# PL/SQL Styleguide

None

# Table of contents

1. Introduction to the Insum PL/So	QL and SQL Coding Guidelines	4
1.1 License		4
1.2 Revision History		5
2. Document Conventions		7
2.1 Scope		7
2.2 SQALE		7
3. Naming Conventions		11
3.1 General Guidelines		11
3.2 Naming Conventions for PL	/SQL	11
3.3 Database Object Naming Co	nventions	12
4. Coding Style		18
4.1 General Style		18
4.2 Comments Style		20
5. Language Usage		23
5.1 General		23
5.2 Variables & Samp; Types		32
5.3 DML & amp; SQL		50
5.4 Control Structures		62
5.5 Exception Handling		87
5.6 Dynamic SQL		95
5.7 Stored Objects		99
5.8 Patterns		127
6. Code Reviews		135

# 1. Introduction to the Insum PL/SQL and SQL Coding Guidelines

SQL and PL/SQL code is fundamentally some of the most important code that Insum writes for our customers and partners. The difference between SQL and PL/SQL that performs well and that doesn't can be the difference between a successful system (our customers and partners) and a huge disappointment (Healthcare.gov's rollout for example, not done by Insum...).

For a PDF version of these guidelines use Primus-PLSQL-and-SQL-Coding-Guidelines.pdf

This documentation is created upon a fork of the Insum-Styleguide which in turn has been a fork of the Trivadis guidelines. Currently we are on our way of adapting it to our companie's requirements.

### 1.0.1 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.

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 unless the existing standards have fundamental flaws that would decrease performance and/or significantly decrease the maintainability of code. Almost every system has a mixture of "code that follows the standards" and "code that doesn't follow the standards". Gentle migration over time to follow a good set of reasonable standards will always be much better than giving up because standards were not followed in the past.

Overall, the most important thing when writing good code is that you must be able to defend your work.

#### 1.1 License

The Primus Solutions PL/SQL & SQL Coding Guidelines are licensed under the Apache License, Version 2.0. You may obtain a copy of the License at <a href="http://www.apache.org/licenses/LICENSE-2.0">http://www.apache.org/licenses/LICENSE-2.0</a>.

#### 1.1.1 Trademarks

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

#### 1.1.2 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.

# 1.2 Revision History

Version	Who	Date	Comment
1.0	Stefan Knoll	2020.02.05	Forked from the Insum-Styleguide standards with many updates due to coding style and minor updates to grammar, removal of some sections, changes to titles of other sections, etc.

## 2. Document Conventions

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

## 2.1 Scope

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

## 2.2 SQALE

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.

## 2.2.1 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.

## 2.2.2 Severity of the rule



Will or may result in a bug.



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



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



Will have a low impact on the maintenance cost.



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

## 2.2.3 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.

- 9/135 - Primus Solutions AG 2024

- 10/135 - Primus Solutions AG 2024

# 3. Naming Conventions

## 3.1 General Guidelines

- 1. Never use names with a leading numeric character.
- 2. Always choose meaningful and specific names.
- 3. Avoid using abbreviations.
- 4. If abbreviations are used, they must be widely known and accepted.
- 5. Create a glossary with all accepted abbreviations.
- 6. Never use ORACLE keywords as names. A list of ORACLEs keywords may be found in the dictionary view vsreserved words.
- 7. Avoid adding redundant or meaningless prefixes and suffixes to identifiers. Example:  $CREATE TABLE emp\_table$ .
- 8. Always use one spoken language (e.g. English, German, French) for all objects in your application.
- 9. Always use the same names for elements with the same meaning.

## 3.2 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  $\{ {\tt prefix} \} {\tt variable} {\tt content} \{ {\tt suffix} \}$  pattern.

The following table shows a possible set of naming conventions.

Identifier	Prefix	Suffix	Example
Global Variable	g_		g_version
Local Variable	1_		1_version
Constants *	k_		k_employee_permanent
Record	r_		r_employee
Array / Table	t_		t_employee
Object	0_		o_employee
Cursor Parameter	p_		p_empno
In Parameter	p_   in_		p_empno
Out Parameter	x_   out_		$x_{ename}$
In/Out Parameter	x_   io_		io_employee
Record Type Definitions	r_	_type	r_employee_type
Array/Table Type Definitions	t_	_type	t_employee_type
Exception	e_		e_employee_exists
Subtypes		_type	big_string_type
Cursor		_cur	employee_cur

\* Why k\_ instead of c\_ for constants? A k is hard (straight lines, hard sound when pronouced in English) while a c is soft (curved lines and soft sound when pronounced in English). C also has the possibility of being vague (some folks use c\_ for cursors) and sounds changable... Also, very big companies (like Google in their coding standards) use k as a prefix for constants.

## 3.3 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.

Edition Based Redefinition (EBR) is one major exception to this guideline. When naming tables that will be covered by editioning views, it is preferable to name the covered table in lower case begining with an underscore (for example: "\_employee"). The base table will be covered by an editioning view that has the name employee. This greatly simplifies migration from non-EBR systems to EBR systems since all existing code already references data stored in employee. "Embracing the abomination of forced lower case names" highlights the fact that these objects shouldn't be directly referenced (execpt, obviously, by forward and reverse cross edition triggers during edition migration, and simple auditing/surrogate key triggers, if they are used). Since developers and users should only be referencing data through editioning views (which to them are effectively the tables of the applications) they won't be tempted to use the base table. In addition, when using tools to look at the list of tables, all editioning view covered tables will be aligned together and thus clearly delinated from non-covered tables.

#### 3.3.1 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:

- employee\_ct
- order ct

#### 3.3.2 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)

Add a useful comment to the database dictionary for every column.

## 3.3.3 DML / Instead of Trigger

Choose a naming convention that includes:

#### either

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

or

- 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

#### Examples:

- employee\_br\_iud
- order\_audit\_trg
- order\_journal\_trg

#### 3.3.4 Foreign Key Constraint

Table name followed by referenced table name followed by a fight and an optional number suffix. If working on a pre-12.2 database, then you will probably end up being forced into abbreviations due to short object name lengths in older databases.

#### Examples:

- employee\_department\_fk
- sct icmd ic fk1 -- Pre 12.2 database

#### 3.3.5 Function

Name is built from a verb followed by a noun in general. Nevertheless, it is not sensible to call a function <code>get\_...</code> 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.

#### 3.3.6 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 <u>lidx</u> as a suffix.

## 3.3.7 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

## 3.3.8 Package

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

Optionally prefixed by a project abbreviation.

#### Examples:

- $\bullet$   ${\tt employee\_api}$  API for the employee table
- logger Utilities including logging support
- constants Constants for use across a project
- types Types for use across a project

#### 3.3.9 Primary Key Constraint

Table name or table abbreviation followed by the suffix  $\prescript{pk}$ .

#### Examples:

- employee pk
- department\_pk
- · contract pk

#### 3.3.10 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

### 3.3.11 Sequence

Name is built from the table name 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:

- employee\_seq
- order\_number\_seq

## 3.3.12 Synonym

Synonyms should share the name with the object referenced in another schema.

### 3.3.13 System Trigger

Name of the event the trigger is based on.

- Activity done by the trigger
- Suffix \_trg

## Examples:

- ddl\_audit\_trg
- logon\_trg

#### 3.3.14 Table

Singular name of what is contained in the table.

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

Optionally prefixed by a project abbreviation.

#### Examples:

- employee
- department
- sct contract
- sct contract line
- sct incentive module

Reason: Singular names have the following advantages over plural names: 1. In general, tables represent entities. Entities are singular. This encourages the art of Entity-Relationship modeling. 2. If all table names are singular, then you don't have to know if a table has a single row or multiple rows before you use it. 3. Plural names can be vastly different from singular names. What is the plural of news? lotus? knife? cactus? nucleus? There are so many words that are difficult and nonstandard to pluralize that it can add significant work to a project to 'figure out the plurals'. 4. For non-native speakers of whatever language is being used for table names, point number 3 is magnified significantly. 5. Plurals add extra unnecessary length to table names. 6. By far the biggest reason: There is no value in going through all the work to plural a table name. SQL statements often deal with a single row from a table with multiple rows, so you can't make the argument that <code>employees</code> is better than <code>employees</code> 'because the SQL will read better'.

#### Example (bad):

```
1 well_bores
2 well_bore_completions
3 well_bore_completion_components
4 well_bore_studies
5 well_bore_study_results
6 wells
```

## Example (good):

```
1 well well_bore
3 well_bore_completion
4 well_bore_completion_component
5 well_bore_study_result
```

## 3.3.15 Surrogate Key Columns

Surrogate primary key columns should be the table name with an underscore and id appended. For example: employee\_id

Reason: Naming the surrogate primary key column the same name that it would have (at least 99% of the time) when used as a foreign key allows the use of the using clause in SQL which greatly increases readability and maintainability of SQL code. When each table has a surrogate primary key column named id, then select clauses that select multiple id columns will need aliases (more code, harder to read and maintain). Additionally, the id surrogate key column means that every join will be forced into the on syntax which is more error-prone and harder to read than the using clause.

#### Example (bad):

```
select e.id as employee_id

d.id as department_id

,e.last_name
d.name
from employee e
join department d on (e.department_id = d.id);
```

#### Example (good):

```
1 select e.employee_id
2 ,department_id
3 ,e.last_name
4 ,d.name
5 from employee e
6 join department d using (department_id);
```

### 3.3.16 Temporary Table (Global Temporary Table)

Naming as described for tables.

Ideally suffixed by gtt

Optionally prefixed by a project abbreviation.

#### Examples:

- employee gtt
- contract\_gtt

#### 3.3.17 Unique Key Constraint

Table name followed by the role of the unique key constraint, a uk and an optional number suffix, if necessary.

## Examples:

- employee\_name\_uk
- department\_deptno\_uk
- sct\_contract\_uk

## 3.3.18 View

Singular name of what is contained in the view.

Ideally, suffixed by an indicator identifying the object as a view like v or vw (mostly used, when a 1:1 view layer lies above the table layer, but *not* used for editioning views)

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

Optionally prefixed by a project abbreviation.

