# PL/SQL & SQL

**CODING GUIDELINES
VERSION 2.0**

ORACLE® **Platinum
Partner**

trivadis

makes **IT** easier.

# PL/SQL
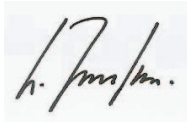# & SQL

## Coding Guidelines

Trivadis AG

# Foreword

Coding Guidelines are an important quality standard for any programming language. Not only that coding based on standards results in better software products, standards also help programmers to work in teams. These coding guidelines represent many years of experience – not only delightful ones - in constructing and maintaining PL/SQL and SQL programs. These programs are pieces of software to implement business logic in order to support high end data processing. Or in other words to get value out of one of the most important assets of a company - the company's information. From small functions to help to automate a database administrators work to complex frameworks for historization or other purposes. Many software products are programmed fully in PL/SQL or in SQL. I am convinced that these PL/SQL and SQL Coding Guidelines are a valuable contribution to enhance software quality and improve team performance.

May the code be with you.

Urban Lankes
CEO Trivadis

"Roger and his team have done an excellent job of providing a comprehensive set of clear standards that will undoubtedly improve the quality of your code. If you do not yet have standards in place, you should give strong consideration to using these as a starting point."

Steven Feuerstein
PL/SQL Evangelist

Coding Guidelines are a crucial part of software development. It is a matter of fact, that code is more often read than written – therefore we should take efforts to ease the work of the reader, which is not necessarily the author.
I am convinced that this standard may be a good starting point for your own guidelines.

Roger Troller
Senior Consultant Trivadis

# License

## Trademarks

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

## Disclaimer

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

# Revision History

| Vers | Who | Date | Comment |
|------|-----|------|---------|
| 0.1 | Troller | 17.03.2009 | Created. |
| 0.2 | Kulessa | 04.05.2009 | Extended. |
| 0.3 | Reiner | 12.05.2009 | Extended with comments in code. |
| 0.4 | Troller | 14.05.2009 | Extended formatting. |
| 0.5 | Kulessa | 20.05.2009 | Added more CodeXpert rules. |
| 0.6 | Troller | 22.05.2009 | Formatting changes. Added categories to rules. |
| 0.7 | Reiner | 10.06.2009 | Extended with example code commenting. |
| 0.8 | Troller | 18.06.2009 | Finalized. |
| 0.9 | Bushnell | 23.06.2009 | Translation |
| 1.0 | Troller | 01.07.2009 | Ready for inspection |
| 1.1 | Troller | 19.08.2009 | Added Inspection results AFl |
| 1.2 | Troller | 21.08.2009 | Added Inspection results ThM |
| 1.3 | Troller | April 2010 | Several Corrections<br>New Rule Oracle Supplied Packages |
| 1.3.1 | Troller | October 2010 | Some formatting |
| 2.0 | Troller | August 2011 | Added Error Handling Section<br>Added Rule 23<br>Added Rule 30<br>Added Rule 31<br>Added Rule 43 |

# Table of Contents

# 1. Introduction

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

## 1.1. Scope

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

## 1.2. Document Conventions

### 1.2.1 Coloring & Emphasis:

| | |
|---|---|
| Blue | Text colored blue indicates a PL/SQL or SQL keyword. |
| **Bold** | Text with additional emphasis. |

### 1.2.2 Keywords

| | |
|---|---|
| **Always** | Emphasizes this rule must be enforced. |
| **Never** | Emphasizes this action must not happen. |
| **Do Not** | 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 recommendation. |
| **Reason** | Explains the thoughts and purpose behind a rule or recommendation. |

### 1.2.3 Icons

| | |
|---|---|
| | **Information Tag** |
| | **Caution** |
| | **Performance relevance** |
| | **Maintainability** |
| | **Readability** |

# 2. Why are standards important

For a machine executing a program, code formatting is of no importance. But 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).
- Developers do not have to think about where to search for something - it is already defined.
- The developers do not have to think about how to name something - it is already defined.
- The code has a structure that makes it easier to avoid making errors.
- The code is more efficient with regards to 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.

# 3. Naming Conventions

## 3.1. General Guidelines

1. Do not use names with a leading numeric character.
2. Always choose meaningful and specific names.
3. Avoid using abbreviations unless the full name is excessively long.
4. Avoid long abbreviations. Abbreviations should be shorter than 5 characters.
5. Any abbreviations must be widely known and accepted. Create a glossary with all accepted abbreviations.
6. Do not use ORACLE reserved words as names. A list of ORACLE's reserved words may be found in the dictionary view V$RESERVED_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 variables

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.

A widely used convention is to follow {prefix_} variablecontent_{_suffix} pattern.
The following table shows a possible set of naming conventions.

*Naming Conventions for PL/SQL*

| Identifier | Prefix / Suffix | Example |
|---|---|---|
| Global Variable | P: g | g_version |
| Local Variable | P: l | l_version |
| Cursor | P: c | c_employees |
| Record | P: r | r_employee |
| Array / Table | P: t | t_employees |
| Object | P: o | o_employee |
| Cursor Parameter | P: cp | cp_empno |
| In Parameter | P: in | in_empno |
| Out Parameter | P: out | out_ename |
| In/Out Parameter | P: io | io_employee |
| Type Definitions | S: type | r_employee_type |
| Exception | P: e | e_employee_exists |
| Constants | P: co | co_empno |

## 3.3. Database Object Naming Conventions

*Naming Conventions for Database Objects*

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.

| Identifier | Naming Convention |
|---|---|
| **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.<br><br>**Examples:**   -  employees_ct<br>              -  orders_ct |
| **Column** | • Singular name of what is stored in the column<br>   o  unless the column data type is a collection, then you use a plural name<br><br>• Add a comment to the database dictionary for every column. |

| Identifier | Naming Convention |
|---|---|
| **DML / Instead of Trigger** | Choose a naming convention that includes:<br><br>Either<br>• the name of the object the trigger is added to,<br>• any of the triggering events:<br>   o _br_iud   → Before Row on Insert, Update and Delete<br>   o _io_id    → Instead of Insert and Delete<br><br>Or<br>• the name of the object the trigger is added to,<br>• the activity done by the trigger,<br>• the suffix "_trg"<br><br>**Examples:**    -   employees_br_iud<br>                -   orders_audit_trg<br>                -   orders_journal_trg |
| **Foreign Key Constraint** | • Table abbreviation followed by referenced table abbreviation followed by a "_fk" an and optional number suffix.<br><br>• Optionally prefixed by a project abbreviation.<br><br>**Examples:**    -   empl_dept_fk<br>                -   sct_icmd_ic_fk1 |
| **Function** | • Name is built from a verb followed by a noun. The name of the function should answer the question "What is the outcome of the function?"<br><br>**Examples**:          get_employee<br><br>• If more than one function provides the same outcome, you have to be more specific with the name.<br><br>**Examples**:          get_employee_by_name<br>                    get_employee_by_email<br>                    get_employee_by_phone_no |
| **Index** | • Indexes serving a constraint (primary, unique or foreign key) are named accordingly.<br><br>• Other indexes should have the name of the table and columns (or their purpose) in their name and should also have _idx as a suffix. |
| **Object Type** | • The name of an object type is built by its content (singular) followed by an "_ot" suffix.<br><br>**Examples**:          employee_ot |

