Table of Contents

INTRODUCTION **3**
- UPDATES 3
- TERMINOLOGY 3

GENERAL DEVELOPMENT RULES **4**
- DON'T DUPLICATE 4
- TAKE PRIDE IN YOUR WORK 4
- MAKE TIME TO LEARN YOUR CRAFT 4
- MAKE TIME TO LEARN YOUR TOOLS 4
- KNOW WHERE TO GET ANSWERS 4
- GET ANOTHER PAIR OF EYES 5
- TEST YOUR CODE 5

DATABASE PROGRAMMING ENVIRONMENT **6**
- PROGRAMMING TOOLS 6
- SOURCE CODE CONTROL 6
- DBA TOOLS 6
- DATA MODELS AND DDL 6
- DATABASE DIAGRAMS 7

NAMING IDENTIFIERS **8**

SOURCE FILES **9**

STORED OBJECTS **10**
- INDEPENDENT FUNCTIONS & PROCEDURES 10
- PACKAGES 10
  - *Packaged Procedures* *10*
  - *Packaged Functions* *10*
  - *Package Initialization Block* *11*
- TRIGGERS 11
- MATERIALIZED VIEWS 11
- VIEWS 11
- JOBS 12
- SYNONYMS 12
- SEQUENCES 12

STYLE AND FORMATTING **13**
- PL/SQL OBJECT TEMPLATES 13
- CODE BEAUTIFICATION 13
- COPYRIGHT 13
- REVISION HISTORY 13
- GENERAL COMMENT RULES 13
- PACKAGE COMMENTS 13
- PROCEDURE/FUNCTION COMMENTS 14
- VIEW AND TRIGGER COMMENTS 14
- KEYWORD CASE 14
- INDENTATION 14
- SQL STATEMENTS 15
- MARGIN 15
- LISTS 15

WHITESPACE 16
EVALUATIONS AND EXPRESSIONS 16
CONTROL STRUCTURES 16

PL/SQL PROGRAMMING 18

DATA STRUCTURES 18
Datatypes 18
Literals, Constants and Global Variables 18
Variables 19
Collections 19
Records 19
ROUTINE CONSTRUCTION 19
Overloading 20
Control Structures 20
Cursors 20
Blocks and Labels 21
Actual Parameters 21
Formal Parameters 21
HANDLING THE INFAMOUS NULL 21
TRANSACTION CONTROL 22
BULK PROCESSING 22
INVOKER AND DEFINER RIGHTS 22
SQL OPTIMIZATION AND TIPS 22
Aliases 23
Accidentally Disabling Indexes 23
Full Outer Joins 24
Cursor Expressions 24
Table Functions 24
DYNAMIC SQL 24

EXCEPTION HANDLING AND LOGGING 26

WHEN TO HANDLE EXCEPTIONS 27
HOW TO HANDLE EXCEPTIONS 27
MESSAGE LEVELS 27
ASSERTIONS 27

PL/SQL DEBUGGING 28

PL/SQL TUNING 28

CONVENTIONS 29

DATES, TIMEZONE AND TIMESTAMPS 29

APPENDIX A: TEMPLATES 30

PACKAGE SPECIFICATION 30
PACKAGE BODY 30
PUBLIC OR PRIVATE ROUTINE COMMENT BLOCK 31
TRIGGER COMMENT BLOCK 31
VIEW COMMENT BLOCK 31

APPENDIX B: TOAD OPTIONS 32

Introduction

This is a repository of PL/SQL and SQL coding standards, conventions and guidelines. Adherence to the standard throughout the team or enterprise will yield much better work products, code that is higher quality, more robust, easier to test and less costly to maintain.

Refer to the rules in blue until they become a part of you and the peer reviews you perform. If your team decides a standard doesn't fit as written, feel free to change it. Just ensure that the decision is documented in here and communicated to the rest of the team.

Updates

For each new standard that requires a paragraph or more to explain or demonstrate, please attempt to distill the rule down into a single sentence. Place the single sentence above the paragraph that explains it, and highlight the sentence by using the "Rule" Style. This will allow a developer to quickly cover all the standards by just reading the blue rules, but preserve full explanations for those new to the environment or Oracle programming.

Terminology

In this context a *standard* is a generic rule that applies to the way in which all PL/SQL **must** be coded. The standards even apply to non-production PL/SQL and SQL, like unit tests, anonymous blocks and scripts. Generally a standard is indicated by the presence of the word "must", but can also be worded in the form of a direct command.

A *convention* is a common trend or approach that applies to an individual application or team. A convention documents "how we do things here", and may supersede a standard due to legacy issues and other constraints. A convention, like guidelines, is typically indicated by the presence of the word "should".

