence_number employees.emloyee_id%type;
begin
   l_sequence_number := employees_seq.nextval;
end;
/
```

Patterns

Checking the Number of Rows

G-8110: Never use SELECT COUNT(*) if you are only interested in the existence of a row.

Major

Efficiency

REASON

If you do a `select count(*)` all rows will be read according to the `where` clause, even if only the availability of data is of interest. For this we have a big performance overhead. If we do a `select count(*) ... where rownum = 1` there is also a overhead as there will be two communications between the PL/SQL and the SQL engine. See the following example for a better solution.

EXAMPLE (BAD)

```
declare
   l_count pls_integer;
   co_zero   constant simple_integer := 0;
   co_salary constant employees.salary%type := 5000;
begin
   select count(*)
     into l_count
     from employees
    where salary < co_salary;
    if l_count > co_zero then
       <<emp_loop>>
       for r_emp in (select employee_id
                       from employees)
       loop
          if r_emp.salary < co_salary then
             my_package.my_proc(in_employee_id => r_emp.employee_id);
          end if;
       end loop emp_loop;
    end if;
end;
/
```

EXAMPLE (GOOD)

```
declare
   co_salary constant employees.salary%type := 5000;
begin
    <<emp_loop>>
    for r_emp in (select e1.employee_id
                    from employees e1
                   where exists(select e2.salary
                                  from employees e2
                                 where e2.salary < co_salary))
    loop
       my_package.my_proc(in_employee_id => r_emp.employee_id);
    end loop emp_loop;
end;
/
```

G-8120: Never check existence of a row to decide whether to create it or not.

⚠ **Major**

Efficiency, Reliability

REASON

The result of an existence check is a snapshot of the current situation. You never know whether in the time between the check and the (insert) action someone else has decided to create a row with the values you checked. Therefore, you should only rely on constraints when it comes to prevention of duplicate records.

EXAMPLE (BAD)

```
create or replace package body department_api is
   procedure ins (in_r_department in departments%rowtype) is
      l_count pls_integer;
   begin
      select count(*)
        into l_count
        from departments
       where department_id = in_r_department.department_id;

       if l_count = 0 then
          insert into departments
               values in_r_department;
       end if;
   end ins;
end department_api;
/
```

EXAMPLE (GOOD)

```
create or replace package body department_api is
   procedure ins (in_r_department in departments%rowtype) is
   begin
      insert into departments
           values in_r_department;
   exception
      when dup_val_on_index then null; -- handle exception
   end ins;
end department_api;
/
```

Access objects of foreign application schemas

G-8210: Always use synonyms when accessing objects of another application schema.

⚠ **Major**

Changeability, Maintainability

REASON

If a connection is needed to a table that is placed in a foreign schema, using synonyms is a good choice. If there are structural changes to that table (e.g. the table name changes or the table changes into another schema) only the synonym has to be changed no changes to the package are needed (single point of change). If you only have read access for a table inside another schema, or there is another reason that does not allow you to change data in this table, you can switch the synonym to a table in your own schema. This is also good practice for testers working on test systems.

EXAMPLE (BAD)

```
declare
   l_product_name oe.products.product_name%type;
   co_price constant oe.products@list_price%type := 1000;
begin
   select p.product_name
     into l_product_name
     from oe.products p
    where list_price > co_price;
exception
   when no_data_found then
      null; -- handle_no_data_found;
   when too_many_rows then
      null; -- handle_too_many_rows;
end;
/
```

EXAMPLE (GOOD)

```
create synonym oe_products for oe.products;

declare
   l_product_name oe_products.product_name%type;
   co_price constant oe_products.list_price%type := 1000;
begin
   select p.product_name
     into l_product_name
     from oe_products p
    where list_price > co_price;
exception
   when no_data_found then
      null; -- handle_no_data_found;
   when too_many_rows then
      null; -- handle_too_many_rows;
end;
/
```

Validating input parameter size

G-8310: Always validate input parameter size by assigning the parameter to a size limited variable in the declaration section of program unit.

Minor

Maintainability, Reliability, Reusability, Testability

REASON

This technique raises an error (`value_error`) which may not be handled in the called program unit. This is the right way to do it, as the error is not within this unit but when calling it, so the caller should handle the error.

EXAMPLE (BAD)

```
create or replace package body department_api is
   function dept_by_name (in_dept_name in departments.department_name%type)
      return departments%rowtype is
      l_return departments%rowtype;
   begin
       if    in_dept_name is null
          or length(in_dept_name) > 20
       then
          raise err.e_param_to_large;
       end if;
       -- get the department by name
       select *
         from departments
        where department_name = in_dept_name;

       return l_return;
   end dept_by_name;
end department_api;
/
```

EXAMPLE (GOOD)