| Identifier | Naming Convention |
|---|---|
| **Package** | • Name is built from the content that is contained within the package.<br><br>**Examples**:    - employees_api (API package for the employee table).<br>       - logging_up (Utility package including logging support). |
| **Primary Key Constraint** | • Table name or table abbreviation followed by the suffix "_pk".<br>• Optionally prefixed by a project abbreviation.<br><br>**Examples**:    - employees_pk<br>       - departments_pk<br>       - sct_contracts_pk |
| **Procedure** | • Name is built from a verb followed by a noun. The name of the procedure should answer the question "What is done?" Procedures and functions are often named with underscores between words because some editors write all letters in upper case in the object tree (e.g. Quest), so it is difficult to read them.<br><br>**Examples**:    - calculate_salary<br>       - set_hiredate<br>       - check_order_state |
| **Sequence** | • Name is built from the table name (or its abbreviation) the sequence serves as primary key generator and the suffix _seq or the purpose of the sequence followed by a _seq.<br><br>**Examples**:    - employees_seq<br>       - order_number_seq |
| **Synonym** | • Synonyms should be used to address an object in a foreign schema rather than to rename an object. Therefore synonyms should share the name with the referenced object. |
| **System Trigger** | • Name of the event the trigger is based on.<br>• Activity done by the trigger.<br>• Suffix "_trg".<br><br>**Examples**:    - ddl_audit_trg<br>       logon_trg |
| **Table** | • Plural name of what is contained in the table.<br>   o unless the table is designed to always hold one row only – then you should use a singular name<br>• Optionally prefixed by a project abbreviation.<br>• Add a comment to the database dictionary for every table.<br><br>**Examples**:    - employees<br>       - departments<br>       - sct_contracts<br>       - sct_contract_lines<br>       - sct_incentive_modules |

| Identifier | Naming Convention |
|---|---|
| **Temporary Table** (Global Temporary Table) | • Naming as described for tables.<br>• Optionally suffixed by "_tmp"<br><br>**Examples**:     -   employees_tmp<br>                -   contracts_tmp |
| **Unique Key Constraint** | • Table name or table abbreviation followed by the role of the unique constraint, an "_uk" and an optional number suffix.<br>• Optionally prefixed by a project abbreviation.<br><br>**Examples**:     -   employees_name_uk<br>                -   departments_deptno_uk<br>                -   sct_contracts_uk<br>                -   sct_coli_uk<br>                -   sct_icmd_uk1 |
| **View** | • Plural name of what is contained in the view.<br>• Optionally prefixed by a project abbreviation.<br>• Optionally suffixed by an indicator identifying the object as a view (mostly used, when a 1:1 view layer lies above the table layer)<br>• Add a comment to the database dictionary for every view and every column.<br><br>**Examples**:     -   active_orders<br>                -   orders_v (a view to the orders table) |

# 3.4. Database Object Check Scripts

*List of Check Scripts*

| Script | Checks | |
|---|---|---|
| Check_naming_conventions.sql | **Checks** | -   Table Comments<br>-   Column Comments<br>-   Sequence Suffix<br>-   Unique Constraint Suffix<br>-   Primary Key Suffix<br>-   Foreign Key Suffix |

# 4.  Coding Style

## 4.1.  Formatting

*PL/SQL Code Formatting*

| Rule | Description |
|---|---|
| 1 | Keywords are written uppercase, names are written in lowercase. |
| 2 | 3 space indention. |
| 3 | One command per line. |
| 4 | Keywords THEN, LOOP, IS, ELSE, ELSIF, 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. |

```
PROCEDURE set_salary(in_empno IN emp.empno%TYPE)

IS
❷CURSOR c_emp(cp_empno emp.empno%TYPE)
    IS
      ❶ SELECT ename
           ❺ ,sal
             FROM emp
            WHERE empno = cp_empno
        ORDER BY ename;

   --
   r_emp          c_emp%ROWTYPE;
   l_new_sal      emp.sal%TYPE;
BEGIN
   OPEN c_emp(in_empno);
 ❸FETCH c_emp INTO   r_emp;
   CLOSE c_emp;

   --
   get_new_salary (p_empno_in => in_empno
           ❻ ,p_sal_out => l_new_sal);

   --
 ❽-- Check whether salary has changed
   IF r_emp.sal <> l_new_sal
 ❹ THEN
       UPDATE emp
     ❼  SET sal = l_new_sal
        WHERE empno = in_empno;
   END IF;
END set_salary;
```

### 4.1.1 Tool support

Formatting your code as described above may be achieved using various tools. We have written templates for

- SQL Navigator
- Toad
- PL/SQL Developer

### 4.1.2 Code Formatting Files

***Formatting Tool and Templates***

| Tool | File Name |
|---|---|
| Quest SQL Navigator | SQLNavigatorFormating.opt |
| Quest Toad | SQLNavigatorFormating.opt |
| www.sqlinform.com | |
| PL/SQL Developer | PL-SQL-Developer-Beautifier.zip |
| TORA | |

# 4.2. Code Commenting

Inside a program unit only use the line commenting technique "--".

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. Tags beginning with % are always on a new line (exception {%link...}).

Documentation generation can be done using PLSQLDoc (Plugin to PL/SQL Developer and therefore only available when working with this tool) or PLDoc (Open Source Product – no longer improved and therefore not recommendable).

***Commenting Tags***

| Tag | Meaning and Pattern | Program | Example |
|---|---|---|---|
| {*} | enumeration within another tag<br>{*} <value> <description> | PLSQLDoc | {*} 0 this is zero<br>{*} 1 this is one<br>{*} other all others |
| author | author of an object<br>%author <name> | PLSQLDoc | %author Max Mustermann |
| deprecated | old version of a program unit<br>%deprecated | PLDoc | %deprecated |
| link | link within text to a location in the documentation or on the Web with alternate display text<br>{%link <link> <display text>} | PLSQLDoc | .. {%link EMP.html EMP}…<br>…{%link<br>http://www.trivadis.com<br>Trivadis-Homepage}… |
| param | parameter of a procedure or function<br>%param <parameter> <description> | PLDoc<br>PLSQLDoc | %param p_deptno departmentno. |
| raises | raised exception in a program unit<br>%raises <exception> <description> | PLSQLDoc | %raises<br>NO_DATA_FOUND<br>description |

| Tag | Meaning and Pattern | Program | Example |
|---|---|---|---|
| return | return value of a function<br>%return <value or description> | PLDoc<br>PLSQLDoc | %return sum_sal |
| see | reference to a site in the documentation or in the web<br>%see <link> | PLSQLDoc | %see EMP.html<br>%see<br>http://www.trivadis.com |
| skip | the current comment is not used for generating documentation<br>{%skip} | PLSQLDoc | ...<br>{%skip}<br>.... |
| throws | thrown exception in a program unit<br>%throws <exception> <description> | PLDoc | %throws<br>NO_DATA_FOUND<br>description |
| usage | usage of an object<br>%usage <description> | PLSQLDoc | %usage This object is used to figure out the salary. |
| value | possible values of parameters, results or other variables<br>%value <value> <description> | PLSQLDoc | %value 0 OK<br>%value 1 exception |
| version | version number of an object<br>%version <number of version> | PLSQLDoc | %version 3.2.1 |

To format text in the documentation HTML tags are used inside comment blocks.