A *guideline* is expert advice on how to construct good database code. Nevertheless, unlike a standard, it can be ignored, or a better approach taken, without significant impacts. A guideline is typically indicated by the presence of the word "should".

When referring to standalone, or packaged procedures and functions collectively, this document will use the term *routines*. If a standard generically applies to both procedures and functions, but the context is singular, the term *routine* will be used. If a standard is mutually exclusive to functions or procedures, the explicit term *function* or *procedure* will be used instead.

General Development Rules

Don't Duplicate

- If you find a duplicate, modularize, centralize and share it. This applies to everything: variables, code blocks, algorithms, processing approaches, comments, deliverables, etc.

This is a widely accepted best practice within software development circles. It is often referred to as the DRY principle (Don't Repeat Yourself). This has many beneficial results and side effects, not the least of which is simple, elegant, easily read and maintainable code.

Take pride in your work

- If creating new code, care enough to do your very best work.
- If maintaining old code, leave it better than you found it (refactor).
- What you create becomes your legacy.

Managers and executives see developers as laborers or "resources". Some even go so far as to think of us as genuine engineers, professionals. But a programmer is more like an artisan than an engineer or line worker. Each coding problem can be solved in many ways. Each bug or business problem is another exciting challenge to apply logic, intuition, and creativity to arrive at the best solution. The more simple, elegant, clean, tight and flexible the solution is, the closer you are to being a true software artisan.

Leave a trail of excellence and contented customers in your wake, instead of software rot, death and destruction.

Make time to learn your craft

- Use the Internet, professional associations, conferences, Usenet groups, list servers, volunteer opportunities, training and books to hone your skills.

If you've got the passion and love the thrill of problem solving, this will come naturally. If you are bored; if your brain feels like it's shriveling; if you don't feel like work has given you anything new in the last year; find another line of work. Someone who loves the craft of software creation will find or create avenues to learn and teach when things get slow.

Make time to learn your tools

- Take 5-10 minutes every morning to explore a new feature in one of your tools.

Explore forgotten corners of your productivity and development software until you know your toolset inside and out, and the team starts to think of you as the "go-to" person because you seem to have all the answers and know the shortcuts that save hours of coding. For example, just learning how to use macros in your editor properly will save weeks of repetitive, mundane typing. Do not get stuck in a rut, unwilling to try a new feature or tool. There is always a better way.

Know where to get answers

- Don't struggle. Find the piece of reusable code, the expert, or the web page where your problem was already discussed and solved.

The best programmers often learn how to code well by reading someone else's masterpiece. They are productive problem solvers because their first line of defense is finding the wheel, rather than reinventing it. Although a really tough problem can be a fun challenge, if you'd like to finish your projects on time and have a life outside of work, get in the habit of consulting an expert resource first. The following resources are listed in order of usefulness (as rated by this author):

- Technical Lead

- AskTom
- Metalink Forums
- Metalink (All Sources)
- Peers
- Web search engines
- In-house Problem/Solution Database

Get another pair of eyes

- Do not struggle with a strange problem more than 30 minutes. When you hit an error or bug you just can't figure out, get another pair of eyes to look it over with you.

In the author's experience, this one rule has saved more time than any other technique tried. If it takes over 30 minutes, either 1) you've discovered a bug in a tool or in Oracle itself, which isn't likely, or 2) you're just too close to the problem. Usually the hardest bugs are small, like a missing comma or single-quote, extra semi-colon, etc. So swallow your pride and get someone else to look at it, even if they don't know the tool or language you're using. Nine times out of ten, in the first two minutes of either explaining the problem to them, or their glancing over your shoulder, the solution will just jump out at one of you. Try it. If nothing else, it fosters cooperation and humility among the team.

Test your Code

- Don't ever get so confident that you stop testing your code.
- Know your data.

If you want to be trusted and rewarded, your code should be nearly bug-free. Plan your functional tests before you write even the first line of code. Then write your interfaces. Test your interfaces before you code the implementations. Write the code, testing frequently as you get farther along. Always ensure you test with a representative sample of production data. Even better, create or generate a sample of littered, awful data that is much worse than production and full of unexpected errors and oddities. Finally, test in a performance environment that resembles production with a busy machine, concurrent users, and sufficient data volumes to ensure that your code scales and performs to plan.

Database Programming Environment

Check with your technical lead for access to licensed tools. Here we use <insert toolset here>, connecting to Oracle using InstanClient, and a modified sqlnet.ora and ldap.ora file to access OID for database connection resolution.

Programming Tools

For most database development, use PL/SQL Developer or TOAD. PL/SQL Developer is preferred. Install TOAD if you need to perform DBA tasks as well. An increasingly viable option is Oracle's new, free SQL Developer.