#### Examples:

- active order -- A view that selects only active orders from the order table
- $\bullet$   ${\tt order\_v}$  -- A view to the order table
- order -- An editioning view that covers the "\_order" base table

# 4. Coding Style

## 4.1 General Style

#### 4.1.1 Formatting

#### Rules

Rule	Description
1	All code is written in lowercase.
2	3 space indention.
3	One command per line.
4	Keywords loop, else, elseif, 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 employee.employee_id%type)
 3
        cursor c_employee(p_employee_id IN employee.employee_id%type) is
 4
           select last name
              , first_name
             , salary
from employee
       where employee_id = p_employee_id
order by last_name
 8
       r_employee c_employee%rowtype;
l_new_salary employee.salary%type;
11
12
13
        open c_employee(p_employee_id => in_employee_id);
14
       fetch c_employee into r_employee;
1.5
       close c_employee;
16
       new_salary (in_employee_id => in_employee_id
    , out_salary => l_new_salary);
17
18
19
       -- Check whether salary has changed
20
       if r_employee.salary <> l_new_salary then
        update employee
set salary = 1_new_salary
21
22
            where employee_id = in_employee_id;
23
       end if;
24
     end set salary;
25
26
27
```

## **Package Version Function**

28

When version control is not available, each package could have a  ${\tt package\_version}$  function that returns a varchar2.

Note: If you are using a version control system (like Git for example) to track all code changes and you feel that you'll be able to track everything below using your version control system, and everyone that might need to figure out 'what is happening', from all developers to purely operational DBAs, knows how to use the version control system to figure out the below, then you might consider the below redundant and 'extra work'. If so, feel free not implement this function.

#### PACKAGE SPEC

```
--This function returns the version number of the package using the following rules:
--1. If there is a major change that impacts multiple packages, increment the first digit, e.g. 03.05.09 -> 04.00.00
--2. If there is a change to the package spec, increment the first dot, e.g. 03.02.05 -> 03.03.00
--3. If there is a minor change, only to the package body, increment the last dot e.g. 03.02.05 -> 03.02.06
--4. If the function returns a value ending in WIP, then the package is actively being worked on by a developer.
function package_version return varchar2;
```

#### PACKAGE BODY

```
-- Increment the version number based upon the following rules
     -- 1. If there is a major change that impacts multiple packages, increment the first digit, e.g. 03.05.09 \rightarrow 04.00.00
 3 -- 2. If there is a change to the package spec, increment the first dot, e.g. 03.02.05 -> 03.03.00 4 -- 3. If there is a minor change, only to the package body, increment the last dot e.g. 03.02.05 -> 03.02.06
     -- 4. If a developer begins work on a package, increment the comment version and include the words 'IN PROGRESS' in
            the new version line. Increment the return value and add WIP to the return value. Example: return '01.00.01 WIP'
 6
            And then IMMEDIATELY push/commit & compile the package.
     -- As you are working on the package and make updates to lines, use the version number at the end of the line to indicate when -- the line was changed. Example: l_person := 'Bob'; -- 01.00.01 Bob is the new person, was Joe. -- 5. Once work is complete, remove 'IN PROGRESS' from the comment and remove WIP from the return value.
     -- 6. If your work crosses the boundary of a sprint, having WIP in the return value will indicate that the package should not be promoted.
    function package_version return varchar2
12
     begin
13
        -- 01.00.00 YYYY-MM-DD First & Last Name Initial Version
14
       -- 01.00.01 YYYY-MM-DD First & Last Name Fixed issue number 72 documented in Jira ticket 87: https://ourjiraurl.com/f?p=87
15
        return '01.00.01';
16 end package version;
18
```

Some notes on the above: We are computer scientists, we write dates as YYYY-MM-DD, not DD-MON-RR or MON-DD-YYYY or any other way.

If you are in the middle of an update, then the function would look like this:

## 4.2 Comments Style

#### 4.2.1 Commenting Goals

Code comments are there to help future readers of the code (there is a good chance that future reader is you... Any code that you wrote six months to a year ago might as well have been written by someone else) understand how to use the code (especially in PL/SQL package specs) and how to maintain the code (especially in PL/SQL package bodies).

#### 4.2.2 The JavaDoc Template

Use the JavaDoc style comments, as seen in the example below and read more here JavaDoc Template and JavaDoc for the Oracle Database a la DBDOC.

```
1  //**
2  * Description
3  *
4  *
5  * @example
6  *
7  * @issue
8  * @author
9  * @created
10  * @param
11  * @return
12  */
```

#### 4.2.3 Commenting Tags

Tag	Meaning	Example
example	Code snippet that shows how the procedure or function can be called.	
issue	Ticketing system issue or ticket that explains the code functionality	@issue IE-234
param	Description of a parameter.	<pre>@param in_string input string</pre>
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

#### 4.2.4 Generated Documentation

If you used the JavaDoc syntax then you can use plsql-md-doc to generate an easy to read document.

Alternatively, Oracle SQL Developer or PL/SQL Developer include documentation functionality based on a javadoc-like tagging.

### 4.2.5 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.

#### 4.2.6 Code Instrumentation

Code Instrumentation refers, among other things, to an ability to monitor, measure, and diagnose errors. In short, we'll call them debug messages or log messages.

By far, the best logging framework available is Logger from OraOpenSource.

Consider using logger calls **instead** of comments when the information will, explain the logic, help diagnose errors, and monitor execution flow.

## For example:

```
procedure verify_valid_auth
      l_scope logger_logs.scope%type := k_scope_prefix || 'verify_valid_auth';
       logger.log('BEGIN', l_scope);
      if is_token_expired then
  logger.log('Time to renew the expired token, and set headers.', l_scope);
  hubspot_auth;
 8
      logger.log('We have a good token, set headers.', 1_scope);
set_rest_headers;
end if;
10
11
12
13
      logger.log('END', l_scope);
14
15 exception
     when OTHERS then
16
      logger.log_error('Unhandled Exception', 1_scope);
raise;
17
raise;
end verify_valid_auth;
20
21
```

# 5. Language Usage

## 5.1 General

5.1.1 G-1010: Try to label your sub blocks.



Maintainability

#### Reason

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

### Example (bad)

```
1 begin
2 begin
3 null;
4 end;
5 begin
7 end;
8 end;
9 /
```

## 5.1.2 G-1020: Have a matching loop or block label.



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)

35

```
1
     declare
        i integer;
k_min_value constant integer := 1;
k_max_value constant integer := 10;
k_increment constant integer := 1;
 6 begin
         <<pre><<pre><<pre><<pre><<pre>data>>
        begin
null;
end;
 9
10
         <<pre><<pre><<pre>codess_data>>
11
         begin
12
        null;
end;
13
14
         i := k_min_value;
<<while_loop>>
while (i <= k_max_value)</pre>
15
16
17
         loop
  i := i + k_increment;
18
19
         end loop;
20
21
          <<basic_loop>>
         loop
exit basic_loop;
end loop;
23
24
25
         <<for_loop>>
for i in k_min_value..k_max_value
26
27
         loop
            sys.dbms_output.put_line(i);
28
         end loop;
29
30
31
33
34
```

## Example (good)

```
declare
2    i integer;
3    k_min_value constant integer := 1;
4    k_max_value constant integer := 10;
5    k_increment constant integer := 1;
6    begin
              <<pre><<pre><<pre>pare_data>>
             begin
null;
end prepare_data;
10
            <<pre><<pre><<pre>open
11
12
             null;
end process_data;
13
14
             i := k_min_value;
<<white_loop>>
while (i <= k_max_value)</pre>
15
16
17
            loop
    i := i + k_increment;
end loop while_loop;
18
19
20
21
              <<basic_loop>>
             loop
    exit basic_loop;
end loop basic_loop;
22
23
24
       <<for_loop>>
for i in k_min_value..k_max_value
loop
    sys.dbms_output.put_line(i);
end loop for_loop;
end;
end;
25
26
27
28
30 /
31
32
33
34
```

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



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 employee.last_name%type;
l_first_name employee.first_name%type;
k_department_id constant department_id%type := 10;
 5
              e_good exception;
         begin
           begin
   select e.last_name
   into l_last_name
   from employee e
   where e.department_id = k_department_id;
 9
10
        exception
11
        when no_data_found then null; -- handle_no_data_found; when too_many_rows then null; -- handle_too_many_rows;
12
13
         end my_proc;
14 end my_package;
15 /
16
17
```

```
create or replace package body my_package is
       procedure my_proc is

l_last_name employee.last_name%type;

k_department_id constant department.department_id%type := 10;
            e_good exception;
         begin
            into l_last_name
from employee e
 8
 9
            where e.department_id = k_department_id;
10
            raise e_good;
11
        exception
12
         when no_data_found then null; -- handle_no_data_found; when too_many_rows then null; -- handle_too_many_rows;
14
        end my_proc;
15 end my_package;
16 /
17
18
```

## 5.1.4 G-1040: Always avoid dead code.



Maintainability

#### Reason

38 39

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)

```
1 declare
        k_dept_purchasing constant departments.department_id%type := 30;
    begin
        if 2=3 then
       null; -- some dead code here end if;
 5
        null; -- some enabled code here
 8
 9
        <<my_loop>>
       loop
exit my_loop;
null; -- some dead code here
end loop my_loop;
10
11
12
13
14
        null; -- some other enabled code here
15
16
        when 1 = 1 and 'x' = 'y' then
null; -- some dead code here
17
18
19
             null; -- some further enabled code here
20
       end case;
21
       <<my_loop2>>
22
       for r_emp in (select last_name
23
                       from employee
where department_id = k_dept_purchasing
or commission_pct is not null
and 5=6)
24
25
26
                       -- "or commission_pct is not null" is dead code
27
28
        sys.dbms_output.put_line(r_emp.last_name);
end loop my_loop2;
29
30
31
        null; -- some dead code here
33 end;
34 /
35
36
37
```

## 5.1.5 G-1050: Avoid using literals in your code.



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.

#### Example (bad)

```
declare
       l_job employee.job_id%type;
       select e.job_id
       into l_job
from employee e
        where e.manager_id is null;
      if l_job = 'ad_pres' then
 9
       end if;
11 exception
      when no_data_found then
null; -- handle_no_data_found;
12
13
       when too_many_rows then
14
         null; -- handle_too_many_rows;
15 end;
16 /
17
18
```

```
create or replace package constants is
  k_president constant employee.job_id%type := 'ad_pres';
 3 end constants;
 4 /
 5
 6 declare
        l_job employee.job_id%type;
 8
        select e.job_id
        into 1_job
from employee e
where e.manager_id is null;
 9
       if l_job = constants.k_president then
13
14
       end if;
15 exception
        when no_data_found then
16
        null; -- handle_no_data_found;
when too_many_rows then
17
18
           null; -- handle_too_many_rows;
19 end;
20
```

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



#### Reason

It is an extremely dangerous practice to store ROWIDs 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)

## 5.1.7 G-1070: Avoid nesting comment blocks.



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)

```
1 begin
/* comment one -- nested comment two */
3 null;
4 -- comment three /* nested comment four */
null;
6 end;
7 /
```

```
1 begin
2 /* comment one, comment two */
3 null;
4 -- comment three, comment four
5 null;
6 end;
7
```

## 5.2 Variables & amp; Types

#### 5.2.1 General

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



Maintainability, Reliability

#### REASON

Changing the size of the database column last\_name in the employee table from varchar2 (20 char) to varchar2 (30 char) 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);

k_first_row constant integer := 1;

begin

select e.last_name
    into l_last_name
    from employee e

where rownum = k_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;

end my_package;