### HTML Tags

| HTML | Meaning | Excample |
|---|---|---|
| <b> | Bold | <b>text in bold</b> |
| <i> | Italic | <i>text in italic</i> |
| <u> | underline | <u>underlined text</u> |
| <br> | new line | ... <br> |
| <hr> | horizontal line | <hr> |
| <code> | example in courier font | <code>if...else...</code> |
| <ul><br><li> | unnumbered list | <ul><br><li>first<br><li>second<br></ul> |
| <table><br><tr><br><td> | insert a table | <table><br><tr><br>    <td>...</td><br>    <td>...</td><br></tr><br>...<br></table> |

```
CREATE OR REPLACE FUNCTION  get_SalProzent (in_empno IN NUMBER)
   RETURN NUMBER
IS
/**
  General <b>Information</b> about <i>function</i> to show HTML-
  formatting
  <br>

  To show the cross-reference the name of the procedure D1D_faktor is
  noticed here. <br>

  The tag 'link' can appear within another text as shown below:
  The employees are listed in the table {%link emp.html EMP}.
  And the {%link http://www.trivadis.com Trivadis}-Company has a
  website.

  %param in_empno Number of employee
  %return part in percent

  %value 100 return value is 100%
  %value 0 return value is 0%.
  %value 25 return value is 25%.

  %raises NO_DATA_FOUND If no salary is found, no part of salary
  can be issued.

  %author Scott Tiger
  %version 1.2.3
  %usage description of the function

  %see sal_info.get_sal;2
  %see http://www.trivadis.com
*/
  l_sal   NUMBER(7,2);
  l_summe NUMBER(7,2);
BEGIN
   SELECT sal
     INTO l_sal
     FROM emp
    WHERE empno = in_empno;
   (…)
END get_SalProzent;
```

## 4.2.1 IDE Templates

### *Code Templates*

Templates are stored in the templates subfolder of each tool.

| Object Type | QUEST SQL Navigator | QUEST Toad | Allround Automations PL/SQL Developer | ORACLE SQL Developer |
|---|---|---|---|---|
| Function | Func.txt | Function.fnc | Function.fnc | Function.sql |
| Package Specification | Pkgspec.txt | Package.spc | Package.pks | Package.pks |
| Package Body | Pkgbody.txt | Packagebody.pkb | Packagebody.bdy | Packagebody.pkb |
| Procedure | Proc.txt | Procedure.prc | Procedure.prc. | Procedure.sql |
| Object Type Specification | Typspec.txt | Type.tps | Type.tps | Type.sql |
| Object Type Body | Typbody.txt | Typbody.tpb | Typebody.tpb | Typebody.sql |

# 5. Language Usage

The language usage section is partly based on the rules defined within the tool CodeXpert by Quest Software (www.quest.com). All text taken directly from this source is marked by **(CodeXpert)**. Every rule, that may be checked by CodeXpert ist marked with a red label including the rule number within CodeXpert.

## 5.1. General

1. Try to label your sub blocks.
[CodeXpert 4403]

2. Always have a matching loop or block label.
[CodeXpert 4403]

**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**:

```
-- Good
BEGIN

    <<prepare_data>>
    BEGIN

        …;
    END prepare_data;

    …
    <<process_data>>
    BEGIN

        …
    END process_data;
END;
```

3. Avoid defining variables that are not used.
[CodeXpert 6405]

4. Avoid dead code in your programs.
[CodeXpert 6801]

**Reason:**   Any part of your code, which is no longer used or cannot be reached, should be eliminated from your programs.

5. Avoid using literals in your code.
[CodeXpert 4602]

**Reason:**   Literals are often used more than once in your code. Having them defined as a constant reduces typos in your code.

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 get_<name> deterministic package function for every constant.

**Example**:

```
-- Bad
DECLARE
   l_function  player.function_name%TYPE;
BEGIN
   SELECT p.function
     INTO l_function
     FROM player p
   WHERE …
   --
   IF l_function = 'LEADER'
   THEN
      …
```

```
-- Good
CREATE OR REPLACE PACKAGE constants_up
IS
   co_leader CONSTANT player.function_name%TYPE := 'LEADER';
END constants_up;
/

DECLARE
   l_function    player.function_name%TYPE;
BEGIN
   SELECT p.function
     INTO l_function
    FROM player p
   WHERE …
   --
   IF l_function = constants_up.co_leader
   THEN
```

6. Avoid storing ROWIDs or UROWIDs in a table. **(CodeXpert)**
[CodeXpert 5801]

> **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.

7. Avoid nesting comment blocks. **(CodeXpert)**
[CodeXpert 2401]

> **Reason:** A start-of-comment ( /* ) was found within comments. This can make the code more difficult to read.
> This situation can arise as you go through multiple modifications to code.

## 5.2. Variables & Types

### 5.2.1 General

8. Try to use anchored declarations for variables, constants and types.

**Reason**      Changing the size of the database column ename in the emp table from VARCHAR2(10) to VARCHAR2(20) will result in an error within your code whenever a value larger than 10 bytes is read. This can be avoided using anchored declarations.

**Example**:

```
-- Bad
DECLARE
    l_ename  VARCHAR2(10);
BEGIN
    SELECT e.ename
      INTO l_ename
      FROM emp e
     WHERE …
END;
```

```
-- Good
DECLARE
    l_ename emp.ename%TYPE;
BEGIN
    SELECT e.ename
      INTO l_ename
      FROM emp e
     WHERE …
END;
```

9. Try to have a single location to define your types. This single type could either be a type specification package or the database (database defined types).
[CodeXpert 2812]

**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.

10. Try to use subtypes for constructs used often in your application.
[CodeXpert 2812]

**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.

**Example**:

```
-- Bad
l_code_section VARCHAR2(30) := 'TEST_PCK';
```

```
-- Good
l_code_section types_pck.ora_name_type := 'TEST_PCK';
```

### List of possible subtype definitions

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

11. Never initialize variables with NULL.
[CodeXpert 5003]
    **Reason**: Variables are initialized to NULL by default.

12. Avoid comparisons with null value, consider using IS [NOT] NULL. **(CodeXpert)**
[CodeXpert 5001]
    **Reason:** The NULL value can cause confusion both from the standpoint of code review and code execution. You should always use the IS NULL or IS NOT NULL syntax when you need to check to see if a value is or is not NULL.

13. Avoid initializing variables using functions in the declaration section.
[CodeXpert 2802]
    **Reason**: If your initialization fails you will not be able to handle the error in your exceptions block.