For heavy-duty editing, where advanced editing features are required, use PSPad, a powerful freeware editor. You may use your personal editor of choice as long as it doesn't hinder productivity or produce different results (some use tabs/spaces differently or are improperly configured for SQL files).

For FTP, again use your tool of choice, but I recommend free FileZilla. There is an adequate FTP client built into TOAD, and a FTP plugin for PL/SQL Developer, but FileZilla is much more powerful and easy to use.

For Unix/Linux shell access, use free PuTTY SSH.

For precise comparison tasks CompareIt! is recommended. There are many comparison tools in the marketplace, but this one is the best for the money. We did not find a freeware comparison tool that was adequate.

As a side note, FreeCommander is a fantastic replacement for Windows Explorer. And it is free.

Source Code Control

- All deliverables must be Added and Committed to Subversion (SVN).
- Always work from a versioned file. NEVER work from the stored object in the database.

Subversion for source code control is a great choice. Java developers have an Eclipse Subversion plugin. PL/SQL developers use TortoiseSVN as a windows client to Subversion.

Versioning is a proven and beautiful technique. It is your personal time machine and disaster insurance. Using it well can save hours to weeks of work in case hard drives crash, or decisions are reversed, or new modifications cause more problems than they solve, etc. You should be as comfortable with Subversion as you are with your favorite editor.

For existing code: Make modifications to the source file in your sandbox. Compile and test it. Ensure it follows standards and get it peer reviewed. Then commit your changes to SVN.

For new code, get it to an initial draft state, where it would take longer to rewrite it than to recover your hard drive, and then Add it to SVN. Commit your changes periodically -- perhaps daily -- as you complete large sections of code.

Remember to never modify the PL/SQL stored in the database. Work from the source code file instead. Yes, modifying the compiled code inside the database is technically feasible, but a really bad idea. It is comparable in some ways to modifying Java bytecode or C object files.

DBA Tools

For certain DBA-centric tasks, the physical DBAs use TOAD and/or Oracle Grid Control (OEM).

Data Models and DDL

- Most DDL is still created by hand. Make it clean, commented and readable, just like source code.

For modeling data structures, developers and designers design tables and views manually, usually with their PL/SQL editor, and then commit these scripts to SVN. Ideally you will have a great modeling tool in place like ER/Studio in which you can track changes and generate delta (changes) files to run against the database.

If you need to get DDL for an existing Oracle object, use PL/SQL Developer or TOAD to reverse engineer it. Another option is OEM (or DBMS_METADATA which OEM uses under the covers), but the DDL is not formatted as nicely, in which case you'll need to manually remove all the double-quote wrappers and then feed the DDL through a DDL formatter, like DDL Wizard.

Database Diagrams

PL/SQL Developer and TOAD have a built-in "poor-man's" diagramming tool. Use it for smaller diagrams ideal for quick white board design sessions and functional meetings. If a larger diagram is required, speak with someone who has access to the enterprise modeling tool and a plotter printer.

Naming Identifiers

See the "Oracle Naming Standard" document[1] for naming of database objects, including schema-level types.

PL/SQL identifiers must follow the scheme below. Print this out. Refer to this often until it becomes second nature.

Identifier Component Abbreviations

Direction		Scope		Type	Type Code
IN	i	Local	l	Object	obj
OUT	o	Global	g*	Record	rec
INOUT	io			Cursor	cur
				Ref Cursor (Cursor Variable)	rcur
				Associative Array (aka PL/SQL Table or Index-By Table)	arr
				Nested Table	tab
				Exception	exc
				ROWID / UROWID	rid/urid
				Timestamp	stmp
				Interval	intvl

* Use g as scope both for items that are truly public (declared in package spec) and those that are public only within the package body. Experience hasn't found much value in a naming scheme to distinguish the two shades of public access.

Identifier Syntax

Category	Syntax[2]	Examples
Parameters	{Direction}{_Name}[_Type*]	i_user_id, i_display_only, o_prop_rpt_dtl_rcur, io_ctrct_obj
Constants	{Scope}{**c**}{_Name}	gc_pkg_nm, gc_seconds_in_day, lc_prime_pct, lc_default_dir
Simple Variables	{Scope}{_Name}[_Type*]	l_mission_id, l_codemap_rec, l_created_stmp, g_debug_flg, g_local_tz, l_dept_obj, l_urid
Collection Variables	{Scope}{_Name}[_Type*][**tab**|**arr**]	l_pk_id_arr, g_state_cd_tab, l_state_list, l_lock_id_tab, l_updated_stmptab
Cursors and Exceptions	{Scope}{_Name}[_Type*]	l_ppi_data_cur, g_emp_contact_cur, g_record_locked_exc, l_item_missing_exc
Type & Subtype declarations	{Scope}{**t**}{_Name}[_Type*][**tab**|**arr**]	gt_ssn, lt_user_id_tab, lt_user_id_arr, lt_account_rec, gt_assembly_obj, t_bid_objarr, lt_contract_rectab

* Use optional [Type] for RECORD, REF CURSOR, OBJECT, TIMESTAMP, INTERVAL, and [U]ROWID identifiers.