```
create or replace package body department_api is
   function dept_by_name (in_dept_name in departments.department_name%type)
      return departments%rowtype is
      l_dept_name departments.department_name%type not null := in_dept_name;
      l_return departments%rowtype;
   begin
       -- get the department by name
       select *
         from departments
        where department_name = l_dept_name;

       return l_return;
   end dept_by_name;
end department_api;
/
```

FUNCTION CALL

```
...
   r_department := department_api.dept_by_name('Far to long name of a department');
...
exception
   when value_error then ...
```

Ensure single execution at a time of a program unit

G-8410: Always use application locks to ensure a program unit is only running once at a given time.

Minor

Efficiency, Reliability

REASON

This technique allows us to have locks across transactions as well as a proven way to clean up at the end of the session.

The alternative using a table where a “Lock-Row” is stored has the disadvantage that in case of an error a proper cleanup has to be done to “unlock” the program unit.

EXAMPLE (BAD)

```
-- Bad
/* Example */
create or replace package body lock_up is
   -- manage locks in a dedicated table created as follows:
   --   CREATE TABLE app_locks (
   --      lock_name VARCHAR2(128 CHAR) NOT NULL primary key
   --   );

   procedure request_lock (in_lock_name in varchar2) is
   begin
      -- raises dup_val_on_index
      insert into app_locks (lock_name) values (in_lock_name);
   end request_lock;

   procedure release_lock(in_lock_name in varchar2) is
   begin
      delete from app_locks where lock_name = in_lock_name;
   end release_lock;
end lock_up;
/

/* Call bad example */
declare
   co_lock_name constant varchar2(30 char) := 'APPLICATION_LOCK';
begin
   lock_up.request_lock(in_lock_name => co_lock_name);
   -- processing
   lock_up.release_lock(in_lock_name => co_lock_name);
exception
   when others then
      -- log error
      lock_up.release_lock(in_lock_name => co_lock_name);
      raise;
end;
/
```

EXAMPLE (GOOD)

```
/* Example */
create or replace package body lock_up is
   function request_lock(
      in_lock_name         in varchar2,
      in_release_on_commit in boolean := false)
   return varchar2 is
      l_lock_handle varchar2(128 char);
   begin
      sys.dbms_lock.allocate_unique(
         lockname        => in_lock_name,
         lockhandle      => l_lock_handle,
         expiration_secs => constants_up.co_one_week
      );
      if sys.dbms_lock.request(
            lockhandle        => l_lock_handle,
            lockmode          => sys.dbms_lock.x_mode,
            timeout           => sys.dbms_lock.maxwait,
            release_on_commit => coalesce(in_release_on_commit, false)
         ) > 0
      then
         raise err.e_lock_request_failed;
      end if;
      return l_lock_handle;
   end request_lock;

   procedure release_lock(in_lock_handle in varchar2) is
   begin
      if sys.dbms_lock.release(lockhandle => in_lock_handle) > 0 then
         raise err.e_lock_request_failed;
      end if;
   end release_lock;
end lock_up;
/

/* Call good example */
declare
   l_handle varchar2(128 char);
   co_lock_name constant varchar2(30 char) := 'APPLICATION_LOCK';
begin
   l_handle := lock_up.request_lock(in_lock_name => co_lock_name);
   -- processing
   lock_up.release_lock(in_lock_handle => l_handle);
exception
   when others then
      -- log error
      lock_up.release_lock(in_lock_handle => l_handle);
      raise;
end;
/
```

Use dbms_application_info package to follow progress of a process

G-8510: Always use dbms_application_info to track program process transiently.

Minor

Efficiency, Reliability

REASON

This technique allows us to view progress of a process without having to persistently write log data in either a table or a file. The information is accessible through the `v$session` view.

EXAMPLE (BAD)

```
create or replace package body employee_api is
   procedure process_emps is
   begin
      <<employees>>
      for emp_rec in (select employee_id
                        from employees
                       order by employee_id)
      loop
         null; -- some processing
      end loop employees;
   end process_emps;
end employee_api;
/
```

EXAMPLE (GOOD)

```
create or replace package body employee_api is
   procedure process_emps is
   begin
      sys.dbms_application_info.set_module(module_name => $$plsql_unit
                                          ,action_name => 'Init');
      <<employees>>
      for emp_rec in (select employee_id
                        from employees
                       order by employee_id)
      loop
         sys.dbms_application_info.set_action('Processing ' || emp_rec.employee_id);
      end loop employees;
   end process_emps;
end employee_api;
/
```

Complexity Analysis

Using software metrics like complexity analysis will guide you towards maintainable and testable pieces of code by reducing the complexity and splitting the code into smaller chunks.

Halstead Metrics

Calculation

First, we need to compute the following numbers, given the program:

- n_1 = the number of distinct operators
- n_2 = the number of distinct operands
- N_1 = the total number of operators
- N_2 = the total number of operands

From these numbers, five measures can be calculated:

- Program length: $N = N_1 + N_2$
- Program vocabulary: $n = n_1 + n_2$
- Volume: $V = N \cdot log_2 n$
- Difficulty: $D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$
- Effort: $E = D \cdot V$

The difficulty measure D is related to the difficulty of the program to write or understand, e.g. when doing code review.

The volume measure V describes the size of the implementation of an algorithm.

McCabe's Cyclomatic Complexity

Description

Cyclomatic complexity (or conditional complexity) is a software metric used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code.

Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program.

The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. For instance, if the source code contains no decision points, such as `if` statements or `for` loops, the complexity would be 1, since there is only a single path through the code. If the code has a single `if` statement containing a single condition there would be two paths through the code, one path where the `if` statement is evaluated as `true` and one path where the `if` statement is evaluated as `false`.

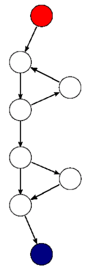
Calculation

Mathematically, the cyclomatic complexity of a structured program is defined with reference to a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second (the control flow graph of the program). The complexity is then defined as:

$M = E - N + 2P$

where

- M = cyclomatic complexity
- E = the number of edges of the graph
- N = the number of nodes of the graph
- P = the number of connected components.



Take, for example, a control flow graph of a simple program. The program begins executing at the red node, then enters a loop (group of three nodes immediately below the red node). On exiting the loop, there is a conditional statement (group below the loop), and finally the program exits at the blue node. For this graph, $E = 9$, $N = 8$ and $P = 1$, so the cyclomatic complexity of the program is 3.