    **Example**:

```
-- Bad
DECLARE
    l_code_section  VARCHAR2(30) := 'TEST_PCK';
    l_company_name VARCHAR2(30) := util_pck.get_company_name(in_id => 47);
BEGIN
    …
END;
```

```
-- Good
DECLARE
    l_code_section  VARCHAR2(30) := 'TEST_PCK';
    l_company_name  VARCHAR2(30);
BEGIN
    <<init>>
    BEGIN
        l_companyName := util_pck.get_company_name(inId => 47);
    EXCEPTION
        WHEN VALUE_ERROR
        THEN
            ...;
    END init;
END;
```

## 14. Never overload data structure usages.

**Example**:

```
-- Bad
<<main>>
DECLARE
    l_variable   VARCHAR2(30) := 'TEST_PCK';
BEGIN
    <<sub>>
    DECLARE
        l_variable VARCHAR2(30) := 'TEST';
    BEGIN
        DBMS_OUTPUT.PUT_LINE(l_variable || ' -'  || main.l_variable);
    END sub;
END main;
```

## 15. Never use quoted identifiers.

**Reason:**     Quoted identifiers make your code hard to read and maintain.

**Example**:

```
-- Bad
DECLARE
    "sal+comm"   NUMBER(10);
BEGIN
    ...
```

## 16. Avoid using overly short names for declared or implicitly declared identifiers.
**(CodeXpert)**

**Reason:**     You should ensure that the name you've 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.

## 17. Avoid the use of ROWID or UROWID **(CodeXpert)**

**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.

## 5.2.2 Numeric Data Types

**18. Avoid declaring NUMBER variables or subtypes with no precision.**
[CodeXpert 2829]

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

**19. Try to use PLS_INTEGER instead of NUMBER for arithmetic operations with integer values (no decimal point).**
[CodeXpert 2831]

**Reason:** PLS_INTEGER having a length of -2,147,483,648 to 2,147,483,647, on a 32bit system.

There are many reasons to use PLS_INTEGER instead of NUMBER:

- PLS_INTEGER uses less memory
- PLS_INTEGER uses machine arithmetic, which is up to three times faster than library arithmetic which is used by NUMBER.

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

## 5.2.3 Character Data Types

**20. Avoid using CHAR data type.**
[CodeXpert 2804]

**Reason:** CHAR is a fixed length data type which should only be used when appropriate. CHAR columns/variables are always filled to the specified length, this may lead to side-effects.

**21. Avoid using VARCHAR data type.**
[CodeXpert 2805]

**Reason:** The VARCHAR data type is a subtype of VARCHAR2. There is a strong possibility, that the meaning of VARCHAR might change in future version of ANSI SQL Standard. ORACLE recommends that you avoid using VARCHAR and use VARCHAR2 instead.

22. Never use zero-length strings to substitute NULL.

**Reason:** Today zero-length strings and NULL are handled similarly by ORACLE. There is no guarantee that this will still be the case in future releases, therefore if you mean NULL use NULL.

**Example**:

```
-- Bad
l_char := '';
```

```
-- Good
l_char := NULL;
```

23. Always define your VARCHAR2 variables using CHAR SEMANTIC.

**Reason:** Changes to the NLS_LENGTH_SEMANTIC will only be picked up by your code after a recompilation.

**Example**:

```
-- Good
DECLARE
    l_name VARCHAR2(10 CHAR);
    ...
BEGIN
    ...
```

## 5.2.4 Boolean Data Types

24. Try to use boolean data type for values with dual meaning.

**Reason:** The use of TRUE and FALSE clarifies that this is a boolean value and makes the code easier to read.

**Example**:

```
-- Bad
DECLARE
    l_bigger NUMBER(1);
BEGIN
    IF l_newFile < l_oldFile
    THEN
        l_bigger := 1;
    ELSE
        l_bigger := 0;
    END IF;
...
```

```
-- Good
DECLARE
   l_bigger BOOLEAN;
BEGIN
   IF l_newFIle < l_oldFile
   THEN
      l_bigger := TRUE;
   ELSE
      l_bigger := FALSE;
   END IF;
...
```

```
-- Better
DECLARE
   l_bigger BOOLEAN;
   ...
BEGIN
   l_bigger := NVL(l_newFile < l_oldFile, FALSE);
...
```

### 5.2.5  Large Objects

25. Avoid using the LONG and LONG RAW data types.
[CodeXpert 2803]
**Reason:**    LONG and LONG RAW data type support will be discontinued in future ORACLE releases.

## 5.3.  DML and SQL

26. Always specify the target columns when executing an insert command.
**Reason:**    Data structures often change. Having the target columns in your insert statements will lead to change-resistant code.
**Example:**

```
-- Bad
INSERT INTO messages
     VALUES (l_mess_no
            ,l_mess_typ
            ,l_mess_text );
```

```
-- Good
INSERT INTO messages (mess_no
                     ,mess_typ
                     ,mess_text )
  VALUES (l_mess_no
         ,l_mess_typ
         ,l_mess_text );
```

**27. Always use table aliases when your SQL statement involves more than one source.**
**Reason:** It is more human readable to use aliases instead of writing columns with no table information.

**Example**:

```
-- Good
SELECT a.pid ,a.name
       ,a.birthday ,b.country
  FROM person a JOIN country b ON (a.cid = b.cid)
  WHERE …
```

**28. Try to use ANSI-join syntax, if supported by your ORACLE version.**
**Reason:** ANSI-join syntax does not have as many restrictions as the ORACLE join syntax. Furthermore ANSI join syntax supports the full outer join. A third advantage of the ANSI join syntax is the separation of the join condition from the query filters.

**Example**:

```
-- Good
SELECT a.pid ,a.name
       ,a.birthday ,b.country
  FROM person a JOIN country b ON (a.cid = b.cid)
  WHERE …
```

**29. Try to use anchored records as targets for your cursors.**
**Reason:** Using cursor-anchored records as targets for your cursors enables the possibility of changing the structure of the cursor without regard to the target structure.

**Example**:

```
-- Bad
DECLARE
   CURSOR c_user IS
      SELECT user_id, firstname, lastname
        FROM user;
   --
   l_user_id    user.user_id%TYPE;
   l_firstname  user.firstname%TYPE;
   l_lastname   user.lastname%TYPE;
BEGIN
   OPEN c_user;
   FETCH c_user INTO l_user_id, l_firstname, l_lastname;
   WHILE c_user%FOUND
   LOOP
      -- do something with the data
      FETCH c_user INTO l_user_id, l_firstname, l_lastname;
   END LOOP;
   CLOSE c_user;
END;
```

```
-- Good
DECLARE
    CURSOR c_user IS
        SELECT user_id, firstname, lastname
            FROM user;
    r_user c_user%ROWTYPE;
BEGIN
    OPEN c_user;
    FETCH c_user INTO r_user;
    <<process_user>>
    WHILE c_user%FOUND
    LOOP
        -- do something with the data
        FETCH c_user INTO r_user;
    END LOOP process_user;
    CLOSE c_user;
END;
```

## 5.3.1 BULK OPERATIONS

30. Use BULK OPERATIONS (BULK COLLECT, FORALL) whenever you have to repeatedly execute a DML or SELECT command for more than 4 times.