In most cases, a well-named variable clearly indicates the underlying datatype, and therefore the allowed/required operations upon it. For this reason, the simpler datatypes of DATE, NUMBER and VARCHAR2 are not mentioned above as needing a special type code.

Composite variables can be composed of two type codes, like arrays of object, or nested tables of ROWID. In these cases, using the optional Type element is recommended, e.g. l_unit_config_objarr, l_ridtab, etc.

This scheme for Type and Variable declarations works well for the ordinarily complex scenario of setting up user-defined types, the collection type of the user-defined type, and the local variables (instances) of the collection type.

```
DECLARE
   SUBTYPE lt_ssn IS VARCHAR2(9);
   TYPE lt_account_rec IS RECORD (acct_id NUMBER, cust_id INTEGER, cust_ssn lt_ssn);
   TYPE lt_account_recarr IS TABLE OF lt_account_rec INDEX BY PLS_INTEGER;
   l_account_rec lt_account_rec;
   l_account_recarr lt_account_recarr;
   -- OR --
   l_accounts lt_account_recarr; -- Type Code and tab|arr are optional
BEGIN
```

[1] This document is part of the PL/SQL Starter Framework, found under the Standards subfolder.

[2] Items in bold are string literals. Items in [ ] brackets are optional. Items in { } brackets are required

Source Files

- Every PL/SQL stored object must be contained in a separate file.
- Package specifications and package bodies must be saved as separate files.
- The file name should be lowercase and match the name of the object it creates.

See the "Oracle Naming Standard" and the "Abbreviations and Acronyms" documents[3] for guidance on naming the Oracle objects within source files.

PL/SQL source files get their own special extensions, and should be versioned (kept in a version control system). It is recommended that your project's version repository directory have a structure similar to the following:

```
<workspace/working copy, e.g. C:\dev>
  \<versioned project name>
    \src
      \db
        \dba
        \ddl
        \dml
        \plsql
          \contexts
          \directories
          \java
          \jobs
          \mviews
          \packages
          \queues
          \triggers
          \types
          \views
        \tests
```

The recommended extensions for PL/SQL source code files are:

Stored Object Type	File Extension
Package Specification	pks
Package Body	pkb
Trigger	trg
Materialized View	mv
View	vw
Type Specification	tps
Type Body	tpb
DBMS_JOB submit	job
Queue Definitions	qd

DDL statements are stored in ".ddl" files and versioned under "src/db/ddl[/release]". These files include statements to create, drop and alter tables, columns, indexes, synonyms, sequences, directories, database links, schemas, tablespaces, libraries, profiles, clusters and permissions.

DML statements are stored in ".sql" files and versioned under "src/db/dml[release]". These are SQL statements and anonymous blocks that are meant to alter the contents of tables.

[3] These documents are part of the PL/SQL Starter Framework, found under the Standards subfolder.

Stored Objects

- Keep each stored object in its own dedicated source file. See Source Files above.

Independent Functions & Procedures

- Standalone functions and procedures are not acceptable, unless dictated by fine-grain security requirements. Group related procedures and functions into a package instead.

Standalone routines are not acceptable. There are only two exceptions. One is in conjunction with a function-based index, where a non-packaged, user-defined function is required. The other is when security specifications require individual routines so that each may be granted to very different lists of authorized users and roles.

Of course, you may write stand-alone routines for non-production scripts and test code.

Packages

- Start with the package templates and fill out the comment block completely.
- Each routine should do one thing, and one thing well.
- Use private routines to simplify and modularize otherwise complex and lengthy driving routines.
- End each routine or package with the routine/package name.
- Clean up your packages: Remove unused variables and constants. Revisit comments. Re-format. Add whitespace to improve readability. See Take pride in your work.
- Keep the use of global variables to within the package body. If the value must be read or set elsewhere, provide getter/setter routines.

It is a foregone conclusion nowadays that all PL/SQL belongs in packages. There are books written on this one subject and every guru in the industry agrees. If you feel you are up to challenging them, you may make your case in a public presentation to debate their wisdom at IOUG's Collaborate conference.

Test-driven and Test-first development instruct us to design and test the interfaces to the packaged routines before implementing them. When comfortable that the interfaces are complete and well though-out, then construct the bodies of the routines.