/ 4
```

```
create or replace package body my_package is

procedure my_proc is

l_last_name employee.last_name%type;

k_first_row constant integer := 1;

begin

select e.last_name
into l_last_name
from employee e

where rownum = k_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;

/

/

15
```

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



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)

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



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

Туре	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 (table_name_id) columns.

#### EXAMPLE (BAD)

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

/

create or replace package body my_package is
procedure my_proc is
1_note types.big_string_type;
begin
1_note := some_function();
end my_proc;
end my_package;
/

13
```

#### G-2140: Never initialize variables with NULL.



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;
//
```

#### G-2150: Never use comparisons with NULL values, use IS [NOT] NULL.



Portability, Reliability

#### REASON

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

#### EXAMPLE (BAD)

```
1 declare
2 l_value integer;
3 begin
4 if l_value is null then
5 null;
6 end if;
end;
7 /
```

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



Reliability

REASON

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

### EXAMPLE (BAD)

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

```
declare
    k_department_id constant integer := 100;
    k_unkown_name constant department_department_name%type := 'unknown';
    l_department_name department.department_name%type;
    begin
    <<ii>decinit>>
    begin
    l_department_name := department_api.name_by_id(in_id => k_department_id);
    exception
    when value_error then
    l_department_name := k_unkown_name;
    end init;
    sys.dbms_output.put_line(l_department_name);
    end;
    /
}
```

### G-2170: Never overload variables.



Reliability

REASON

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

### EXAMPLE (BAD)

```
1
      begin
<<main>>
           declare
               k_main constant user_objects.object_name%type := 'test_main';
k_sub constant user_objects.object_name%type := 'test_sub';
k_sep constant user_objects.object_name%type := ' - ';
l_variable user_objects.object_name%type := k_main;
 6
 8
               declare
             l_variable user_objects.object_name%type := k_sub;
beqin
10
              sys.dbms_output.put_line(l_variable || k_sep || main.l_variable); end sub;
12
13
           end main;
14 end;
15 /
16
17
```

```
1 begin
              <<main>>
           declare
   k_main constant user_objects.object_name%type := 'test_main';
   k_sub constant user_objects.object_name%type := 'test_sub';
   k_sep constant user_objects.object_name%type := ' - ';
   l_main_variable user_objects.object_name%type := k_main;
begin
  6
  8
                  <<sub>>
  9
             l_sub_variable user_objects.object_name%type := k_sub;
begin
10
11
                 sys.dbms_output.put_line(l_sub_variable || k_sep || l_main_variable);
end sub;
12
13
14 end;
15 /
16
17
```

## G-2180: Never use quoted identifiers.



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;
/
11
```

```
declare
lead of the content of
```

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



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)

```
1  declare
2    l_sal_comm    integer;
3    k_my_constant constant integer := 1;
4    e my_exception exception;
5    begin
6    l_sal_comm := k_my_constant;
exception
7    when e_my_exception then
8    null;
9    end;
10  /
11
```

## G-2190: Avoid using ROWID or UROWID.



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

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

# 5.2.2 Numeric Data Types

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



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  ${\tt PLS\_INTEGER}$  instead of  ${\tt NUMBER}$  :

- PLS\_INTEGER uses less memory
- ullet 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 is
    k_big_increase constant number(1,0) := 1;

function big_increase return number is
begin
    return k_big_increase;
end big_increase;
end big_increase;
end constants;
// page 1.
```

```
create or replace package body constants is
    k_big_increase constant pls_integer := 1;

function big_increase return pls_integer is
begin
    return k_big_increase;
end big_increase;
end constants;
//
```

## G-2230: Try to use SIMPLE\_INTEGER datatype when appropriate.



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

#### EXAMPLE (BAD)

```
create or replace package body constants is
    k_big_increase constant simple_integer := 1;

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

# 5.2.3 Character Data Types

# G-2310: Avoid using CHAR data type.



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)

```
1 create or replace package types
2 is
3 subtype description_type is char(200);
4 end types;
5 /
```

```
1 create or replace package types
2 is
3 subtype description_type is varchar2(200 char);
4 end types;
5 /
```

## G-2320: Avoid using VARCHAR data type.



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)

```
1 create or replace package types is
2    subtype description_type is varchar(200 char);
3  end types;
4 /
```

```
1    create or replace package types is
2        subtype description_type is varchar2(200 char);
3    end types;
4    /
```

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



Portability

## REASON

Today zero-length strings and  $_{\text{NULL}}$  are currently handled identical by ORACLE. There is no guarantee that this will still be the case in future releases, therefore if you mean  $_{\text{NULL}}$  use  $_{\text{NULL}}$ .

## EXAMPLE (BAD)

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

function null_string return varchar2 is
begin
    return k_null_string;
end null_string;
end constants;
//
```

```
create or replace package body constants is

function empty_string return varchar2 is
begin
return null;
end empty_string;
end constants;

//
```

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



## REASON

Changes to  ${\tt NLS\_LENGTH\_SEMANTICS}$  will only be picked up by your code after a recompilation.

In a multibyte environment a VARCHAR2 (50) definition may not necessarily hold 50 characters, when multibyte characters a part of the value that should be stored unless the definition was done using the char semantic.

Additionally, business users never say last names should be 50 bytes in length.

# EXAMPLE (BAD)

```
1 create or replace package types is
2 subtype description_type is varchar2(200 char);
3 end types;
4 /
```

# 5.2.4 Boolean Data Types

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



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
k_newfile constant pls_integer := 1000;
k_oldfile constant pls_integer := 500;
l_bigger pls_integer;
begin
if k_newfile < k_oldfile then
l_bigger := constants.k_numeric_true;
else
l_bigger := constants.k_numeric_false;
end if;
end;
/
12</pre>
```

### EXAMPLE (BETTER)

```
declare
k_newfile constant pls_integer := 1000;
k_oldfile constant pls_integer := 500;
l_bigger boolean;
begin
if k_newfile < k_oldfile then
l_bigger := true;
else
l_bigger := false;
end if;
end;
/
12</pre>
```

```
declare
k_newfile constant pls_integer := 1000;
k_oldfile constant pls_integer := 500;
l_bigger boolean;
begin
l_bigger := nvl(k_newfile < k_oldfile,false);
end;
//
8</pre>
```

# 5.2.5 Large Objects

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



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)

```
1 create or replace package example_package is
2 procedure do_something;
3 end example_package;
4 /
5
6 create or replace package body example_package is
g_long clob;
g_raw blob;
9 procedure do_something is
10 begin
11 null;
end do_something;
end de_xample_package;
/
14
15
```

# 5.3 DML & amp; SQL

## 5.3.1 General

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



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 department
values (department_seq.nextval
,'Support'
,100
,10);
```

## EXAMPLE (GOOD)

Note: The above good example assumes the use of an identity column for department\_id.

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



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 behaviors and results.

Also, note that even if you have a single table statement, it will almost always at some point in the future end up getting joined to another table, so you get bonus points if you use table aliases all the time.

EXAMPLE (BAD)

```
select last_name
first_name

department_name

from employee
    join department using (department_id)
where extract(month from hire_date) = extract(month from sysdate);
```

### EXAMPLE (BETTER)

## EXAMPLE (GOOD)

Using meaningful aliases improves the readability of your code. \\

## EXAMPLE SUBQUERY (BAD)

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

```
select last_name
from employee
where employee_id in (select employee_id
from job
where job_title like '%manager%');
```

## EXAMPLE SUBQUERY (GOOD)

If the <code>job</code> table has no <code>employee\_id</code> column this query will return an error due to the directive (given by adding the table alias to the column) to read the <code>employee\_id</code> column from the <code>job</code> table.

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



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);
```

```
select emp.employee_id
,emp.last_name
,emp.first_name
,dept.department_name
from employees emp
join departments dept using (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.



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)

```
cursor c_employee is
   select employee_id, first_name, last_name
              from employee;
       l_employee_id employee.employee_id%type;
l_first_name employee.first_name%type;
l_last_name employee.last_name%type;
begin 8
      open c_employee;

fetch c_employee into l_employee_id, l_first_name, l_last_name;

<<pre><<pre>cpen_c_employee>
9
      11
12
13
14
15
16 end;
17 /
18
19
```

```
1 declare
        cursor c_employee is
select employee_id, first_name, last_name
 3
               from employee;
    r_employee c_employee%rowtype;
begin
        open c_employee;
fetch c_employee into r_employee;
<<pre><<pre>c
        while c_employee%found
       loop
-- do something with the data
10
        -- do something with the dock
fetch c_employee into r_employee;
end loop process_employee;
12
13
        close c_employee;
14 end;
15 /
16
```

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



Maintainability, Reliability

RESTRICTION

## ORACLE 12c or higher

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 location (
location_id number(10) not null
location_name varchar2(60 char) not null
location_name varchar2(30 char) not null
location_name varchar2(30 char) not null
location_pk primary key (location_id)
location_pk primary key (location_id)
locate sequence location_seq start with 1 cache 20
locate or replace trigger location_bri
locate insert on location
location_for each row
location_id := location_seq.nextval;
location_id := location_s
```

```
create table location (
location_id number(10) generated by default on null as identity
,location_name varchar2(60 char) not null
city varchar2(30 char) not null
,constraint location_pk primary key (location_id))
/
```

### G-3160: Avoid visible virtual columns.



Maintainability, Reliability

RESTRICTION

### **ORACLE 12c**

REASON

In contrast to visible columns, invisible columns are not part of a record defined using <code>%rowtype</code> 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 employee
       add total_salary generated always as
            (salary + nvl(commission_pct,0) * salary)
    declare
 6
      r_employee employee%rowtype;
l_id employee.employee_id%type := 107;
 8
       r_employee := employee_api.employee_by_id(1_id);
r_employee.salary := r_employee.salary * constants.small_increase();
       update employee
set row = r_employee
12
13
         where employee_id = l_id;
14 end;
15 /
16
17 Error report -
     ORA-54017: UPDATE operation disallowed ON virtual COLUMNS
1.8
    ORA-06512: at line 9
19
2.0
21
```

```
alter table employee
add total_salary invisible generated always as

(salary + nvl(commission_pct,0) * salary)

/

declare
r_employee employee%rowtype;
k_id constant employee.employee_id%type := 107;
begin
r_employee := employee_api.employee_by_id(k_id);
r_employee.salary := r_employee.salary * constants.small_increase();

update employee
set row = r_employee
where employee_id = k_id;
end;

/

declare
r_employee = 107;
begin
r_employee := employee_by_id(k_id);
r_employee.salary := r_employee.salary * constants.small_increase();

/
declare
r_employee id = k_id;
end;
/
declare
```

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



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)

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



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)

```
1 select upper(first_name)
2 ,last_name
3 ,salary
4 ,hire_date
5 from employee
6 order by 4,1,3;
```

```
1 select upper(first_name) as first_name
2 ,last_name
3 ,salary
4 ,hire date
5 from employee
6 order by hire_date
7 ,first_name
7 ,salary;
```

## G-3190: Avoid using NATURAL JOIN.



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 (updated\_by, updated) will result in inappropriate join conditions.

EXAMPLE (BAD)