**Reason:** Context switches between PL/SQL and SQL are extremely costly. (Depending on the PLSQL_OPTIMIZE_LEVEL parameter this will be done automatically.)

**Example**:
```
-- Bad
DECLARE
    TYPE t_employee_type IS TABLE OF emp.empno%TYPE;
    t_employees t_employee_type := t_employee_type(7369,7698,7839,7844,7876);
BEGIN
    FOR i IN 1..t_employees.COUNT()
    LOOP
        UPDATE emp
            SET sal = sal * 1.1
            WHERE empno = t_employees(i);
    END LOOP;
END;
```

```
-- Good
DECLARE
   TYPE t_employee_type IS TABLE OF emp.empno%TYPE;
   t_employees t_employee_type := t_employee_type(7369,7698,7839,7844,7876);
BEGIN
   FORALL i IN 1..t_employees.COUNT()
      UPDATE emp
         SET sal = sal * 1.1
       WHERE empno = t_employees(i);
END;
```

# 5.4. Control Structures

## 5.4.2 CURSOR

31. Always use %NOTFOUND instead of NOT %FOUND to check whether a cursor was successful.
[CodeXpert 2613]

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

**Example:**

```
-- Bad
LOOP
    FETCH c_employees INTO r_employee;
    EXIT WHEN NOT c_employees%FOUND;
    ...
END LOOP;
```

```
-- Good
LOOP
    FETCH c_employees INTO r_employee;
    EXIT WHEN c_employees%NOTFOUND;
    ...
END LOOP;
```

32. Avoid using %NOTFOUND directly after the FETCH when working with BULK OPERATIONS and LIMIT clause. Use [array_name].COUNT() instead to check whether further FETCHs are needed.

**Reason:** %NOTFOUND is set to TRUE as soon as less then the number of rows defined in the LIMIT clause have been read.

**Example:**

```
-- This example will only show 10 of 14 employees
DECLARE
   TYPE t_employee_type IS TABLE OF emp%ROWTYPE;
   t_employees  t_employee_type;
   CURSOR c_emp
      IS SELECT *
           FROM emp
          ORDER BY empno;
BEGIN
   OPEN c_emp;
   <<process_emp>>
   LOOP
      FETCH c_emp BULK COLLECT INTO t_employees LIMIT 5;
      EXIT process_emp WHEN c_emp%NOTFOUND;

      <<display_emp>>
      FOR i IN 1..t_employees.COUNT()
      LOOP
         sys.dbms_output.put_line(t_employees(i).ename);
      END LOOP display_emp;
   END LOOP process_emp;
   CLOSE c_emp;
END;
```

```
-- This example does 4 fetches where 3 were sufficient
DECLARE
   TYPE t_employee_type IS TABLE OF emp%ROWTYPE;
   t_employees  t_employee_type;
   CURSOR c_emp
      IS SELECT *
           FROM emp
          ORDER BY empno;
BEGIN
   OPEN c_emp;
   <<process_emp>>
   LOOP
      FETCH c_emp BULK COLLECT INTO t_employees LIMIT 5;
      EXIT WHEN t_employees.COUNT() = 0;
      <<display_emp>>
```

```
        FOR i IN 1..t_employees.COUNT()
        LOOP
            sys.dbms_output.put_line(t_employees(i).ename);
        END LOOP display_emp;
    END LOOP process_emp;
    CLOSE c_emp;
END;
```

```
-- This examples does the trick (3 fetches only and process all rows)
DECLARE
    TYPE t_employee_type IS TABLE OF emp%ROWTYPE;
    t_employees  t_employee_type;
    CURSOR c_emp
        IS SELECT *
            FROM emp
           ORDER BY empno;
BEGIN
    OPEN c_emp;
    <<process_emp>>
    LOOP
        FETCH c_emp BULK COLLECT INTO t_employees LIMIT 5;
        <<display_emp>>
        FOR i IN 1..t_employees.COUNT()
        LOOP
            sys.dbms_output.put_line(t_employees(i).ename);
        END LOOP display_emp;
        EXIT WHEN t_employees.COUNT() = 0 OR c_emp%NOTFOUND;
    END LOOP process_emp;
    CLOSE c_emp;
END;
```

**33. Always close locally opened cursors.**

**Reason:** Any cursors left open can consume additional System Global Area (i.e. SGA) memory space 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". For example, the following procedure opens and fetches, but does not close its cursor – which may cause problems like those described above.

**Example:**

```
-- bad
CREATE PROCEDURE not_close_cursor (out_count OUT INTEGER)
AS
   CURSOR c1
   IS
      SELECT COUNT (*)
        FROM all_users;
BEGIN
   out_count := 0;
   OPEN c1;
   FETCH c1
    INTO out_count;
END not_close_cursor;
   ...
```

```
-- Good
CREATE PROCEDURE close_cursor (out_count OUT INTEGER)
AS
   CURSOR c1
   IS
      SELECT COUNT (*)
        FROM all_users;
BEGIN
   out_count := 0;
   OPEN c1;
   FETCH c1
    INTO out_count;
   CLOSE c1;
END close_cursor;
```

34. Avoid procedure or function calls between a SQL operation and an implicit cursor test. **(CodeXpert)**

[CodeXpert 2603]

**Reason:** Oracle provides a variety of cursor attributes, such as %FOUND and %ROWCOUNT, that you can use to obtain information about the status of your 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.

**Example:**

```
-- Bad
CREATE PROCEDURE remove_emp_and_process (in_id IN emp.empno%TYPE)
AS

BEGIN
    DELETE FROM emp
          WHERE empno = in_id
      RETURNING deptno INTO l_deptno;

    process_department (...);

    IF SQL%ROWCOUNT > 1
    THEN
        -- Too many rows deleted! Rollback and recover...
        ROLLBACK;
    END IF;
END remove_emp_and_process;
```

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

35. Try to use CASE rather than an IF statement with multiple ELSIF paths.
[CodeXpert 4213]

**Reason:**    IF statements containing multiple ELSIF tend to become complex quickly.

**Example:**

```
-- bad
IF l_color = 'red'
THEN
   ...
ELSIF l_color = 'blue'
THEN
   ...
ELSIF l_color = 'black'
THEN
   ...
```

```
-- Good
CASE l_color
   WHEN 'red'   THEN ...
   WHEN 'blue'  THEN ...
   WHEN 'black' THEN ...
END
```

36. Try to use CASE rather than DECODE.
[CodeXpert 5816]

**Reason:**    DECODE is an old function that has been replaced by the easier-to-understand and more common CASE function. Contrary to the DECODE statement CASE may also be used directly within PL/SQL.

**Example:**

```
-- Bad
BEGIN
   SELECT DECODE(dummy, 'A', 1
                      , 'B', 2
                      , 'C', 3
                      , 'D', 4
                      , 'E', 5
                      , 'F', 6
                      , 7)
     INTO l_result
     FROM dual;
...
```

```
-- Good
BEGIN
   l_result := CASE dummy
                     WHEN 'A' THEN 1
                     WHEN 'B' THEN 2
                     WHEN 'C' THEN 3
                     WHEN 'D' THEN 4
                     WHEN 'E' THEN 5
                     WHEN 'F' THEN 6
                     ELSE 7
               END;
...
```

37. Always use COALESCE instead of NVL, if parameter 2 of the NVL function is a function call or a SELECT statement.

**Reason:** The NVL function always evaluates both parameters before deciding which one to use. This can be harmful if parameter 2 is either a function call or a select statement, as it will be executed regardless of whether parameter 1 contains a NULL value or not.
The COALESCE function does not have this drawback.