When feasible, design added flexibility in from the start. Provide table-based configuration, getter/setter routines, and overloaded versions to allow runtime modification to your packaged routines' behavior.

Ending the routine or package block with the block's name is a built-in feature of PL/SQL. Use it. For example:

```
PACKAGE arch_common IS
…
END arch_common;
```

Packaged Procedures

- Provide overloaded procedures when the number and type of the known inputs may vary.
- Centralize shared code between overloaded versions in private routines.

Ensure you are following the DRY principle. If you have overloaded routines that do similar things, ensure there is no copy/paste going on. Nothing should be duplicated. Take the shared code and refactor into a private routine, callable by both versions of the overloaded routine bodies.

Packaged Functions

- Use packaged functions to encapsulate getter routines that return one value: be it the result of an algorithm, a flag, a generated message, or a piece of fetched data.

- If OUT parameters are needed, then the function should be rewritten as a procedure.
- Each function should have only one RETURN statement in its logic.
- Ensure that binary functions are written to return only one of two possible answers, e.g. 0/1 Y/N or TRUE/FALSE. Do not allow them to return NULL or other unexpected results.

Package Initialization Block

Package initialization blocks are fairly rare in practice. They are most often used to avoid lots of IO by loading the lookup tables and frequently-used literals into session memory. The loading only happens once: the first time the package is called in a given session. Use this technique where the memory requirements of such structures are not too large, and where performance goals require it.

Triggers

- Start with the trigger template and fill out the comment block completely.
- Complex triggers should be refactored into packaged routines, and then called from the trigger.

  You will need to pass the :new and :old records to the routine in order for it to see all the columns. Since you can't pass the records directly, each column from :new and :old have to be assigned to %ROWTYPE records that **can** be passed. Use macros, templates or code generators to reduce the tedium of this step.
- All trigger bodies that are dependent on other triggers, or belong to the same trigger type (before vs. after and row vs. statement) should also be stored in the same trigger to control the order in which they execute.

  This is less of an issue on 11g where trigger firing order can be controlled, and where compound triggers are provided.
- Triggers should be used to enforce data integrity and business rules only after other options have been ruled out.

  Triggers are slow and sometimes tricky. Use them with caution and be sure to test their side-effects with production-like data, DML operations, user requests and transaction frequency.
- Instead-of triggers should anticipate and handle DML statements which accomplish nothing. Use %FOUND, %ROWCOUNT or RETURNING to determine and report DML efficacy.
- System event triggers should use the ORA_* public synonyms to determine what caused the system trigger to fire.

Note that MERGE is not yet valid as a triggering event. Instead, you must create triggers on the INSERT and UPDATE operations to which the MERGE operation decomposes.

Materialized Views

<To be completed>

Views

- Start with the view template and fill out the comment block completely.
- Use the FORCE keyword in the CREATE statement to avoid certain compilation sequence and privilege problems often encountered during migrations and upgrades.
- Use SET SQLBLANKLINES ON at the top of the .vw DDL script.

  This ensures that blank lines in the view or the view's comment block do not interrupt compilation.
- Do not include things like ROWNUM or analytics in your view unless necessary.

  Certain SQL constructs like ROWNUM and the new analytic functions prevent the CBO from merging the view, whereas it otherwise could merge it to improve performance.

Jobs

- Keep each call to dbms_job.submit in a separate .job file.
- If it is sensible to make the job ID static, make it a publicly accessible, well-named constant in a package specification.
- Keep all of the logic for a job in a packaged procedure that the job calls. Do not put more than a simple BEGIN <call> END; PL/SQL block in the WHAT parameter of the job.

The "what" column of the database's job tables is an astoundingly bad place to keep source code. The code is hidden from most GUI tools that have source code search features, and is hidden even from Oracle's own dependency tracking features. It is also very difficult to write scripts that must pass large amounts of code in the "what" parameter as every single quote has to be double-quoted, etc. Keep the code in a packaged routine and simply call that routine in the "what".

Synonyms

- Never use a database link directly in your code. Always decouple your code from database links using synonyms.

So instead of coding SQL directly against mytable@myremote, create a synonym named "mytable" defined as "SELECT * FROM mytable@myremote". If anything about the remote connection or description ever changes, you only need to modify the synonym definition online instead of having to offline systems and recompile code.

Sequences

- Use a separate sequence for each surrogate key.

Most sequences can be created with the default attributes. In other words, the MINVALUE, START WITH, CACHE, etc. if not specified, are all fine. However, for performance-intensive pieces of code, a larger CACHE should be investigated for possible improvements in throughput and response time.

A shared sequence that is used to fill the values of multiple surrogate keys is a really bad idea. I've seen it attempted and it caused various costly headaches a few years into the client's business.