```
,last_name
,first_name
    from employee natural join department
 order by department_name , last_name;
, last_na
DEPARTMENT_NAME
7
                                   LAST_NAME
                                                             FIRST_NAME
8 Accounting
                                                             William
9 Executive
                                  De Haan
                                                             Lex
11
alter table department add updated date default on null sysdate;
    alter table employee add updated date default on null sysdate;
13
14 select department_name
     ,last_name
,first_name
15
16
     from employee natural join department
17
    order by department_name
18
            ,last_name;
19
20 No data found
21
22
23
```

### EXAMPLE (GOOD)

- 59/135 - Primus Solutions AG 2024

## G-3200: Avoid using an ON clause when a USING clause will work.



Maintainability

## REASON

An on clause requires more code than a using clause and presents a greater possibility for making errors. The using clause is easier to read and maintain.

Note that the using clause prevents the use of a table alias for the join column in any of the other clauses of the sql statement.

### EXAMPLE (BAD)

```
select department_id
dept.department_name
,emp.last_name
from employee emp join department dept using (department_id);
```

# 5.3.2 BULK Operations

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



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;
        k_increase constant employee.salary%type := 0.1;
k_department_id constant departments.department_id%type := 10;
 5 begin
        \verb|t_employee_ids| := employee_api.employee_ids_by_department(|
 6
                                id_in => k_department_id
 8
      <<pre><<pre><<pre>process employees>>
 9
        for i in 1..t_employee_ids.count()
        update employee
11
               set salary = salary + (salary * k_increase)
12
       where employee_id = t_employee_ids(i);
end loop process_employees;
13
14 end;
15 /
16
17
```

```
t_employee_ids employee_api.t_employee_ids_type;
 3
         k_increase constant employee.salary%type := 0.1;
k_department_id constant departments.department_id%type := 10;
        k increase
        t_employee_ids := employee_api.employee_ids_by_department(
                                id_in => k_department_id
 8
        <<pre><<pre>cess_employees>>
       forall i in 1..t_employee_ids.count()
update employee
 9
               set salary = salary + (salary * k_increase)
11
            where employee_id = t_employee_ids(i);
12
12 end;
14
15
```

# 5.4 Control Structures

## **5.4.1 CURSOR**

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



Maintainability

REASON

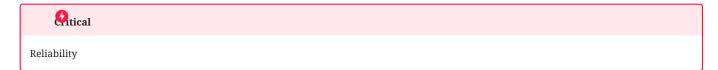
The readability of your code will be higher when you avoid negative sentences.

EXAMPLE (BAD)

```
declare
        cursor employee_cur is
          select last_name
            ,first_name
from employee
where commission_pct is not null;
 6
 7 r_employee employee_cur%rowtype;
8 begin
        open employee_cur;
10
         <<read_employees>>
12
       fetch employee_cur into r_employee;
exit read_employees when not employee_cur%found;
end loop read_employees;
1.3
14
15
         close employee_cur;
17 end;
18 /
19
20
```

```
declare
      cursor employee_cur is
         select last_name
          , first_name
from employee
where commission_pct is not null;
 5
 6
 7 r_employee employee_cur%rowtype; 8 begin
       open employee_cur;
10
       <<read_employees>>
12
       fetch employee_cur into r_employee;
13
           exit read_employees when employee_cur%notfound;
14
      end loop read_employees;
15
       close employee_cur;
16
17 end;
18 /
19
```

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



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

```
cursor employee_cur is
          select *
from employee
            order by employee_id;
 5
 6
        type t_employee_type is table of employee_cur%rowtype;
        t_employee t_employee_type;
k_bulk_size constant simple_integer := 10;
10
        open employee_cur;
11
12
        loop
fetch employee_cur bulk collect into t_employee limit k_bulk_size;
13
14
         exit process_employees when employee_cur%notfound;
15
           <<display_employees>>
16
         for i in 1..t_employee.count() loop
17
18
        sys.dbms_output.put_line(t_employee(i).last_name);
end loop display_employees;
end loop process_employees;
19
21
        close employee cur;
23 end;
24
25
26
27
```

#### EXAMPLE (BETTER)

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

```
1
     declare
         cursor employee_cur is select *
 2
              from employee
            order by employee_id;
        type t_employee_type is table of employee_cur%rowtype;
        t_employee t_employee_type;
k_bulk_size constant simple_integer := 10;
 8
 9 begin
        open employee_cur;
1.0
         <<pre><<pre><<pre>process employees>>
12
13
         fetch employee_cur bulk collect into t_employee limit k_bulk_size;
exit process_employees when t_employee.count() = 0;
14
          for i in 1..t_employee.count()
17
        sys.dbms_output.put_line(t_employee(i).last_name);
end loop display_employees;
end loop process_employees;
18
19
2.0
         close employee_cur;
21
22 end;
23
24
25
```

#### EXAMPLE (GOOD)

26

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

```
declare
 2
         cursor employee_cur is
           select *
from employee
order by employee_id;
 3
 6
         type t_employee_type is table of employee_cur%rowtype;
         t_employee t_employee_type;
k_bulk_size constant simple_integer := 10;
 8
 9 begin
10
         open employee_cur;
         fetch employee_cur bulk collect into t_employee limit k_bulk_size;
<<display_employees>>
for i in 1..t_employee.count()
loop
12
13
14
15
16
          sys.dbms_output.put_line(t_employee(i).last_name);
end loop display_employees;
exit process_employees when t_employee.count() <> k_bulk_size;
17
18
19
         end loop process_employees;
20
21
          close employee_cur;
22 end;
23
24
25
```

## G-4130: Always close locally opened cursors.



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 department.department_id%type)
           return number is
           cursor department_salary_cur(p_dept_id in department.department_id%type) is
             select sum(salary) as sum_salary
                from employee
               where department_id = p_dept_id;
           r_department_salary_department_salary_cur%rowtype;
 8
 9
         open department_salary_cur(p_dept_id => in_dept_id);
fetch department_salary_cur into r_department_salary;
10
11
           return r_department_salary.sum_salary;
12
       end department_salary;
13
    end employee_api;
14
15
16
```

```
1 create or replace package body employee_api as
       function department_salary (in_dept_id in department.department_id%type)
          return number is
          cursor department_salary_cur(p_dept_id in department.department_id%type) is
            select sum(salary) as sum_salary
               from employee
where department_id = p_dept_id;
 6
          r_department_salary_department_salary_cur%rowtype;
       begin
          open department salary cur(p dept id => in dept id);
          fetch department_salary_cur into r_department_salary;
close department_salary_cur;
           return r_department_salary.sum_salary;
12
       end department_salary;
13
    end employee_api;
14 /
1.5
16
```

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



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)

24

```
create or replace package body employee_api as
        k_one constant simple_integer := 1;
        procedure process_dept(in_dept_id in departments.department_id%type) is
          null;
 6
 8
        procedure remove_employee (in_employee_id in employee.employee_id%type) is
 9
           l_dept_id
                           employee.department_id%type;
10
        begin
         delete from employee
where employee_id = in_employee_id
11
12
           returning department_id into l_dept_id;
13
          process_dept(in_dept_id => l_dept id);
15
          if sql%rowcount > k_one then -- too many rows deleted.
16
17
18
          end if;
19
       end remove_employee;
20 end employee_api;
21 /
22
23
```

```
1    create or replace package body employee_api as
2    k_one constant simple_integer := 1;
        procedure process_dept(in_dept_id in departments.department_id%type) is
begin
  3
  4
  5
        null;
end process_dept;
  8
        10
        begin

delete from employee

where employee_id = in_employee_id

returning department_id into l_dept_id;
11
12
13
14
         1_deleted_emps := sql%rowcount;
15
16
          process_dept(in_dept_id => l_dept_id);
17
18
         if 1_deleted_emps > k_one then
    -- too many rows deleted.
    rollback;
19
20
21
          end if;
end remove_empl
end employee_api;
/
        end remove_employee;
25
26
27
```

# 5.4.2 CASE / IF / DECODE / NVL / NVL2 / COALESCE

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



Maintainability, Testability

REASON

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

EXAMPLE (BAD)

```
declare
l_color varchar2(7 char);
begin

if l_color = constants.k_red then
    my_package.do_red();
elsif l_color = constants.k_blue then
    my_package.do_blue();
elsif l_color = constants.k_black then
    my_package.do_black();
endif l_color = constants.k_black then
    my_package.do_black();
endif;
end;
/
12
```

```
declare

l_color types.color_code_type;

begin

case l_color

when constants.k_red then

my_package.do_red();

when constants.k_blue then

my_package.do_blue();

when constants.k_black then

my_package.do_black();

else null;

end case;

end;

13

14
```

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



Maintainability, Portability

## REASON

DECODE is an ORACLE specific function that can be hard to understand (particularly when not formatted well) and is restricted to SQL only. The CASE function is much more common has a better readability and may be used within PL/SQL too.

## EXAMPLE (BAD)

```
1 select decode(dummy, 'x', 1
2 , 'y', 2
3 , 'z', 3
4 , 0)
5 from dual;
```

```
1 select case dummy
2 when 'x' then 1
3 when 'y' then 2
4 when 'z' then 3
5 else 0
end
6 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.