**Example:**

```
-- Bad
SELECT NVL(dummy, function_call())
   FROM dual;
```

```
-- Good
SELECT COALESCE(dummy, function_call())
   FROM dual;
```

38. Always use CASE instead of NVL2 if parameter 2 or 3 of NVL2 is either a function call or a SELECT statement.

**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
SELECT NVL2(dummy, 'Yes', 'No')
   FROM dual;
```

```
-- Good
SELECT CASE
            WHEN dummy IS NULL THEN 'No'
            ELSE 'Yes'
        END
   FROM dual;
```

## 5.4.4 Flow Control

39. Never use GOTO statements in your code.
[CodeXpert 4001,4002,4003]

40. Always label your loops.
[CodeXpert 4402,4403,4405]

**Example:**

```
-- Good
BEGIN

   <<process_employees>>
   FOR r_employee IN (SELECT * FROM emp)

   LOOP

       …
   END LOOP process_employees;

END;
```

41. Always use a CURSOR FOR loop to process the complete cursor results unless you are using bulk operations.

**Example:**

```
-- Good
BEGIN

   <<read_employees>>
   FOR r_employee IN c_employee

   LOOP

       …
   END LOOP read_employees;

END;
```

42. Always use a NUMERIC FOR loop to process a dense array.

**Example:**

```
-- Good
BEGIN

   <<process_employees>>
   FOR i IN 1..t_employees.COUNT()

   LOOP

       …
   END LOOP process_employees;

END;
```

43. Use 1 as lower boundary and COUNT() as upper boundary when looping over a dense array.

**Reason:** Doing so will not raise a VALUE_ERROR when looping over an empty array. If you want to use FIRST()..LAST(), you need to check the array for emptiness.

**Example:**

```
-- Good
DECLARE
   t_employees  t_employee_type := t_employee_type();
BEGIN
   <<process_employees>>
   FOR i IN 1..t_employees.COUNT()
   LOOP
      …
   END LOOP process_employees;
END;
```

**Example:**

```
-- Good
DECLARE
   t_employees  t_employee_type := t_employee_type();
BEGIN
   <<process_employees>>
   IF t_employees IS NOT EMPTY
   THEN
      FOR i IN t_employees.FIRST()..t_employees.LAST()
      LOOP
         …
      END LOOP process_employees;
   END IF;
END;
```

**Example:**

```
-- Bad (raises a VALUE ERROR as FIRST and LAST return NULL)
DECLARE
   t_employees  t_employee_type := t_employee_type();
BEGIN
   <<process_employees>>
   FOR i IN t_employees.FIRST()..t_employees.LAST()
   LOOP
      …
   END LOOP process_employees;
END;
```

44. Always use a WHILE loop to process a loose array.

**Example:**

```
-- Good
DECLARE
   l_index PLS_INTEGER;
BEGIN
   l_index := t_employees.FIRST();

   <<process_employees>>
   WHILE l_index IS NOT NULL
   LOOP
      …
      l_index := t_employees.NEXT(l_index);
   END LOOP process_employees;
END;
```

45. Avoid using EXIT to stop loop processing unless you are in a basic loop.
[CodeXpert 4804]

46. Always use EXIT WHEN instead of an IF statement to exit from a loop.
[CodeXpert 4801] [CodeXpert 4802]

**Example:**

```
-- Bad
BEGIN
   <<process_employees>>
   LOOP
      ...
      IF ...
      THEN
         EXIT process_employees;
      END IF;
      ...
   END LOOP process_employees;
END;
```

```
-- Good
BEGIN
   <<process_employees>>
   LOOP
      ...
      EXIT process_employees WHEN (...);
   END LOOP process_employees;
END;
```

47. Try to label your EXIT WHEN statements.
    **Example:**

```
-- Good
BEGIN
    l_outerlp := 0;
    <<outerloop>>
    LOOP
        l_innerlp := 0;
        l_outerlp := NVL(l_outerlp,0) + 1;
        <<innerloop>>
        LOOP
            l_innerlp := NVL(l_innerlp,0) + 1;
            DBMS_OUTPUT.PUT_LINE('Outer Loop counter is ' || l_outerlp ||
                                ' Inner Loop counter is ' || l_innerlp);
            EXIT outerloop WHEN l_innerlp = 3;
        END LOOP innerloop;
    END LOOP outerloop;
END;
```

48. Do not use a cursor for loop to check whether a cursor returns data.
    **Example:**

```
-- Bad
DECLARE
    l_employee_found BOOLEAN := FALSE;
    …
BEGIN
    <<check_employees>>
    FOR r_employee IN c_employee
    LOOP
        l_employee_found := TRUE;
    END LOOP check_employees;
END;
```

```
-- Good
DECLARE
    l_employee_found BOOLEAN := FALSE;
    …
BEGIN
    OPEN c_employee;
    FETCH c_employee INTO r_employee;
    l_employee_found := c_employee%FOUND;
    CLOSE c_emplyoee;
END;
```

**49. Avoid use of unreferenced FOR loop indexes. (CodeXpert)**
[CodeXpert 4806]

**Reason:** The loop index is not 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 don't need the loop at all.

**50. Avoid hard-coded upper or lower bound values with FOR loops. (CodeXpert)**
[CodeXpert 4807]

**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) in a package specification and reference this named element instead of the hard-coded value.

# 5.5. Exception Handling

**51. Never handle unnamed exceptions using the error number.**

**Example:**

```
-- Bad
BEGIN
   ...
EXCEPTION
   WHEN OTHERS
   THEN
      IF SQLCODE = -2291
      THEN
         ...
      END IF;
END;
```

```
-- Good
DECLARE
   e_parent_missing EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_parent_missing,-2291);
   ...
BEGIN
   ...
EXCEPTION
   WHEN e_parent_missing
   THEN
      ...
END;
```

**52. Never assign predefined exception names to user defined exceptions.**
[CodeXpert 3010]

**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.

**53. Avoid use of WHEN OTHERS clause in an exception section without any other specific handlers. (CodeXpert)**
[CodeXpert 3001]

**Reason:** There isn't 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
EXCEPTION
   WHEN OTHERS
   THEN
      IF SQLCODE = -1
      THEN
         update_instead (...);
      ELSE
         err.log;
         RAISE;
      END IF;
```

```
--Good
EXCEPTION
   WHEN DUP_VAL_ON_INDEX
   THEN
      update_instead (...);
   WHEN OTHERS
   THEN
      err.log;
      RAISE;
```