```
begin
   for i in 1..3
   loop
      dbms_output.put_line('in loop');
   end loop;
   --
   if 1 = 1
   then
      dbms_output.put_line('yes');
   end if;
   --
   dbms_output.put_line('end');
end;
/
```

For a single program (or subroutine or method), P is always equal to 1. Cyclomatic complexity may, however, be applied to several such programs or subprograms at the same time (e.g., to all of the methods in a class), and in these cases P will be equal to the number of programs in question, as each subprogram will appear as a disconnected subset of the graph.

It can be shown that the cyclomatic complexity of any structured program with only one entrance point and one exit point is equal to the number of decision points (i.e., `if` statements or conditional loops) contained in that program plus one.

Cyclomatic complexity may be extended to a program with multiple exit points; in this case it is equal to:

\pi = s + 2

Where

- \pi is the number of decision points in the program, and
- s is the number of exit points.

Code Reviews

Code reviews check the results of software engineering. According to IEEE-Norm 729, a review is a more or less planned and structured analysis and evaluation process. Here we distinguish between code review and architect review.

To perform a code review means that after or during the development one or more reviewer proof-reads the code to find potential errors, potential areas for simplification, or test cases. A code review is a very good opportunity to save costs by fixing issues before the testing phase.

What can a code-review be good for?

- Code quality
- Code clarity and maintainability
- Quality of the overall architecture
- Quality of the documentation
- Quality of the interface specification

For an effective review, the following factors must be considered:

- Definition of clear goals.
- Choice of a suitable person with constructive critical faculties.
- Psychological aspects.
- Selection of the right review techniques.
- Support of the review process from the management.
- Existence of a culture of learning and process optimization.

Requirements for the reviewer:

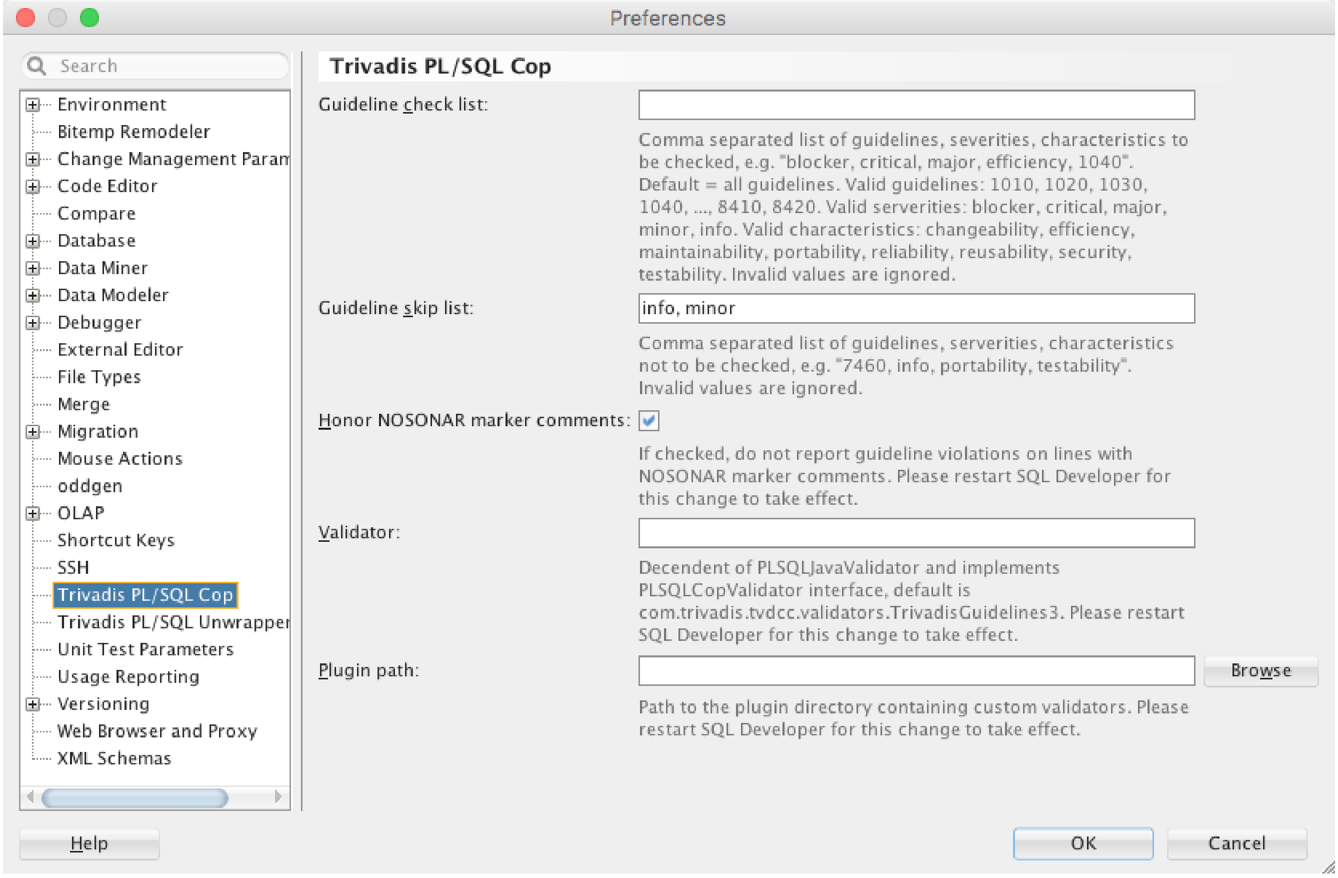
- He must not be the owner of the code.
- Code reviews may be unpleasant for the developer, as he could fear that his code will be criticized. If the critic is not considerate, the code writer will build up rejection and resistance against code reviews.

Tool Support

Development

Trivadis offers a cost-free extension to ORACLE SQL Developer to test compliance with this coding guideline. The extension may be parameterized to your preferred set of rules and allows checking this set against a program unit.

Setting the preferences



Trivadis PL/SQL Cop

Guideline check list:

Comma separated list of guidelines, severities, characteristics to be checked, e.g. "blocker, critical, major, efficiency, 1040". Default = all guidelines. Valid guidelines: 1010, 1020, 1030, 1040, ..., 8410, 8420. Valid serverities: blocker, critical, major, minor, info. Valid characteristics: changeability, efficiency, maintainability, portability, reliability, reusability, security, testability. Invalid values are ignored.

Guideline skip list: info, minor

Comma separated list of guidelines, serverities, characteristics not to be checked, e.g. "7460, info, portability, testability". Invalid values are ignored.

Honor NOSONAR marker comments: ☑

If checked, do not report guideline violations on lines with NOSONAR marker comments. Please restart SQL Developer for this change to take effect.

Validator:

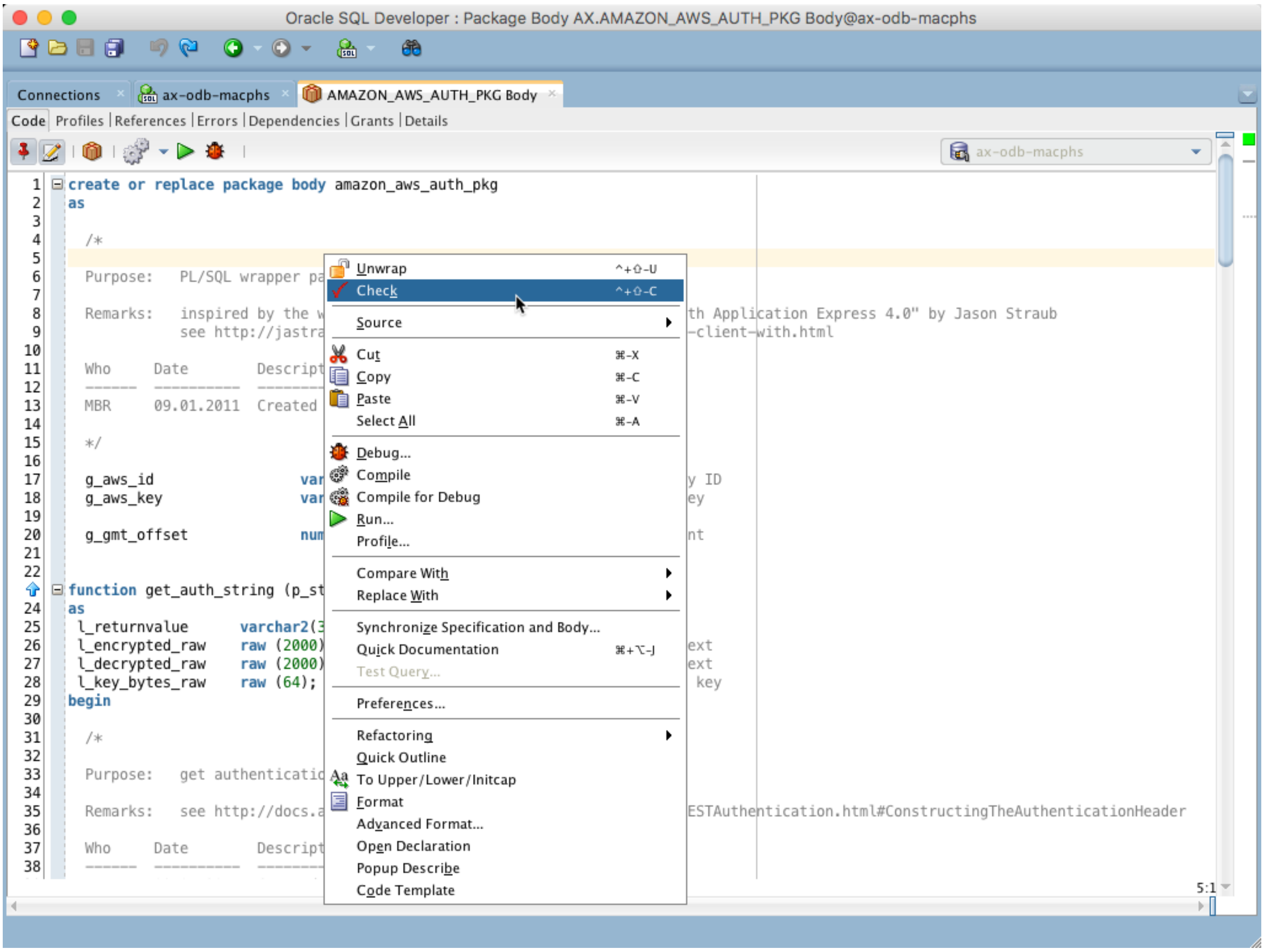
Decendent of PLSQLJavaValidator and implements PLSQLCopValidator interface, default is com.trivadis.tvdcc.validators.TrivadisGuidelines3. Please restart SQL Developer for this change to take effect.

Plugin path:

Path to the plugin directory containing custom validators. Please restart SQL Developer for this change to take effect.

There is an include list as well as an exclude list to define which rules to be checked or ignored.

Activate PLSQL Cop using context menu



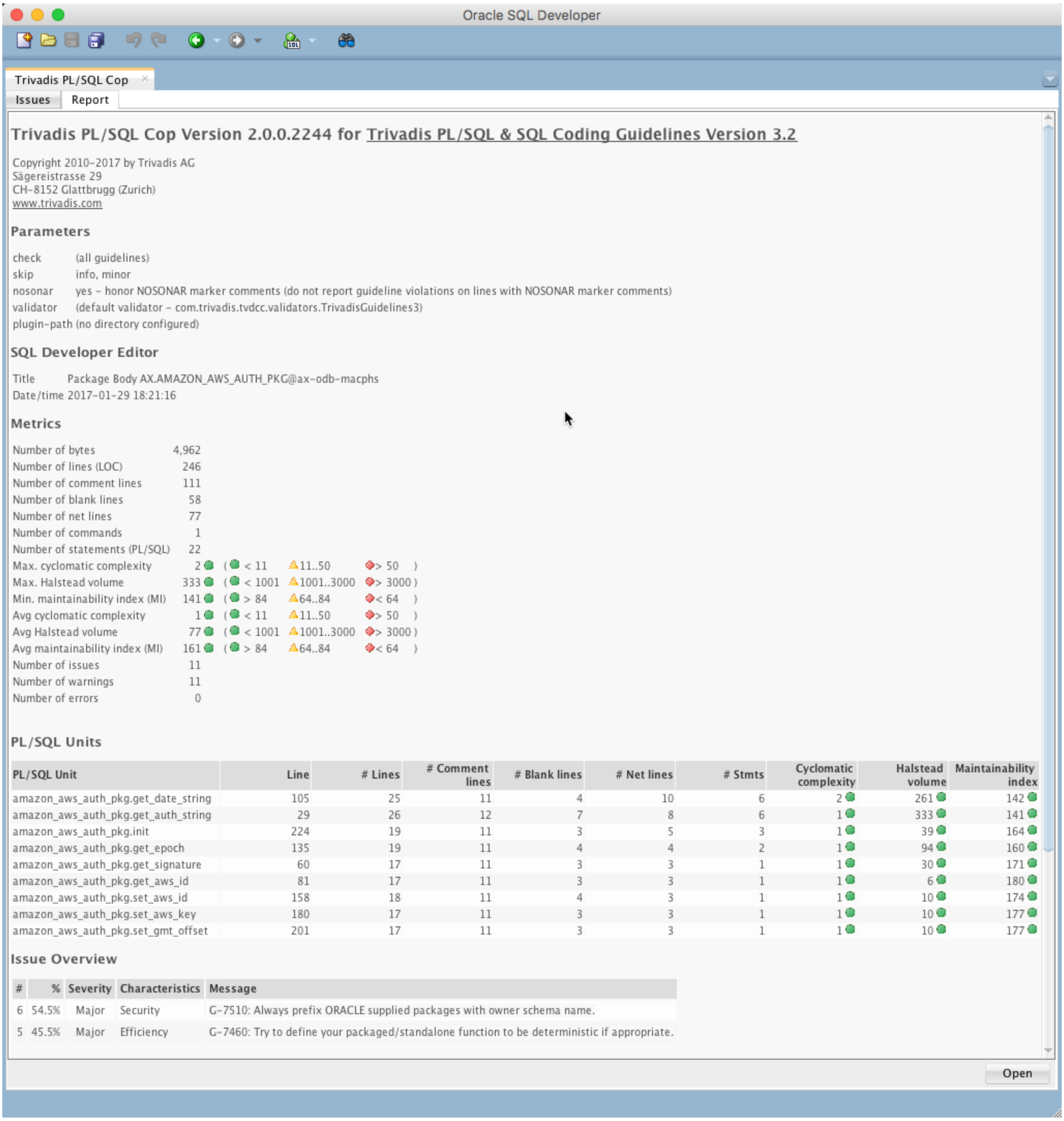
```
create or replace package body amazon_aws_auth_pkg
as