Efficiency, Reliability

## REASON

The <code>nvl</code> 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)

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

```
1 select coalesce(dummy, my_package.expensive_null(value_in => dummy))
2 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.



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)

```
1 select case
2 when dummy is null then
3 my_package.expensive_null(value_in => dummy)
4 else
5 my_package.expensive_nn(value_in => dummy)
6 end
7 from dual;
```

## 5.4.3 Flow Control

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



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. Trying to use indentation to show the logical structure of a goto, however, is difficult or impossible.

Use of gotos is a matter of religion. In modern languages, you can easily replace nine out of ten gotos with equivalent structured constructs. In these simple cases, you should replace gotos out of habit. In the hard cases, you can break the code into smaller routines; 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 exercise.

EXAMPLE (BAD)

34 35

```
create or replace package body my_package is
       procedure password_check (in password in varchar2) is
   k_digitarray constant string(10 char) := '0123456789';
         l_password_length pls_integer;
 8
          1_array_length pls_integer;
9
         l_password_length := length(in_password);
10
                            := length(k_digitarray);
        <<check_digit>>
13
         for i in k_lower_bound .. l_array_length
14
         loop
      if substr(in_password, j, 1) = substr(k_digitarray, i, 1) then
l_isdigit := true;
goto check_other_things;
end if;
16
18
19
20
             end loop check pw char;
21
       end loop check_digit;
22
23
        <<check_other_things>>
24
          null;
25
26
        if not l_isdigit then
          raise_application_error(k_errno, k_errmsg);
end if;
27
28
29
    end my_package;
30
31
32
33
```

- 72/135 -

#### EXAMPLE (BETTER)

```
1 create or replace package body my_package is
        procedure password_check (in_password in varchar2) is
k_digitarray constant string(10 char) := '0123456789';
k_lower_bound constant simple_integer := 1;
k_errno constant simple_integer := -20501;
k_errmsg constant string(100 char) := 'Password must contain a digit.';
l_iadigit boolean := false;
 2
 3
 4
 5
             l_isdigit boolean := false;
l_password_length pls_integer;
              l_array_length pls_integer;
 9
         begin
            l_password_length := length(in_password);
l_array_length := length(k_digitarray);
10
11
12
             <<check_digit>>
13
            for i in k_{lower_bound} \dots l_{array_length}
14
           loop
                  <<check_pw_char>>
15
             <<check_pw_char>>
for j in k_lower_bound .. l_password_length
loop
16
17
                     if substr(in\_password, j, 1) = substr(k\_digitarray, i, 1) then
18
                        l isdigit := true;
19
                         exit check_digit; -- early exit condition
20
                    end if:
                end loop check_pw_char;
21
          end loop check_digit;
22
23
          <<check_other_things>>
24
             null;
25
26
           if not l_isdigit then
27
                raise_application_error(k_errno, k_errmsg);
             end if;
28
        end password_check;
29
     end my_package;
30
31
32
33
34
35
```

```
create or replace package body my_package is
procedure password_check (in_password in varchar2) is

k_digitpattern constant string(2 char) := '\d';

k_errno constant simple_integer := -20501;

k_errmsg constant string(100 char) := 'Password must contain a digit.';

begin

if not regexp_like(in_password, k_digitpattern)
then
    raise_application_error(k_errno, k_errmsg);
end if;
end password_check;
end my_package;
/

13
```

### G-4320: Always label your loops.



Maintainability

REASON

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

EXAMPLE (BAD)

```
1 declare
 declare
i integer;

k_min_value constant simple_integer := 1;

k_max_value constant simple_integer := 10;

k_increment constant simple_integer := 1;
 10
       end loop;
      loop
exit;
12
13
       end loop;
14
        for i in k_min_value..k_max_value
loop
15
16
        sys.dbms_output.put_line(i);
end loop;
17
18
19
        for r_employee in (select last_name from employee)
20
        loop
        sys.dbms_output.put_line(r_employee.last_name);
end loop;
21
22
23 end;
24
25
26
27
```

```
declare
i integer;
          k min_value constant simple_integer := 1;
k_max_value constant simple_integer := 10;
k_increment constant simple_integer := 1;
 4
5  k_increment constant simp
6  begin
7  <<while_loop>>
8  while (i <= k_max_value)
9  loop
10  i := i + k_increment;
11  end loop while_loop;</pre>
10
11
12
          <<br/>loop<br/>exit basic_loop;<br/>end loop basic_loop;
13
14
15
16
          <<for_loop>>
for i in k_min_value..k_max_value
loop
17
18
19
          sys.dbms_output.put_line(i);
end loop for_loop;
20
21
          22
23
24
          sys.dbms_output.put_line(r_employee.last_name);
end loop process_employees;
25
26
27 end;
28 /
29
30
31
32
```

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



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)

```
cursor employee_cur is
          select employee_id, last_name
from employee;
        r_employee employee_cur%rowtype;
 5
 6 begin
        open employee_cur;
 8
         <<output_employee_last_names>>
 9
       fetch employee_cur into r_employee;
exit read_employees when employee_cur%notfound;
10
        sys.dbms_output.put_line(r_employee.last_name);
end loop output_employee_last_names;
13
14
15 end;
16 /
18
```

```
declare
cursor employee_cur is
select employee_id, last_name
from employee;
begin
<coutput_employee_last_names>>
for r_employee in employee_cur
loop
sys.dbms_output.put_line(r_employee.last_name);
end loop output_employee_last_names;
end;
/
12
```

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



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)

```
1 declare
         type t_employee_type is varray(10) of employee.employee_id%type;
t_employee t_employee_type;
k_himuro constant integer := 118;
         k_livingston constant integer := 177;
k_min_value constant simple_integer := 1;
k_increment constant simple_integer := 1;
           i pls_integer;
         t_employee := t_employee_type(k_himuro, k_livingston);
i := k_min_value;
 9
10
11
            <<pre><<pre><<pre><<pre><<pre><<pre><<pre><<pre>
12
13
           exit process_employees when i > t_employee.count();
sys.dbms_output.put_line(t_employee(i));
i := i + k_increment;
14
15
           end loop process_employees;
17 end;
18
19
20
```

### EXAMPLE (GOOD)

```
declare
type t_employee_type is varray(10) of employee.employee_id%type;
t_employee t_employee_type;
k_himuro constant integer := 118;
k_livingston constant integer := 177;
begin
t_employee := t_employee_type(k_himuro, k_livingston);

for i in 1..t_employee.count()
loop
sys.dbms_output.put_line(t_employee(i));
end loop process_employees;
end;
//
14
```

- 77/135 -

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



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 employee.employee_id%type;

t_employee t_employee_type := t_employee_type();

begin

c
cyprocess_employees>>
for i in t_employee.first()..t_employee.last()
loop
sys.dbms_output.put_line(t_employee(i)); -- some processing
end loop process_employees;
end;

/

/

/
```

EXAMPLE (BETTER)

Raise an unitialized collection error if  $t_{\tt employee}$  is not initialized.

```
declare
type t_employee_type is table of employee.employee_id%type;
t_employee t_employee_type := t_employee_type();
begin
```

EXAMPLE (GOOD)

Raises neither an error nor checking whether the array is empty. t\_employee.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 employee.employee_id%type;
t_employee t_employee_type := t_employee_type();

begin
if t_employee is not null then

for i in 1..t_employee.count()
    loop
    sys.dbms_output.put_line(t_employee(i)); -- some processing
end loop process_employees;
end if;
end;
/
13
```

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



Efficiency

### REASON

When a loose 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 employee.employee_id%type;
              t_employee t_employee_type;
k_rogers constant integer := 134;
                                        constant integer := 143;
                                         constant integer := 158;
              k_index_matos constant integer := 2;
             t_employee := t_employee_type(k_rogers, k_matos, k_mcewen);
t_employee.delete(k_index_matos);
10
            if t_employee is not null then
11
                   <<pre><<pre><<pre>cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
12
                    for i in 1..t_employee.count()
1.3
                   loop
14
                          sys.dbms_output.put_line(t_employee(i));
15
                   end loop process_employees;
              end if;
16
17 end;
18
```

```
declare
       type t_employee_type is table of employee.employee_id%type;
       t_employee type;
k_rogers constant integer := 134;
 3
                     constant integer := 143;
       k mcewen
                     constant integer := 158;
 6
       k index matos constant integer := 2;
                   pls_integer;
       t_employee := t_employee_type(k_rogers, k_matos, k_mcewen);
       t_employee.delete(k_index_matos);
10
       l_index := t_employee.first();
12
13
       <<pre><<pre><<pre>process employees>>
14
       while l_index is not null
15
       loop
          sys.dbms_output.put_line(t_employee(l_index));
16
          1_index := t_employee.next(l_index);
17
       end loop process_employees;
18
21
22
```

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



Maintainability

### REASON

34 35

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)

```
1 declare
      i integer;
k_min_value constant simple_integer := 1;
 k_max_value constant simple_integer := 10;
       i := k_min_value;
       <<while_loop>>
while (i <= k_max_value)</pre>
 8
      i := i + k_increment;
exit while_loop when i > k_max_value;
end loop while_loop;
 9
10
11
12
13
        <<basic_loop>>
14
      loop
exit basic_loop;
hasic loop
15
       end loop basic_loop;
16
17
18
       for i in k_min_value..k_max_value loop
19
20
       exit for_loop when i = k_max_value;
end loop for_loop;
21
22
23
        <<pre><<pre><<pre>process_employees>>
for r_employee in (select last_name)
24
25
                               from employee)
26
        sys.dbms_output.put_line(r_employee.last_name);
null; -- some processing
27
28
            exit process_employees;
29
        end loop process_employees;
30 end;
31
32
33
```

```
1 declare
 2
       i integer;
          k min_value constant simple_integer := 1;
k_max_value constant simple_integer := 10;
k_increment constant simple_integer := 1;
 3
 4
5  k_increment constant simp
6  begin
7  <<while_loop>>
8  while (i <= k_max_value)
9  loop
10  i := i + k_increment;
11  end loop while_loop;</pre>
10
11
12
        <<basic_loop>>
loop
   exit basic_loop;
end loop basic_loop;
13
14
15
16
          <<for_loop>>
for i in k_min_value..k_max_value
loop
17
18
19
         sys.dbms_output.put_line(i);
end loop for_loop;
20
21
        22
23
24
         sys.dbms_output_line(r_employee.last_name); -- some processing
end loop process_employees;
25
26
27 end;
28 /
29
30
31
32
```

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



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)

### G-4380 Try to label your EXIT WHEN statements.



Maintainability

REASON

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

EXAMPLE (BAD)

```
declare
      k_init_loop constant simple_integer
       k_increment constant simple_integer
                                                   := 1;
      6
      l_outerloop pls_integer;
l_innerloop pls_integer;
 8 begin
      l_outerloop := k_init_loop;
       <<outerloop>>
10
       l_innerloop := k_init_loop;
l_outerloop := nvl(l_outerloop,k_init_loop) + k_increment;
12
13
          <<innerloop>>
14
        loop
        l_innerloop := nvl(l_innerloop, k_init_loop) + k_increment;
sys.dbms_output.put_line(k_outer_text || l_outerloop ||
15
16
                                     k_inner_text || l_innerloop);
17
18
             exit when l_innerloop = k_exit_value;
19
        end loop innerloop;
21
          exit when l_innerloop = k_exit_value;
       end loop outerloop;
22
    end;
23
24
2.5
26
27
```

```
k_init_loop constant simple_integer
k_increment constant simple_integer
k_exit_value constant simple_integer
                                                          := 1;
:= 3;
        k_outer_text constant types.short_text_type := 'outer loop counter is ';
k_inner_text constant types.short_text_type := ' inner loop counter is ';
        l_outerloop pls_integer;
l_innerloop pls_integer;
 8 begin
 9
        l_outerloop := k_init_loop;
10
         <<outerloop>>
        loop
        l_innerloop := k_init_loop;
l_outerloop := nvl(l_outerloop,k_init_loop) + k_increment;
12
13
         14
15
16
17
18
                exit outerloop when l_innerloop = k_exit_value;
19
            end loop innerloop;
20
        end loop outerloop;
21 end;
22 /
23
24
```

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



Efficiency

REASON

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

Also, check out rule G-8110: Never use SELECT COUNT(\*) if you are only interested in the existence of a row.