## 54. Avoid use of EXCEPTION_INIT pragma for a -20,NNN error (CodeXpert)
[CodeXpert 3002]

**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. Oracle allocates 1000 error numbers, between -20999 and -20000, for users to use for their own application-specific errors (such as "Employee must be 18 years old" or "Reservation date must be in the future"). Define all error numbers and their associated messages in a database table or operating system file. Build a package that gives names to these errors, and then raise the errors using those names and not any hard-coded values. Here is a fairly typical hard-coded, error-prone program with RAISE_APPLICATION_ERROR. This procedure is designed to stop updates and inserts when an employee is younger than 18 and is based on the assumption that no one has used error -20734 yet:

**Example:**

```
--Bad
CREATE OR REPLACE PROCEDURE check_hiredate (date_in IN DATE)
IS
BEGIN
   IF date_in < ADD_MONTHS (SYSDATE, -1 * 12 * 18)
   THEN
      RAISE_APPLICATION_ERROR (
         -20734,
         'Employee must be 18 years old.');
   END IF;
END check_hiredate;
```

```
--Good
CREATE OR REPLACE PROCEDURE check_hiredate (date_in IN DATE)
IS
BEGIN
   IF emp_rules.emp_too_young (date_in)
   THEN
      err.raise (errnums.emp_too_young);
   END IF;
END check_hiredate;
```

**55. Avoid use of the RAISE_APPLICATION_ERROR built-in procedure with a hard-coded -20,NNN error number or hard-coded message. (CodeXpert)**
[CodeXpert 3003]

**Reason:** If you are not very organized in the way you allocate, define and use the error numbers between -20,999 and -20,000 (those reserved by Oracle for its user community), it is very easy to end up with conflicting usages. You should assign these error numbers to named constants and consolidate all definitions within a single package. When you call RAISE_APPLICATION_ERROR, you should reference these named elements and error message text stored in a table. Use your own raise procedure in place of explicit calls to RAISE_APPLICATION_ERROR. If you are raising a "system" exception like NO_DATA_FOUND, you must use RAISE. But when you want to raise an application-specific error, you use RAISE_APPLICATION_ERROR. If you use the latter, you then have to provide an error number and message. This leads to unnecessary and damaging hard-coded values. A more fail-safe approach is to provide a predefined raise procedure that automatically checks the error number and determines the correct way to raise the error.

**Example:** see rule 54

**56. Avoid unhandled exceptions**
[CodeXpert 3005]

**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.

**57. Avoid using Oracle's predefined exceptions (CodeXpert)**
[CodeXpert 3006]

**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.

# 5.6. Dynamic SQL

58. Always use a string variable to execute dynamic SQL.

**Reason**: Having the executed statement in a variable makes it easier to debug your code.

**Example:**

```
-- Bad
DECLARE
   l_empno emp.empno%TYPE := 4711;
BEGIN
   EXECUTE IMMEDIATE 'DELETE FROM emp WHERE epno = :p_empno' USING l_empno;
END;
```

```
-- Good
DECLARE
   l_empno emp.empno%TYPE := 4711;
   l_sql   VARCHAR2(32767);
BEGIN
   l_sql := 'DELETE FROM emp WHERE epno = :p_empno';
   EXECUTE IMMEDIATE l_sql USING l_empno;
EXCEPTION
   WHEN others
   THEN
      DBMS_OUTPUT.PUT_LINE(l_sql);
END;
```

59. Try to use output bind arguments in the RETURNING INTO clause of dynamic INSERT, UPDATE, or DELETE statements rather than the USING clause. **(CodeXpert)**
[CodeXpert 5814]

**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:**

```
DECLARE
    sql_stmt VARCHAR2(200);
    my_empno NUMBER(4)  := 7902;
    my_ename VARCHAR2(10);
    my_job     VARCHAR2(9);
    my_sal     NUMBER(7,2) := 3250.00;
BEGIN
    sql_stmt := 'UPDATE emp SET sal = :1 WHERE empno = :2
        RETURNING ename, job INTO :3, :4';

    /* OLD WAY: Bind values returned through USING clause. */
    EXECUTE IMMEDIATE sql_stmt
        USING my_sal, my_empno, OUT my_ename, OUT my_job;

    /* NEW WAY: Bind values returned through RETURNING INTO clause. */
    EXECUTE IMMEDIATE sql_stmt
        USING my_sal, my_empno RETURNING INTO my_ename, my_job;
END;
```

# 5.7. Stored Objects

60. Try to use named notation when calling program units.

**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**:

```
-- Good
BEGIN
    r_emp := read_employee(p_empno_in => l_empno
                          ,p_ename_in => l_ename);
END;
```

61. Always add the name of the program unit to its end keyword.
[CodeXpert 4404]

**Example**:

```
-- Good
FUNCTION get_emp (in_empno IN emp.empno%TYPE)
    RETURN emp%ROWTYPE
IS
BEGIN
    …;
END get_emp;
```

62. Always use parameters or pull in definitions rather than referencing external variables in a local program unit. **(CodeXpert)**
[CodeXpert 6404]

**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 makes a reference, however, to 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.

63. Always ensure that locally defined procedures or functions are referenced.
[CodeXpert 5603]

**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.

64. Try to remove unused parameters or modify code to use the parameter.
[CodeXpert 5406]

**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 in your code.

You should go through your programs and remove any part of your code that is no longer used.

## 5.7.1  Packages

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

66. Always use forward declaration for private functions and procedures.

67. Avoid declaring global variables public. **(CodeXpert)**
[CodeXpert 5202]

**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, 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" programs - 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.

68. Avoid using a IN OUT parameters as IN / OUT only. **(CodeXpert)**
[CodeXpert 5402][CodeXpert 5401][CodeXpert 5405]

**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.

## 5.7.2  Procedures

69. Avoid standalone procedures – put your procedures in packages.

70. Avoid using RETURN statements in a PROCEDURE. **(CodeXpert)**
[CodeXpert 5601]

**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.

### 5.7.3 Functions

71. Avoid standalone functions – put your functions in packages.

72. Try to use no more than one RETURN statement within a function.
[CodeXpert 3803]

73. Always make the RETURN statement the last statement of your function.
[CodeXpert 3801]

74. Never use OUT parameters to return values from a function.
[CodeXpert 3801]

**Reason:** A function should return all its data through the RETURN clause. Having an OUT parameter prohibits usage of a function within SQL statements.