  /*

  Purpose:   PL/SQL wrapper pa

  Remarks:   inspired by the w                               th Application Express 4.0" by Jason Straub
             see http://jastra                               -client-with.html

  Who     Date        Descript
  ------  ----------  --------
  MBR     09.01.2011  Created

  */

  g_aws_id                 var                               y ID
  g_aws_key                var                               ey

  g_gmt_offset             num                               nt

function get_auth_string (p_st
as
 l_returnvalue      varchar2(3
 l_encrypted_raw    raw (2000)                               ext
 l_decrypted_raw    raw (2000)                               ext
 l_key_bytes_raw    raw (64);                                 key
begin

  /*

  Purpose:   get authenticatic

  Remarks:   see http://docs.a                               ESTAuthentication.html#ConstructingTheAuthenticationHeader

  Who     Date        Descript
  ------  ----------  --------
```

The result of the ckecking process is a list of violations with direct links to the place in the code as well as software metrics like:

- Cyclomatic complexity
- Halstead volume
- Maintainability Index
- Number of lines of code
- Number of comment lines
- Issue Overview

This statistics are gathered for each program unit in the reviewed code.

Software metrics



Trivadis PL/SQL Cop Version 2.0.0.2244 for Trivadis PL/SQL & SQL Coding Guidelines Version 3.2

Copyright 2010-2017 by Trivadis AG
Sägereistrasse 29
CH-8152 Glattbrugg (Zurich)
www.trivadis.com

Parameters

check	(all guidelines)
skip	info, minor
nosonar	yes - honor NOSONAR marker comments (do not report guideline violations on lines with NOSONAR marker comments)
validator	(default validator - com.trivadis.tvdcc.validators.TrivadisGuidelines3)
plugin-path	(no directory configured)

SQL Developer Editor

Title	Package Body AX.AMAZON_AWS_AUTH_PKG@ax-odb-macphs
Date/time	2017-01-29 18:21:16

Metrics

Number of bytes	4,962				
Number of lines (LOC)	246				
Number of comment lines	111				
Number of blank lines	58				
Number of net lines	77				
Number of commands	1				
Number of statements (PL/SQL)	22				
Max. cyclomatic complexity	2 🟢	(🟢 < 11	⚠11..50	♦> 50	)
Max. Halstead volume	333 🟢	(🟢 < 1001	⚠1001..3000	♦> 3000	)
Min. maintainability index (MI)	141 🟢	(🟢 > 84	⚠64..84	♦< 64	)
Avg cyclomatic complexity	1 🟢	(🟢 < 11	⚠11..50	♦> 50	)
Avg Halstead volume	77 🟢	(🟢 < 1001	⚠1001..3000	♦> 3000	)
Avg maintainability index (MI)	161 🟢	(🟢 > 84	⚠64..84	♦< 64	)
Number of issues	11				
Number of warnings	11				
Number of errors	0				

PL/SQL Units

PL/SQL Unit	Line	# Lines	# Comment lines	# Blank lines	# Net lines	# Stmts	Cyclomatic complexity	Halstead volume	Maintainability index
amazon_aws_auth_pkg.get_date_string	105	25	11	4	10	6	2 🟢	261 🟢	142 🟢
amazon_aws_auth_pkg.get_auth_string	29	26	12	7	8	6	1 🟢	333 🟢	141 🟢
amazon_aws_auth_pkg.init	224	19	11	3	5	3	1 🟢	39 🟢	164 🟢
amazon_aws_auth_pkg.get_epoch	135	19	11	4	4	2	1 🟢	94 🟢	160 🟢
amazon_aws_auth_pkg.get_signature	60	17	11	3	3	1	1 🟢	30 🟢	171 🟢
amazon_aws_auth_pkg.get_aws_id	81	17	11	3	3	1	1 🟢	6 🟢	180 🟢
amazon_aws_auth_pkg.set_aws_id	158	18	11	4	3	1	1 🟢	10 🟢	174 🟢
amazon_aws_auth_pkg.set_aws_key	180	17	11	3	3	1	1 🟢	10 🟢	177 🟢
amazon_aws_auth_pkg.set_gmt_offset	201	17	11	3	3	1	1 🟢	10 🟢	177 🟢

Issue Overview

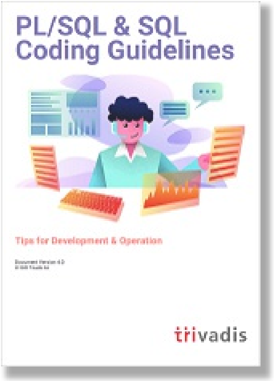
#	%	Severity	Characteristics	Message
6	54.5%	Major	Security	G-7510: Always prefix ORACLE supplied packages with owner schema name.
5	45.5%	Major	Efficiency	G-7460: Try to define your packaged/standalone function to be deterministic if appropriate.

Appendix

A - PL/SQL & SQL Coding Guidelines as PDF

These guidelines are primarily produced in HTML using Material for MkDocs.

However, we provide these guidelines also as PDF produced by wkhtmltopdf.



PL/SQL & SQL Coding Guidelines

Tips for Development & Operation

trivadis

The formatting is not perfect, but it should be adequate for those who want to work with offline documents.

B - Mapping new guidelines to prior versions

Old Id	New Id	Text	Severity	Change-ability	Efficiency	Maintain-ability	Portability
1	1010	Try to label your sub blocks.	Minor			X	
2	1020	Always have a matching loop or block label.	Minor			X	
3	1030	Avoid defining variables that are not used.	Minor		X	X	
4	1040	Avoid dead code.	Minor			X	
5	1050	Avoid using literals in your code.	Minor	X			
6	1060	Avoid storing ROWIDs or UROWIDs in database tables.	Major				