Technically GUIDs are better to use a surrogate keys as they truly have no inherent intelligence or meaning behind them. But after attempting a small subsystem using GUIDs, at least 4 annoying drawbacks were found in practice. Stick with Oracle sequences to fill the surrogate key/primary key columns.

Style and Formatting

This section can seem pedantic and lengthy. Take heart: most of the rules in this section will be handled automatically and instantly if you make use of your PL/SQL IDE's templates and formatting features.

PL/SQL Object Templates

- Use your IDE's template feature when creating new stored objects. Fill in design information and fully comment items where indicated.

Replace your tool's default templates with those defined in Appendix A. Place the new templates on a network drive and have each team member either copy them to their hard drive, overwriting the tool's defaults, or point their tool to the network copies.

Code Beautification

- Use your IDE's formatter to format new and existing code.

If the default format provided by your PL/SQL IDE is not sufficient, modify it and ensure it is copied to, or shared by team members. If your IDE allows it, it is best to copy the beautifier/formatter settings file to a network share so that everyone can read the same copy and instantly inherit any team-wide changes made to it.

Copyright

- The standard copyright header must be included in all source code files, ideally near the top.

Contact your company's counsel to determine if your work products require a copyright placed in the source file's main comment block.

Revision History

- Include a revision history section in your source file comment block only if required by the project.

In most programming circles developers discipline themselves to include a revision history block within their source file's main comment block. This is good practice when no source code control system is available. It is redundant and error-prone in shops where systems like Subversion are in use.

Rely on your source code control/versioning system to let you know who originally created a file, who made modifications, what they did, and the differences between versions. Using Subversion to answer those questions is efficient and accurate.

General Comment Rules

- Only use comments that add value. Don't tell us what, tell us why.
- Use C-style comments (/* code */) to permanently comment-out large chunks of code.
- Use the double-dash style (- -) for all other comments, both standalone and inline.

  In PL/SQL IDE tools, entire blocks can be easily commented out with a single click or keystroke.

If you practice good variable and routine naming, your code will be self-documenting. We will be able to read your code and immediately tell "What" your code is doing. Comments should cover the "Why": things like business rules and requirements, design alternatives rejected, caveats, assumptions, etc. (things usually found only in the programmer's head or things poorly maintained by the business anywhere else).

Package Comments

- Each package specification and package body will have its own comment block, directly following the "CREATE OR REPLACE <package> AS" line.

The body comment block will only include the copyright. The specification comment block will include the copyright and a short description of what the package contains and/or its purpose. The majority of the design and implementation comments are reserved for the comment blocks of the individual packaged routines, not the package itself. The package is only a container.

See Appendix A for examples of package spec and body comment blocks.

Procedure/Function Comments

- The comment block directly **precedes** the routine's declaration.
- Keep the comments for a routine in one place.

  Each routine will have one comment block, in the package spec if public, in the body if private. Do NOT duplicate comments between the spec and body of public routines.
- Comment blocks for routines should be as descriptive as possible.

  This is the ONE area of documentation that stands a chance of actually being updated and kept in sync with the interface and implementation. So keep most business knowledge about a routine here, right alongside the code. Some routines will require more, some less.

  If your comments are extensive, consider using descriptive headers to separate each section from the next. The header will be on its own line, followed by the content. Please don't indent the content; comment formatting is too time-consuming to maintain as the content changes. Your technical reviewer should let you know if your comments are insufficient or difficult to follow.

  An example of a procedure/function comment block template: Click Here.
- Describe the purpose and allowed values of every unintuitive parameter. Use inline comments where possible; use the routine's comment block if more space is required.

  Parameters to public routines must be well commented, unless the parameter is completely obvious and self-documenting. Place the short comment to the right of the parameter declaration if there is space. If the comment extends beyond the right margin, place the parameter comment in the routine's comment block instead.

View and Trigger Comments

- View and trigger comment blocks must be placed at the end of the DDL script that creates them, directly before the semi-colon or slash that ends the object creation statement.

Views and triggers also get full comment blocks with the copyright and design notes. However, Oracle does two sneaky things here. Oracle strips comments from the top and within views before storing them in the data dictionary. And if triggers use a comment block at the top, they are stored in the data dictionary, but any exceptions raised from the trigger are reported at the wrong line number. For these reasons, the comment block for both views and triggers directly precedes the object's execution character at the **end** of the create script, be it a semi-colon at the end or a forward slash.

The templates in Appendix A handle this rule.

Keyword Case

- All built-in Oracle package names, functions, keywords and reserved identifiers will be UPPERCASE.
- All company-created identifiers will be lowercase. This includes table names, column names, view names, package and routine names, constants, variables and other identifiers.

Indentation

- Use 3 **spaces** as your standard indentation length and tab character.