### EXAMPLE (BAD)

```
declare

l_employee_found boolean := false;

cursor employee_cur is

select employee id, last_name

from employee;

r_employee employee_cur%rowtype;

begin

<check_employees>>

for r_employee in employee_cur

loop

l_employee_found := true;

end loop check_employees;

end;

//

14
```

```
declare
l_employee_found boolean := false;
cursor employee_cur is
select employee_id, last_name
from employee;
r_employee employee_cur%rowtype;
begin
open employee_cur;
fetch employee_cur into r_employee;
l_employee_found := employee_cur%found;
close employee_cur;
end;
/
13
```

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



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)

```
1 declare
          l_row pls_integer;
l_value pls_integer;
k_lower_bound constant simple_integer
                                                                                              := 1;
             k_upper_bound constant simple_integer
          k_row_incr constant simple_integer
k_value_incr constant simple_integer
k_delimiter constant types.short_text_type
k_first_value constant simple_integer
                                                                                              := 1;
  6
                                                                                              := 10;
  9 begin
         1_row := k_lower_bound;
1_value := k_first_value;
12
            for i in k\_lower\_bound .. k\_upper\_bound
13
           sys.dbms_output.put_line(l_row || k_delimiter || l_value);
l_row := l_row + k_row_incr;
l_value := l_value + k_value_incr;
end loop for_loop;
14
15
16
end;
17
19
20
21
```

```
declare
        k_lower_bound constant simple_integer
                                                                           := 1;
     k_upper_bound constant simple_integer
k_value_incr constant simple_integer
k_delimiter constant types.short_text_type
k_first_value constant simple_integer
begin
 5
                                                                           := 100;
 6
          <<for_loop>>
 8
         for i in k_lower_bound .. k_upper_bound
              sys.dbms_output.put_line(i || k_delimiter ||
                                                to_char(k_first_value + i * k_value_incr));
12
     end;
13
14
```

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



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

coutput_loop>
for i in 1..5

doop

sys.dbms_output.put_line(i);
end loop output_loop;
end;
//
8
```

# 5.5 Exception Handling

5.5.1 G-5010: Always use an error/logging framework for your application.



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

By far, the best logging framework available is Logger from OraOpenSource.

## Example (bad)

```
begin

sys.dbms_output.put_line('start');

-- some processing

sys.dbms_output.put_line('end');
end;

end;

/
```

```
declare
2    -- see https://github.com/oraopensource/logger
3    l_scope logger_logs.scope%type := 'demo';
begin
    logger.log('start', 1_scope);
    -- some processing
1    logger.log('end', 1_scope);
end;
8    /
9
```

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



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
    k_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 = k_no_data_found then
        null;
end if;
end;
//
```

```
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
```

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



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)

26

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;
k rownum
                            constant simple_integer
         k rownum
         k_no_data_found constant types.short_text_type := 'no_data_found';
          into l_dummy
           from dual
         where rownum = k_rownum;
        if I dummy is null then
        raise no_data_found;
end if;
12
13
     exception
14
       when no_data_found then
15
            sys.dbms_output.put_line(k_no_data_found);
16 end;
17
18 Error report -
19
19 ORA-01403: no data found 20 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.
23
24
25
```

```
declare
      select dummy
into 1_dummy
from dual
where rownum = k_rownum;
10
11
     if l_dummy is null then
12
     raise empty_value;
end if;
13
14 exception
15
      when empty_value then
     sys.dbms_output.put_line(k_empty_value);
when no_data_found then
when no_data_found then sys.dbms_output.put_line(k_no_data_found); end; / 20
21
22
```

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



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.

## Example (bad)

5.5.5 G-5050: Avoid use of the RAISE\_APPLICATION\_ERROR built-in procedure with a hard-coded 20nnn error number or hard-coded message.



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 <code>raise\_application\_error</code>, 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 <code>raise\_application\_error</code>. If you are raising a "system" exception like <code>no\_data\_found</code>, you must use RAISE. However, when you want to raise an application-specific error, you use <code>raise\_application\_error</code>. 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)

```
1 begin
2 raise_application_error(-20501,'invalid employee_id');
3 end;
4 /
```

```
begin
errors.raise(in_error => errors.k_invalid_employee_id);
end;
//
```

## 5.5.6 G-5060: Avoid unhandled exceptions.



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 department.department_id%type)
         return department.department_name%type is
          1 department name department.department name%type;
5
         select department_name
           into l_department_name
           from department
          where department id = in id;
         return l_department_name;
      end name by id;
11
   end department_api;
12
13
14
```

```
create or replace package body department_api is
       function name_by_id (in_id in department.department_id%type)
  return department.department_name%type is
           1\_department\_name \ department.department\_name % type;
       begin
             into l_department_name
             from department
 8
           where department_id = in_id;
           return 1 department name;
1.0
         when no_data_found then return null;
12
           when too_many_rows then raise;
13
       end name_by_id;
14 end department_api;
17
```

## 5.5.7 G-5070: Avoid using Oracle predefined exceptions.



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)

```
1 declare
2 my_exception exception;
3 begin 4 raise my_exception;
5 end;
6 /
```

# 5.6 Dynamic SQL

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



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)

```
procedure trx_to_collection(
   p_appendix_id in px_mandate_appendix.id%TYPE
        k_trx_collection constant varchar2(10) := 'TRX_LINES';
 5
 6
       l_param_names
l_param_values
                                  apex_application_global.vc_arr2;
apex_application_global.vc_arr2;
    begin
       l_param_names(l_param_names.count + 1) := 'APPENDIX_ID';
l_param_values(l_param_names.count) := p_appendix_id;
10
        apex_collection.create_collection_from_query_b
12
13
            p_collection_name => k_trx_collection
        , p_query => q'[select t.id, 'Y' include_flag, 'TRX' type
14
15
                      from px_billing_transactions t
16
                     17
19
       , p_values
20
21 end;
22
23
24
```

## Example (good)

28

```
procedure trx_to_collection(
   p_appendix_id in px_mandate_appendix.id%TYPE
  3 )
  4 is
            k_trx_collection constant varchar2(10) := 'TRX_LINES';
  5
  6
            k_sql constant types.big_string_type :=
    q'[select t.id, 'Y' include_flag, 'TRX' type
    from px_billing_transactions t
    where t.appendix_id = :APPENDIX_ID
    and t.pending_invoice_flag = 'Y']';
10
11
                                                         apex_application_global.vc_arr2;
apex_application_global.vc_arr2;
          l_param_names
l_param_values
12
13
       begin
         l_param_names(l_param_names.count + 1) := 'APPENDIX_ID';
l_param_values(l_param_names.count) := p_appendix_id;
14
15
16
            apex_collection.create_collection_from_query_b
17
         apex_collection.create_collection_rrom_que
(
    p_collection_name => k_trx_collection
, p_query => k_sql
, p_names => l_param_names
, p_values => l_param_values
);
18
19
20
21
22
23
24 end;
25
26
27
```

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



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 employee.employee_id%type ,in_increase_pct in types.percentage
                                   ,out_new_salary out employee.salary%type)
            k_sql_stmt constant types.big_string_type :=
                  'update employee set salary = salary + (salary / 100 * :1) where employee_id = :2
 8
              returning salary into :3';
 9
        begin
         execute immediate k_sql_stmt
10
        using in_increase_pct, in_employee_id, out out_new_salary; end upd salary;
11
12
     end employee_api;
13
14
```

### Example (good)

```
create or replace package body employee_api is
procedure upd_salary (in_employee id in employee.employee_id*type
,in_increase_pct in types.percentage
,out_new_salary out employee.salary*type)
              k\_sql\_stmt constant types.big_string_type :=
 6
                      'update employee set salary = salary + (salary / 100 * :1) where employee_id = :2
 8
                 returning salary into :3';
          begin
          execute immediate k_sql_stmt
                     using in_increase_pct, in_employee_id returning into out_new_salary;
12
          end upd_salary;
13
     end employee_api;
14
1.5
16
```

- 97/135 -

## 5.6.3 G-6030: Always verify parameters that will be used in dynamic SQL with DBMS\_ASSERT



Maintainability, Testability

#### Reason

Parameters used with dynamic sql are subject to SQL injection. The DBMS\_ASSERT package provides an interface to validate properties of parameters.

### Example (bad)

```
procedure schedule_refresh_data_job (p_user varchar2) is
 2
        1 job action varchar2(4000);
 3
        1_scope logger_logs.scope%type := gc_scope_prefix || 'Create refresh job';
       logger.log('START',l_scope);
        l_job_action := 'begin hr.hr_utils.refresh_read_only_data_from_source; end;';
       dbmm_scheduler.create_job
  (job_name => gk_on_demand_job_name
  ,job_type => 'PLSQL_BLOCK'
          ,job_action => l_job_action
,start_date => sysdate
10
          ,enabled => true
          ,auto_drop => true
12
          ,job_class => 'DEFAULT_JOB_CLASS'
13
           ,comments => 'Refresh data from Source to Destination by ' || p_user || ' created on ' || to_char(sysdate ,'mm/dd/yyyy hh:mi:ss am')
14
15
       logger.log('END '.l scope);
    exception
       when others then
18
            logger.log_error('Unable to execute: '|| sqlerrm,1_scope);
19
               raise_application_error(-20001,'Unable to create refresh job :'||sqlerrm);
20 end schedule_refresh_data_job;
21
22
```

### Example (good)

```
procedure schedule_refresh_data_job (p_user varchar2) is
     l_job_action varchar2(4000);
      1_scope logger_logs.scope%type := gc_scope_prefix || 'Create refresh job';
      logger.log('START',l_scope);
      l_job_action := 'begin hr.hr_utils.refresh_read_only_data_from_source; end;';
      dbms_scheduler.create_job
       (job_name => gk_on_demand_job_name
,job_type => 'PLSQL_BLOCK'
 8
        ,job_action => l_job_action
,start_date => sysdate
 9
10
11
        ,auto_drop => true
,job_class => 'DEFAULT_JOB_CLASS'
         14 hh24:mi:ss')
15
16
      logger.log('END ',l_scope);
17
18
19
            logger.log_error('Unable to execute: '|| sqlerrm,l_scope);
20
            raise_application_error(-20001,'Unable to create refresh job :'||sqlerrm);
21 end schedule_refresh_data_job;
```

- 98/135 - Primus Solutions AG 2024

# 5.7 Stored Objects

## 5.7.1 General

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



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

```
declare
    r_employee employee%rowtype;
    k_id constant employee.employee_id%type := 107;

begin
    employee_api.employee_by_id(out_row => r_employee, in_employee_id => k_id);
end;
//
```

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



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 employee.employee_id%type)
return employee%rowtype is
 3
         \label{eq:r_employee} $$r_{employee} employee%rowtype;$$ begin
 5
            select *
 6
             into r_employee
from employee
where employee_id = in_employee_id;
 8
             return r_employee;
10
        exception
           when no_data_found then
12
                 null;
13
        when too_many_rows then raise; end;
14
15
16 end;
17 /
18
19
```

```
create or replace package body employee_api is
function employee_by_id (in_employee_id in employee.employee_id%type)
return employee%rowtype is
        begin
  5
              select *
  6
              into r_employee
from employee
where employee_id = in_employee_id;
  8
  9
              return r_employee;
         exception
11
         when no_data_found then
null;
when too_many_rows then
12
13
14
raise;
15 end employee_by_id;
16 end employee_api;
17 /
18
19
```