7	1070	Avoid nesting comment blocks.	Minor			X	
8	2110	Try to use anchored declarations for variables, constants and types.	Major			X	
9	2120	Try to have a single location to define your types.	Minor	X			
10	2130	Try to use subtypes for	Minor	X			

		constructs used often in your code.					
11	2140	Never initialize variables with NULL.	Minor			X	
12	2150	Avoid comparisons with NULL value, consider using IS [NOT] NULL.	Blocker				X
13	2160	Avoid initializing variables using functions in the declaration section.	Critical				
14	2170	Never overload variables.	Major				
15	2180	Never use quoted identifiers.	Major			X	
16	2185	Avoid using overly short names for explicitly or implicitly declared identifiers.	Minor			X	
17	2190	Avoid the use of ROWID or UROWID.	Major				X
18	2210	Avoid declaring NUMBER variables or subtypes with no precision.	Minor		X		
19	2220	Try to use PLS_INTEGER instead of NUMBER for arithmetic operations with integer values.	Minor		X		
n/a	2230	Try to use SIMPLE_INTEGER datatype when appropriate.	Minor		X		
20	2310	Avoid using CHAR data type.	Major				
21	2320	Avoid using VARCHAR data type.	Major				X
22	2330	Never use zero-length strings to substitute NULL.	Major				X
23	2340	Always define your VARCHAR2 variables using CHAR SEMANTIC (if not defined anchored).	Minor				
24	2410	Try to use boolean data type for values with dual meaning.	Minor			X	
25	2510	Avoid using the LONG and LONG RAW data types.	Major				X
26	3110	Always specify the target columns when coding an insert statement	Major			X	

		insert statement.					
27	3120	Always use table aliases when your SQL statement involves more than one source.	Major			X	
28	3130	Try to use ANSI SQL-92 join syntax.	Minor			X	X
29	3140	Try to use anchored records as targets for your cursors.	Major			X	
n/a	3150	Try to use identity columns for surrogate keys.	Minor			X	
n/a	3160	Avoid virtual columns to be visible.	Major			X	
n/a	3170	Always use DEFAULT ON NULL declarations to assign default values to table columns if you refuse to store NULL values.	Major				
n/a	3180	Always specify column names instead of positional references in ORDER BY clauses.	Major	X			
n/a	3190	Avoid using NATURAL JOIN.	Major	X			
30	3210	Always use BULK OPERATIONS (BULK COLLECT, FORALL) whenever you have to execute a DML statement more than 4 times.	Major		X		
31	4110	Always use %NOTFOUND instead of NOT %FOUND to check whether a cursor returned data.	Minor			X	
32	4120	Avoid using %NOTFOUND directly after the FETCH when working with BULK OPERATIONS and LIMIT clause.	Critical				
33	4130	Always close locally opened cursors.	Major		X		
34	4140	Avoid executing any statements between a SQL operation and the usage of an implicit cursor attribute.	Major				
35	4210	Try to use CASE rather than an IF statement with multiple	Major			X	

		ELSIF paths.					
36	4220	Try to use CASE rather than DECODE.	Minor			X	X
37	4230	Always use COALESCE instead of NVL, if parameter 2 of the NVL function is a function call or a SELECT statement.	Critical		X		
38	4240	Always use CASE instead of NVL2 if parameter 2 or 3 of NVL2 is either a function call or a SELECT statement.	Critical		X		
39	4310	Never use GOTO statements in your code.	Major			X	
40	4320	Always label your loops.	Minor			X	
41	4330	Always use a CURSOR FOR loop to process the complete cursor results unless you are using bulk operations.	Minor			X	
42	4340	Always use a NUMERIC FOR loop to process a dense array.	Minor			X	
43	4350	Always use 1 as lower and COUNT() as upper bound when looping through a dense array.	Major				
44	4360	Always use a WHILE loop to process a loose array.	Minor		X		
45	4370	Avoid using EXIT to stop loop processing unless you are in a basic loop.	Major			X	
46	4375	Always use EXIT WHEN instead of an IF statement to exit from a loop.	Minor			X	
47	4380	Try to label your EXIT WHEN statements.	Minor			X	
48	4385	Never use a cursor for loop to check whether a cursor returns data.	Major		X		
49	4390	Avoid use of unreferenced FOR loop indexes.	Major		X		
50	4395	Avoid hard-coded upper or lower bound values with FOR loops.	Minor	X		X	

n/a	5010	Try to use a error/logging framework for your application.	Critical				
51	5020	Never handle unnamed exceptions using the error number.	Critical			X	
52	5030	Never assign predefined exception names to user defined exceptions.	Blocker				
53	5040	Avoid use of WHEN OTHERS clause in an exception section without any other specific handlers.	Major				