  Due to the length of SQL keywords, three spaces work out really well for most SQL and PL/SQL. Three spaces has become the defacto standard within Oracle programming circles. Items within the declaration

sections and bodies of routines, as well as control structures, must be indented 3 spaces from the enclosing blocks.

Note, do not use standard tab characters. Toad, PSPad and most editors allow the tab character to be interpreted as three spaces. Code written with standard tabbed indentation looks awful when opened in editors that, by default, interpret tab characters as 4 or 8 characters (like vi on Unix, for example).

- Avoid deeply indented code.

  Deeply indented code (more than 3 levels) is very difficult to read and follow, unless the nesting and un-nesting all takes place on one visible page of code. Heavily indented code generally indicates that your routine is attempting to do too much. Try to break it out into modular routines.

SQL Statements

- SQL statements should be formatted to be easily readable and maintainable.

DML keywords should be right-aligned in SQL statements.

Commas should precede items in parameter and DML lists.

AND should precede each line in a WHERE clause, and OR should follow or separate WHERE expression options.

Here is an example of the style conventions mentioned above:

```
SELECT a
     , b
     , c
  INTO l_var1
     , l_var2
     , l_var3
  FROM my_tab
 WHERE (d = l_date
        OR
        f = i_cust_name)
   AND e = (SELECT e
              FROM hist_tab
             WHERE hist_dt >= TRUNC(SYSDATE - 7));
```

- All columns in SELECT and INSERT statements should be specified explicitly.

Using "SELECT *" or "INSERT INTO mytable SELECT…" (missing column list) are bugs waiting to happen the very first time someone adds or removes a column or rebuilds a table with different column order. Using explicit column lists maps table columns to waiting data structures, and won't be broken by modifications to the underlying table.

Margin

- Keep the width of your code within column 80 wherever possible.

  Wide load code is difficult to read on Windows terminals due to horizontal scrolling, and on printouts and Unix terminals due to wrapping. It won't be strictly enforced, but if your code or comments stretch beyond column 100 or so, either you've got too big a monitor, or you've never seen your code from someone else's perspective. Try printing it out. Isn't that awful? Using the formatter will handle this rule for you.

  Set your editor with a visible vertical marker at 80 as a reminder.

Lists

- Most lists should be formatted with one item per line (known as stacked lists), including parameter lists, DML, and DDL.

However, if the arguments to a built-in or user-defined routine can fit on one line within the standard margin, do so. Otherwise break it up into a stacked list. Most PL/SQL IDE formatters can handle this automatically.

Whitespace

- Use whitespace liberally.

  It takes up little to no memory in the compiled Pcode and it makes code much easier to read and digest.
- Use a one-character space on either side of an operator, i.e. ...i = 1, not ...i=1
- In comma-separated lists, there should be a space after every comma.
- In declarations, use at least two spaces between your variable and its datatype.

  Some developers like to indent and justify the datatype following parameter and variable declarations. If you are not using a formatter to do this for you, do not waste your time; instead, just put two spaces between the variable and its datatype.
- Use a full-line of dashes (80 of them) to visually separate routines within a package body.

Evaluations and Expressions

- All evaluations, even the most simple, single Boolean variable evaluation, should be enclosed in parenthesis.
- Mathematical evaluations will follow the regular rules of Algebraic expressions. Ensure that lengthy formulae use parenthesis around operations with a lower order of precedence so that the expressions are evaluated in the correct order.

The Boolean expressions within your evaluation statements frequently grow and change. Following these rules imposes order and rigor on your expressions and makes the code more readable. It also ensures that complex expressions are evaluated in the intended order. It is a best practice from several programming languages and a habit you should already be in from your Algebra days anyway.

Control Structures

- The LOOP keyword should be on the same line as the construct (WHILE, FOR) that started it.
- The THEN keyword should be on the same line as the IF/ELSIF that opened it.

Using the automated formatter will handle these rules for you. This rule makes it easier to visually spot the beginning and end of control structure blocks[4].

Instead of:

```
FOR...
LOOP
   ...
END LOOP;
```

Do this:

```
FOR...LOOP
   ...
END LOOP;
```

If the expression following the IF/WHILE/FOR etc. is too long for one line, break it up and make it readable, ending it with the enclosing right parenthesis and the LOOP or THEN keyword. Again, don't give the LOOP or THEN its own line.

[4] This is less of an issue now that Toad 9, PSD 7, KeepTool 7, and SQLdetective 3.4 can all now visually highlight and fold control blocks

When un-named blocks, IF statements, or loops END, follow each with a short comment identifying which control is coming to an end, e.g.

```
         END IF; -- whether item exists or not
      END IF; -- whether contract exists or not
   END LOOP; -- for each property
```


PL/SQL Programming

Data Structures

Datatypes

- Anchor parameters and variables using %TYPE (or %ROWTYPE for cursor or entire table).
- Use SUBTYPE to avoid hard-coded variable length declarations.
- Do not hard-code VARCHAR2 lengths for your constants and variables.