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



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 employee.employee_id%type) is
   r_employee employee%rowtype;
          function commission return number is
              1_commission employee.salary%type := 0;
          begin
             if r employee.commission pct is not null
 8
                1_commission := r_employee.salary * r_employee.commission_pct;
10
11
              return 1 commission;
12
          end commission;
13
       begin
14
          select *
15
           into r_employee
            from employe
           where employee_id = in_employee_id;
17
18
          sys.dbms_output.put_line(r_employee.salary + commission());
19
20
          when no data found then
21
             null;
          when too_many_rows then
22
             null:
23
       end calc_salary;
24
    end employee_api;
25
27
28
```

### EXAMPLE (GOOD)

```
create or replace package body employee_api is
procedure calc_salary (in_employee_id in employee.employee_id%type) is
r_employee employee%rowtype;
 4
        begin
   if in_comm_pct is not null then
        l_commission := in_salary * in_comm_pct;
 9
10
           end if;
11
12
        return 1_commission;
end commission;
13
14
      begin
       begin
  select *
  into r_employee
  from employee
  where employee_id = in_employee_id;
15
16
17
18
        19
20
21
22
     exception
when no_data_found then
null;
23
24
             null;
25
        when too_many_rows then
      null;
end calc_salary;
26
27
28 end employee_api;
29 /
30
31
32
```

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



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

k_true constant integer := 1;

begin

return k_true;
end my_func;

begin

null;
end my_procedure;
end my_package;

end my_package;

/

12
```

```
create or replace package body my_package is
procedure my_procedure is

function my_func return number is

k_true constant integer := 1;
begin
 return k_true;
end my_func;
begin

sys.dbms_output.put_line(my_func());
end my_procedure;
end my_package;
/
12
```

### G-7150: Try to remove unused parameters.



Efficiency, Maintainability

REASON

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

EXAMPLE (BAD)

```
begin
 6
        <<find_department>>
       begin
        select department_name
  into l_department_name
  from department
  where department_id = in_department_id;
 8
10
    12
13
14
15
        return l_department_name;
16
     end name_by_id;
17
end department_api;
19
20
```

```
create or replace package body department_api is
  function name_by_id (in_department_id in department.department_id%type)
    return department.department_name%type is
 3
             1_department_name department.department_name%type;
 4
        begin <<find_department>>
 5
 6
           begin
              select department_name
              into l_department_name
from department
where department_id = in_department_id;
10
        exception

when no_data_found or too_many_rows then
11
12
                    1_department_name := null;
1.3
          end find_department;
14
15
           return l_department_name;
        end name_by_id;
16
end department_api;
18
19
20
```

# 5.7.2 Packages

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



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.



Changeability

REASON

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

### EXAMPLE (BAD)

```
1 create or replace package department_api is 2 procedure del (in_department_id in department_id@type);
 3 end department_api;
 create or replace package body department_api is
function does_exist (in_department_id in department.department_id%type)
        l_return pls_integer;
begin
        <<check_row_exists>>
begin
10
            select 1
12
            into l_return
from department
where department_id = in_department_id;
13
14
       exception

when no_data_found or too_many_rows then

1_return := 0;
15
16
17
         end check_row_exists;
18
19
       return 1_return = 1;
end does_exist;
20
21
        procedure del (in_department_id in department.department_id%type) is
22
23
        if does_exist(in_department_id) then
    null;
24
25
          end if;
26
        end del;
27 end department_api;
28 /
30
31
```

### EXAMPLE (GOOD)

37

```
create or replace package department_api is
procedure del (in_department_id in department_id%type);
  3 end department_api;
  4 /
  5
  create or replace package body department_api is
function does_exist (in_department_id in department_department_id%type)
return boolean;
           procedure del (in_department_id in department.department_id%type) is
          begin
if does_exist(in_department_id) then
10
11
                null;
12
              end if;
13
          end del;
14
15
          function does_exist (in_department_id in department.department_id%type)
          return boolean is

1_return pls_integer;

k_exists constant pls_integer := 1;

k_something_wrong constant pls_integer := 0;
16
17
18
19
          begin
               <check_row_exists>>
20
            begin
21
                 select k_exists
22
               into 1_return
from department
where department_id = in_department_id;
23
24
25
          exception

when no_data_found or too_many_rows then

l_return := k_something_wrong;
26
           l_return := k_something_wrong;
end check_row_exists;
27
return l_return = k_exists;

return l_return = k_exists;

end does_exist;

return l_return = k_exists;

return l_return = k_exists;
28
33
34
35
36
```

### G-7230: Avoid declaring global variables public.



Reliability

### REASON

You should always declare package-level data 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.

#### EXAMPLE (BAD)

```
create or replace package employee_api as
        k_min_increase constant types.sal_increase_type := 0.01;
k_max_increase constant types.sal_increase_type := 0.5;
        g_salary_increase types.sal_increase_type := k_min_increase;
        procedure set_salary_increase (in_increase in types.sal_increase_type);
         function salary_increase return types.sal_increase_type;
     end employee_api;
 8
 9
10 create or replace package body employee_api as
11 procedure set_salary_increase (in_increase in types.sal_increase_type) is
12
           g_salary_increase := greatest(least(in_increase,k_max_increase)
13
                                            ,k_min_increase);
14
        end set salary increase;
15
16
        function salary_increase return types.sal_increase_type is
17
        begin
           return g_salary_increase;
18
        end salary_increase;
19
    end employee api;
20 /
21
22
```

```
create or replace package employee_api as
procedure set_salary_increase (in_increase in types.sal_increase_type);
function salary_increase return types.sal_increase_type;
 4 end employee_api;
 5 /
 create or replace package body employee_api as
g_salary_increase types.sal_increase_type(4,2);
        procedure init;
10
        procedure set_salary_increase (in_increase in types.sal_increase_type) is
begin
11
12
          g_salary_increase := greatest(least(in_increase
13
                                                , constants.max_salary_increase())
, constants.min_salary_increase());
14
15
         end set_salary_increase;
16
         function salary_increase return types.sal_increase_type is
17
        begin
  return g_salary_increase;
end salary_increase;
18
19
20
        procedure init
21
22
       g_salary_increase := constants.min_salary_increase();
end init;
23
24
25 begin
        init();
26
27 end employee_api;
28 /
29
30
31
32
```

## G-7240: Avoid using an IN OUT parameter as IN or OUT only.



Efficiency, 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.

EXAMPLE (BAD)

34

```
create or replace package body employee_up is
procedure rcv_emp (io_first_name in out employee.first_name%type
,io_last_name in out employee.last_name%type
                                ,io_email
                                                         in out employee.email%type
                                ,io_phone_number in out employee.phone_number%type
                                ,io hire date in out employee.hire date*type
,io_job_id in out employee.job_id*type
,io_salary in out employee.salary*type
                                ,io_commission_pct in out employee.commission_pct%type
                                ,io_manager_id in out employee.manager_id%type
,io_department_id in out employee.department_id%type
10
                                 ,in wait
                                                                  integer) is
11
             l_status pls_integer;
12
             k_pipe_name constant string(6 char) := 'mypipe';
1.3
             k_ok constant pls_integer := 1;
14
15
              -- receive next message and unpack for each column.
           l_status := sys.dbms_pipe.receive_message(pipename => k_pipe_name
16
           if l_status = k_ok then
            sys.dbms_pipe.unpack_message (io_first_name);
sys.dbms_pipe.unpack_message (io_last_name);
20
               sys.dbms_pipe.unpack_message (io_email);
sys.dbms_pipe.unpack_message (io_phone_number);
21
                sys.dbms_pipe.unpack_message (io_hire_date);
22
               sys.dbms_pipe.unpack_message (io_job_id);
sys.dbms_pipe.unpack_message (io_salary);
2.3
24
                sys.dbms_pipe.unpack_message (io_commission_pct);
2.5
                 sys.dbms pipe.unpack message (io manager id);
26
                  sys.dbms_pipe.unpack_message (io_department_id);
            end if:
27
          end rcv_emp;
     end employee_up;
29
30
31
32
33
```

```
create or replace package body employee_up is
procedure rcv_emp (out_first_name out employee.first_name%type

dut_employee.last_name%type
out employee.last_name%type
out employee.email%type
out employee.phone_number%type
                                                     out_ine_date out employee.hire_date%type out_salary out employee.salary%type
                                                     ,out_commission_pct out employee.commission_pct%type
                                                     ,out_manager_id out_employee.manager_id%type
,out_department_id out_employee.department_id%type
10
                                                      ,in_wait
                                                                                             in integer) is
11
               l_status pls_integer;
k_pipe_name constant string(6 char) := 'mypipe';
12
13
                    k_ok constant pls_integer := 1;
14
             begin
               -- receive next message and unpack for each column.

l_status := sys.dbms_pipe.receive_message(pipename => k_pipe_name , timeout => in_wait);
15
16
17
                if l_status = k_ok then
                 if 1_status = k_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_bipe_did);
    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);

18
19
20
21
22
23
24
25
26
                            sys.dbms_pipe.unpack_message (out_department_id);
                   end if;
27
               end rcv_emp;
28
29 end employee_up;
30
31
32
33
34
```

## G-7250: Always use NOCOPY when appropriate



Efficiency

REASON

When we pass OUT or IN OUT parameters in PL/SQL the Oracle Database supports two methods of passing data: By Value and By Reference.

The default, By Value, will copy all the data passed into a temporary buffer. This buffer is passed to the procedure and used during the life of the procedure. Then when processing is complete, the data in the buffer is copied to the original variable.

Passing By Reference is achieved by the NOCOPY hint, and, in contrast, it will pass a reference to the variable's data. Think of a pointer in the C language. This means that no temporary buffer is required. When passing significant amounts of data, the effects of passing values by reference can be significant.

EXAMPLE (BAD)

# 5.7.3 Procedures

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



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)

```
1 create or replace procedure my_procedure is
2 begin
3 null;
4 end my_procedure;
5 /
```

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



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)

```
1 create or replace package body my_package is
      procedure my_procedure is
    l_idx simple_integer := 1;
           k_modulo constant simple_integer := 7;
       begin
           <<mod7_loop>>
 6
           if mod(l_idx,k_modulo) = 0 then
 8
               return;
           end if;
 9
10
            l_idx := l_idx + 1;
       end loop mod7_loop;
end my_procedure;
13
    end my_package;
15
16
```

```
create or replace package body my_package is
procedure my_procedure is

l_idx simple_integer := 1;
k_modulo constant simple_integer := 7;
begin
<mod7_loop>>
loop
exit mod7_loop when mod(l_idx,k_modulo) = 0;

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

# 5.7.4 Functions

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



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)

```
1 create or replace function my_function return varchar2 is begin 3 return null; 4 end my_function; 5 /
```

```
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.