54	n/a	Avoid use of EXCEPTION_INIT pragma for a 20nnn error.	Major				
55	5050	Avoid use of the RAISE_APPLICATION_ERROR built-in procedure with a hard-coded 20nnn error number or hard-coded message.	Major	X		X	
56	5060	Avoid unhandled exceptions	Major				
57	5070	Avoid using Oracle predefined exceptions	Critical				
58	6010	Always use a character variable to execute dynamic SQL.	Major			X	
59	6020	Try to use output bind arguments in the RETURNING INTO clause of dynamic DML statements rather than the USING clause.	Minor			X	
60	7110	Try to use named notation when calling program units.	Major	X		X	
61	7120	Always add the name of the program unit to its end keyword.	Minor			X	
62	7130	Always use parameters or pull in definitions rather than referencing external variables in a local program unit.	Major			X	
63	7140	Always ensure that locally defined procedures or functions are referenced.	Major			X	

64	7150	Try to remove unused parameters.	Minor		X	X	
65	7210	Try to keep your packages small. Include only few procedures and functions that are used in the same context.	Minor		X	X	
66	7220	Always use forward declaration for private functions and procedures.	Minor	X			
67	7230	Avoid declaring global variables public.	Major				
68	7240	Avoid using an IN OUT parameter as IN or OUT only.	Major		X	X	
69	7310	Avoid standalone procedures – put your procedures in packages.	Minor			X	
70	7320	Avoid using RETURN statements in a PROCEDURE.	Major			X	
71	7410	Avoid standalone functions – put your functions in packages.	Minor			X	
73	7420	Always make the RETURN statement the last statement of your function.	Major			X	
72	7430	Try to use no more than one RETURN statement within a function.	Major			X	
74	7440	Never use OUT parameters to return values from a function.	Major				
75	7450	Never return a NULL value from a BOOLEAN function.	Major				
n/a	7460	Try to define your packaged/standalone function to be deterministic if appropriate.	Major		X		
76	7510	Always prefix ORACLE supplied packages with owner schema name.	Major				
77	7710	Avoid cascading triggers.	Major			X	
n/a	7810	Do not use SQL inside PL/SQL to read sequence numbers (or SYSDATE)	Major		X	X	

78	8110	Never use SELECT COUNT(*) if you are only interested in the existence of a row.	Major		X		
n/a	8120	Never check existence of a row to decide whether to create it or not.	Major		X		
79	8210	Always use synonyms when accessing objects of another application schema.	Major	X		X	
n/a	8310	Always validate input parameter size by assigning the parameter to a size limited variable in the declaration section of program unit.	Minor			X	
n/a	8410	Always use application locks to ensure a program unit only running once at a given time.	Minor		X		
n/a	8510	Always use dbms_application_info to track program process transiently	Minor		X		

1. We see a table and a view as a collection. A jar containing beans is labeled "beans". In Java we call such a collection also "beans" (`List<Bean> beans`) and name an entry "bean" (`for (Bean bean : beans) {...}`). An entry of a table is a row (singular) and a table can contain an unbounded number of rows (plural). This and the fact that the Oracle database uses the same concept for their tables and views lead to the decision to use the plural to name a table or a view.
2. It used to be good practice to use uppercase keywords and lowercase names to help visualize code structure. But practically all editors support more or less advanced color highlighting of code, similar to the examples in these guidelines. Hence as of version 4.0 we are now recommending all lowercase, as this is easier and faster for the brain to process. You may choose to prefer the old rule - however, it is important to always be consistent, like for example keywords always in uppercase and names always in lowercase.
3. Tabs are not used because the indentation depends on the editor configuration. We want to ensure that the code looks the same, independent of the editor used. Hence, no tabs. But why not use 8 spaces? That's the traditional value for a tab. When writing a package function the code in the body has an indentation of 3. That's 24 characters as a starting point for the code. We think it's too much. Especially if we try to keep a line below 100 or 80 characters. Other good options would be 2 or 4 spaces. We settled for 3 spaces as a compromise. The indentation is still good visible, but does not use to much space.