75. Never return a NULL value from a BOOLEAN function.

### 5.7.4 Oracle Supplied Packages

76. Alwas prefix ORACLE supplied packages with owner schema name.

**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
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello World');
END;
```

```
-- Good
BEGIN
    SYS.DBMS_OUTPUT.PUT_LINE('Hello World');
END;
```

### 5.7.5 Object Types

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

### 5.7.6 Trigger

77. Avoid cascading triggers.

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

# 5.8. Patterns

## 5.8.1 Checking the Number of Rows

78. Never use SELECT COUNT(*) if you are only interested in the existence of a row.
[CodeXpert 5804]

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

**Example**:

```
-- Bad
...
BEGIN
   SELECT count(*)
     INTO l_count
     FROM cust_order
    WHERE …

   IF l_count > 0
   THEN
      SELECT p.part_nbr, p.name, p.unit_cost
        FROM part p
       WHERE …
```

```
-- Good
...
BEGIN
   SELECT p.part_nbr, p.name, p.unit_cost
     FROM part p
    WHERE EXISTS (SELECT 1
                    FROM cust_order co
                   WHERE co.part_nbr = p.part_nbr)
   …
```

## 5.8.2 Access objects of foreign application schemas

79. Always use synonyms when accessing objects of another application schema.
[CodeXpert 5810]

**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 doesn't 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
…
SELECT p.lastname
  INTO l_lastname
  FROM personal.persons p
 WHERE p.pnbr = p_pnbr_in
…
```

```
-- Good
CREATE SYNONYM rech_s_person FOR personal.persons
…
SELECT p.lastname
  INTO l_lastname
  FROM rech_s_person p
 WHERE p.pnbr = p_pnbr_in
…
```

# 6. Complexity Analysis

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

## 6.1. Halstead Metric

### 6.1.1 Calculation

First we need to compute the following numbers, given the program:
- n1 = the number of distinct operators
- n2 = the number of distinct operands
- N1 = the total number of operators
- N2 = the total number of operands

From these numbers, five measures can be calculated:
- Program length: $N = N1 + N2$
- Program vocabulary: $n = n1 + n2$
- Volume: $V = N \times \log_2 n$
- Difficulty: $D = \dfrac{n1}{2} \times \dfrac{N2}{n2}$
- Effort: $E = D \times V$

The difficulty measure D is related to the difficulty of the program to write or understand, e.g. when doing code review.
The volume measure V describes the size of the implementation of an algorithm.

## 6.1.2 CodeXpert

The calculation of the Halstead volume can be done using CodeXpert for example.

**Operators**

**The following are counted as operators within CodeXpert**

| If | fetch | < > |
| --- | --- | --- |
| else | open | < = |
| elsif | open for | > = |
| case | open for using | = |
| when | pragma | != |
| loop | exception | ; |
| for-loop | procedure call = 1 Operator | , |
| forall | function call = 1 operator | := |
| while-loop | begin end = 1 operator | . |
| exit | () = 1 operator | - |
| exit when | [] = 1 operator | + |
| goto | and | * |
| return | or | / |
| close | not | % |
| | like | < |
| | between | > |

**Operands**

**The following are counted as operands within CodeXpert**

| identifiers | numbers | characters |
| --- | --- | --- |
| strings | exception | < = |

**Analysis Result**

The table below shows the threshold values used by CodeXpert:

| Halstead Volume | Complexity Evaluation | |
|---|---|---|
| 0 – 1000 | Reasonable: | An average programmer should be able to comprehend and maintain this code. |
| 1001 – 3000 | Challenging: | More senior skills most likely required to comprehend and maintain this code. |
| > 3000 | Too complex: | Candidate for re-design or re-factoring to improve readability and maintainability. |

# 6.2. Cyclomatic Complexity (McCabe's)

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

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

## 6.2.1 Description

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

Mathematically, the cyclomatic complexity of a structured program is defined with reference to a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second (the control flow graph of the program). The complexity is then defined as: $M = E - N + 2P$

where

M = cyclomatic complexity

E = the number of edges of the graph

N = the number of nodes of the graph

P = the number of connected components.

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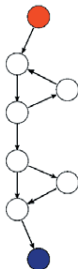


A control flow graph of a simple program. The program begins executing at the red node, then enters a loop (group of three nodes immediately below the red node). On exiting the loop, there is a conditional statement (group below the loop), and finally the program exits at the blue node. For this graph,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

complexity of the program is 3.

```
BEGIN
   FOR i IN 1..3
   LOOP
      dbms_output.put_line('in loop');
   END LOOP;
   --
   IF 1 = 1
   THEN
      dbms_output.put_line('yes');
   END IF;
   --
   dbms_output.put_line('end');
END;
```

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

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

Cyclomatic complexity may be extended to a program with multiple exit points; in this case it is equal to: $\pi - s + 2$

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

## 6.2.2   CodeXpert

A common application of cyclomatic complexity is to compare it against a set of threshold values. The calculation of the cyclomatic complexity can be done using CodeXpert for example.

**Analysis Result**

The table below shows the threshold values used by CodeXpert:

| Cyclomatic Complexity | Risk Evaluation |
|---|---|
| 1 – 10 | A simple program, without much risk |
| 11 – 20 | A more complex program, with moderate risk. |
| 21 – 50 | A complex, high risk program. |
| > 50 | An un-testable program with very high risk. |

# 7. Code Reviews

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

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

What can a code-review be good for?
- Code quality
- Code clarity and maintainability
- Quality of the overall architecture
- Quality of the documentation
- Quality of the interface specification

For an effective review, the following factors must be considered:
- Definition of clear goals.
- Choice of a suitable person with constructive critical faculties.
- Psychological aspects.
- Selection of the right review techniques.
- Support of the review process from the management.
- Existence of a culture of learning and process optimization.

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

# 8. Tool Support

## 8.1. Development

## 8.2. Documentation

| Tool | Supplier | Description |
| --- | --- | --- |
| PLDoc | SourceForge | |
| PLSQLDoc | Allround Automations | Plug-In for PL/SQL Developer |

## 8.3. Code Formatting

| Tool | Supplier | Description |
| --- | --- | --- |
| SQL and PL/SQL Formatter | ORACLE Faq's | Online formatting tool at www.orafaq.com/utilities/sqlformatter.htm |
| PL/SQL Tidy | | |
| Formatter Plus | Quest Software | Add-on to Toad and SQL Navigator |
| SQL Developer | ORACLE | Standard rules not configurable. |
| | | |

## 8.4. Unit Tests

## 8.5. Code Analyzers

| Tool | Supplier | Description |
| --- | --- | --- |
| CodeXpert | Quest | Supports <ul><li>rule enforcement</li><li>measurement of the number of statements</li><li>Halstead Volume (Computational Complexity)</li><li>McCabe's (Cyclomatic Complexity)</li><li>maintainability index</li></ul> |

# 9. References

### Bibliography

Feuerstein, S. (2007). *ORACLE PL/SQL Best Practices*. O'Reilly Media.

Quest Software, CodeXpert,

Trivadis AG P. Pakull, D. Liebhart. (2008). Modernisierung von PL/SQL und Forms Anwendungen.

UNSERE STANDORTE
**BASEL**
**BERN**
**LAUSANNE**
**ZÜRICH**
**DÜSSELDORF**
**FRANKFURT A. M.**
**FREIBURG I. BR.**
**HAMBURG**
**MÜNCHEN**
**STUTTGART**
**WIEN**

UNSERE LÖSUNGEN
- **BUSINESS INTEGRATION SERVICES**
- **BUSINESS INTELLIGENCE**
- **APPLICATION DEVELOPMENT**
- **INFRASTRUCTURE ENGINEERING**
- **MANAGED SERVICES**
- **TRAINING**

**VERSION 2.0**
**© 2011 TRIVADIS AG**

**Info-Tel. 0800 874 823 47**
**www.trivadis.com**

**trivadis**
makes **IT** easier.