Maintainability

REASON

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

EXAMPLE (BAD)

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



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)

```
create or replace package body my_package is
function my_function (in_value in pls_integer) return boolean is
    k_yes constant pls_integer := 1;

begin
    if in_value = k_yes then
    return true;
else
    return false;
end if;
end my_function;
end my_package;

/
12
```

## EXAMPLE (BETTER)

```
create or replace package body my_package is
function my_function (in_value in pls_integer) return boolean is
    k_yes constant pls_integer := 1;

begin
    return in_value = k_yes;
end my_function;
end my_package;
/
```

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



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 return date is
begin
return sysdate;
end my_function;
end my_package;

/
```

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



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;

/
```

```
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.



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)

```
1     create or replace package department_api is
2         function name_by_id (in_department_id in departments.department_id%type)
3         return departments.department_name%type;
4     end department_api;
5     /
```

```
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;
//
```

# 5.7.5 Oracle Supplied Packages

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



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
k hello_world constant string(11 char) := 'Hello World';
begin
dbms_output.put_line(k_hello_world);
end;
end;
6 /
```

```
1  declare
2  k_hello_world constant string(11 char) := 'Hello World';
3  begin
4  sys.dbms_output.put_line(k_hello_world);
6  end;
6 /
```

# 5.7.6 Object Types

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

# 5.7.7 Triggers

## G-7710: Avoid cascading triggers.



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.

Note that the example below is an anti-pattern as Flashback Data Archive should be used for row history instead of history tables.

EXAMPLE (BAD)

```
create or replace trigger dept_br_u before update on department for each row
        insert into department_hist (department_id
,department_name
 4
                                         ,manager_id
                                         ,location id
                                          ,modification_date)
             values (:old.department_id
                   ,:old.department_name
,:old.manager_id
10
                      ,:old.location_id
11
                      ,sysdate);
12
    end;
13
14
    create or replace trigger dept_hist_br_i
15 before insert on department_hist for each row
16 begin
       insert into department_log (department_id
17
                                       , department_name
, modification_date)
18
19
                                values (:new.department_id
                                       ,:new.department_name
,sysdate);
21
22 end;
23
2.4
25
26
```

## EXAMPLE (GOOD)

Note: Again, don't use triggers to maintain history, use Flashback Data Archive instead.

```
create or replace trigger dept_br_u
    before update on department for each row
    begin
       insert into department_hist (department_id
                                    ,department_name
                                     ,manager_id
 6
                                     , \verb|modification_date|)
 8
            values (:old.department_id
                 ,:old.department_name
9
10
                   ,:old.manager_id
                   ,:old.location_id
                   ,sysdate);
13
      insert into department_log (department_id
14
                                   , department_name
, modification_date)
15
                             values (:old.department_id
16
                                   ,:old.department_name
17
                                    ,sysdate);
18
19
    end;
20 /
21
22
```

# G-7720: Avoid triggers for business logic



Efficiency, Maintainability

# REASON

When business logic is part of a trigger, it becomes obfuscated. In general, maintainers don't look for code in a trigger. More importantly, if the code on the trigger does SQL or worse PL/SQL access, this becomes a context switch or even a nested loop that could significantly affect performance.

## G-7730: If using triggers, use compound triggers



Efficiency, Maintainability

REASON

# A single trigger is better than several

EXAMPLE (BAD)

```
create or replace trigger dept_i_trg
before insert
on dept
for each row
begin
    :new.id = dept_seq.nextval;
    :new.created_on := sysdate;
    :new.created_by := sys_context('userenv','session_user');
end;

/ create or replace trigger dept_u_trg
before update
on dept
for each row
begin
    :new.updated_on := sysdate;
    :new.updated_by := sys_context('userenv','session_user');
end;
/ end;
/ /
```

```
create or replace trigger dept_ui_trg
before insert or update
on dept
for each row
begin
if inserting then
:new.id = dept_seq.nextval;
:new.created_on := sysdate;
:new.created_by := sys_context('userenv','session_user');
elsif updating then
:new.updated_on := sysdate;
:new.updated_by := sys_context('userenv','session_user');
end if;
end;
//
end;
//
15
```

# 5.7.8 Sequences

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



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;
//
```

## 5.8 Patterns

# 5.8.1 Checking the Number of Rows

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



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. This could have a big performance impact.

If we do a select count(\*) where rownum = 1 there is also some overhead as there are two context switches between the PL/SQL and SQL engines.

See the following example for a better solution.

EXAMPLE (BAD)

```
declare
      l_count pls_integer;
k_zero constant simple_integer := 0;
      k_salary constant employee.salary%type := 5000;
        into l_count
from employee
      10
11
                          from employee)
12
13
             if r_{emp.salary} < k_{salary} then
14
                my_package.my_proc(in_employee_id => r_emp.employee_id);
          end if;
end loop emp_loop;
15
        end if;
    end;
18
19
20
21
```

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



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 department%rowtype) is
l_count pls_integer;
begin
select count(*)
into l_count
from department
where department_id = in_r_department.department_id;

if l_count = 0 then
insert into department
values in_r_department;
end if;
end in;
end department_api;
//

//
```

```
create or replace package body department_api is

procedure ins (in_r_department in department%rowtype) is

begin

insert into department

values in_r_department;

exception

when dup_val_on_index then null; -- handle exception
end ins;

end department_api;

/

10
```

# 5.8.2 Access objects of foreign application schemas

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



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.product.product_name%type;
k_price constant oe.product.list_price%type := 1000;
begin
select product_name
into l_product_name
from oe.product
where list_price > k_price;
exception
when no_data_found then
null; -- handle_no_data_found;
when too_many_rows then
null; -- handle_too_many_rows;
end;
//
end;
//
14
```

```
create synonym oe_product for oe.product;
 3
        l_product_name oe_product.product_name%type;
k_price constant oe_product.list_price%type := 1000;
 6
        select product_name
         into l_product_name
from oe_product
         where list_price > k_price;
10 exception
        when no data found then
            null; -- handle_no_data_found;
12
       when too many rows then
null; -- handle too many rows;
13
14 end;
15 /
16
```

# 5.8.3 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.



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 department.department_name%type) return department%rowtype is
 3
          l_return department%rowtype;
       begin
if in_dept_name is null
               or length(in_dept_name) > 20
 8
               raise err.e_param_to_large;
       end if;
-- get the department by name
select *
from department
9
10
12
           where department_name = in_dept_name;
13
14
            return 1 return;
15
      end dept_by_name;
16 end department_api;
18
19
```

## EXAMPLE (GOOD)

```
create or replace package body department_api is
       function dept_by_name (in_dept_name in_department.department_name%type)
return department%rowtype is
          l_dept_name department.department_name%type not null := in_dept name;
           1_return department%rowtype;
        -- get the department by name select *
 6
              from department
          where department_name = l_dept_name;
10
       return 1_return;
end dept_by_name;
11
12
    end department_api;
13
14
1.5
```

## FUNCTION CALL

# 5.8.4 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.



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)

33

```
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
 8
       -- raises dup_val_on_index
 9
       insert into app_locks (lock_name) values (in_lock_name);
end request_lock;
10
11
12
       procedure release_lock(in_lock_name in varchar2) is
13
14
         delete from app_locks where lock_name = in_lock_name;
15
        end release_lock;
16 end lock_up;
17
18
    /* call bad example */
19 declare
2.0
       k_lock_name constant varchar2(30 char) := 'APPLICATION_LOCK';
21 begin
       lock_up.request_lock(in_lock_name => k_lock_name);
22
23
      lock_up.release_lock(in_lock_handle => l_handle);
24
       when others then
26
           -- log error
27
          lock_up.release_lock(in_lock_handle => l_handle);
28
           raise;
29 end;
3.0
31
32
```

#### EXAMPLE (GOOD)

48 49

```
1 /* good example */
    create or replace package body lock_up is
      function request_lock(
in varchar2,
        return varchar2 is
       l_lock_handle varchar2(128 char);
begin
          lockname => in_lock_name,
lockhandle => l_lock_handle,
expiration_secs => constants.k_one_week
);
10
11
12
         if sys.dbms_lock.request(
13
          lockhandle => l_lock_handle,
lockmode => sys.dbms_lock.x_mode,
timeout => sys.dbms_lock.maxwait,
14
15
                 release_on_commit => coalesce(in_release_on_commit, false)
16
17
        then
18
             raise errors.e_lock_request_failed;
19
20
       return 1_lock_handle;
end request_lock;
21
22
       procedure release lock(in lock handle in varchar2) is
23
24
        if sys.dbms_lock.release(lockhandle => in_lock_handle) > 0 then
    raise errors.e_lock_request_failed;
25
26
          end if;
       end release_lock;
27
28 end lock_up;
29
30
    /* Call good example */
31
    declare
     1_handle varchar2(128 char);
k_lock_name constant varchar2(30 char) := 'APPLICATION_LOCK';
32
33
34
      1_handle := lock_up.request_lock(in_lock_name => k_lock_name);
35
    lock_up.release_lock(in_lock_handle => 1_handle);
exception
36
37
38
       when others then
39
           -- log error
           lock_up.release_lock(in_lock_handle => 1_handle);
40
           raise;
41
    end;
42
43
44
45
46
47
```

5.8.5 Use dbms\_application\_info package to follow progress of a process

G-8510: Always use dbms\_application\_info to track program process transiently.



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 vssesion view.

EXAMPLE (BAD)

```
create or replace package body employee_api is
procedure process_emps is
begin

<pr
```

```
create or replace package body employee_api is
     procedure process_emps is
      begin
        <<employees>>
       for emp_rec in (select employee_id
                       from employee
                     order by employee_id)
8
9
        sys.dbms_application_info.set_action('processing ' || emp_rec.employee_id); end loop employees;
10
11
     end process_emps;
   end employee_api;
14
15
```

			2.1				0 77			
5 8 5	l lea d	hme a	application	into	markadi	a to	tollows i	nrograce	nt a	nrocace

- 134/135 - Primus Solutions AG 2024

# 6. 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:

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

## Precheck

Developers should complete the following checklist prior to requesting a peer code review.

- Can I answer "Yes" to each of these questions?
- Did I take time to think about what I wanted to do before doing it?
- Would I pay for this?
- Can I defend my work / decisions I made?
- NO sloppiness.
- Code is well formatted.
- Code is not duplicated in multiple places.
- Named variables.
- Tables have foreign keys (and associated indexes)...
- Run the APEX Advisor (if using APEX).
- Code is well commented.
- Package specs includes a description of what the procedure does and what the input variables represent.
- Package body includes comments throughout the code to indicate what is happening.
- The application includes end user help.