  Explicit VARCHAR2 lengths have a nasty habit of changing. Use anchoring and subtypes instead so the code isn't so fragile.
- Do not use CHAR as a datatype.

  The only exception would be a requirement that a field maintain a fixed length.
- Avoid implicit datatype conversions.

  Use the appropriate CAST or conversion function instead.
- Use CLOB, BLOB or BFILE for unstructured content. Do not use LONG or LONG RAW.

Literals, Constants and Global Variables

- Never hard-code literals. Abstract literals behind packaged constants.

  Make the new constants public in the package spec if another system or process may need to use them. Otherwise, make them global to the package body, or private to the routine (if the constant is highly specialized and never used anywhere else).

  Completely obvious and immutable literals, like 'Y' for Yes and 'N' for No, don't really need constants, but everything else does. The only other exception to the "never hard-code" rule is where a literal would make a significant difference in the speed of a SQL statement due to histograms and the infrequent use of the value across a large data set.

  A few system-wide constants are kept in the Starter Framework's CNST package. Become familiar with what is available there. Consult your team lead for the location of application-specific constants.
- Use a function to return a constant if it is possible that the "constant" value may change at times.

  However, if the "constant" could be used in SQL statements, use a regular constant instead to improve performance.

  If it changes enough, create a parameter in the Starter Framework's parameter tables instead and call PARM.get_val('*parameter name*') to retrieve the new flexible, table-driven "constant".
- Do not use public global variables. Get used to getters/setters and parameter-passing instead.

  Public global variables will rarely pass a code review except in cases with code related to SQL*Loader, dynamic SQL and mutating trigger solutions. Global variables declared in the body of a package however, can be useful in rare cases where state or lookup information is initialized once at the top of the session, read frequently, and rarely updated. This technique is used mainly for performance reasons, but produces highly coupled code and is a poor practice. Use it with care. In most cases, make each packaged routine independent of global items. Pass all the information that a routine requires into the routine via parameters.
- Never put all of a system's constants in a single package. Group related public types and constants in separate package specifications.

  All dependent packages are too tightly coupled and highly fragile if you put all your types, subtypes and constants in a single package. If you change or add even one thing to that single package spec, it invalidates everything. This creates a very fragile system that not even 11g's FGDT feature can solve.

Variables

- Unless the variable initialization is guaranteed success, always initialize variables right after the BEGIN statement, not in the declaration section.

  Exceptions raised by errors in the declaration section will bypass your carefully crafted EXCEPTION handler and bubble up at least one level to the caller. The message log will report an error coming out of the incorrect package/procedure.

- Give your variables a DEFAULT or initialized value if they have one.

  String variables can bypass this rule. They default to null. Number variables should not be allowed to default to null. They should all be initialized to 0 in the declaration section of your block. This avoids problems when NULL numbers are included in calculations (rendering the whole result NULL).

Collections

- Associative arrays[5] and nested tables are wonderful. Use them and bulk operations liberally.
- Do not use VARRAYs. They are useless.

  I've not seen a single instance where a nested table wasn't the better choice.

- Be careful when using a collection as the expression within IN or NOT IN lists.

  If any member of NOT IN list is NULL, the result is NULL. No error is produced, which is misleading.

- Always use aliases for the tables and nested tables when unnesting in SQL queries.

  Prevents known bugs (as of 9.2.0.1).

- Always use FIRST..LAST when iterating through collections with a loop.

  Avoid using 1..N or 1..collection.COUNT. Certain collections may be sparse and not start at 1.

- **Always** check your collection first to see if it has any values before working on it.

```
IF (l_arr IS NOT NULL and l_arr.COUNT > 0) THEN
```

  This avoids rare exceptions which are usually not handled or expected by the developer.

- The keyword COLUMN_VALUE is provided as the pseudonym of the column in a simple collection. Create an alias for this column when unnesting.

Records

- When passing large quantities of data between PL/SQL routines, use records, arrays of record, cursor expressions, pipelined tables and cursor variables.

  This renders the interfaces flexible and loosely coupled. Changes to the underlying data structures don't break the code. However, when communicating between the front-end and the backend PL/SQL routines, scalar variables, objects, collections and cursor variables are preferred since no 3GL supports Oracle's record datatype.

Routine Construction

- NEVER use Oracle schema names in your code.

  This makes your code location dependent. If your PL/SQL IDE is doing this to your reverse-engineered code without your permission, find and alter the tool preference that switches it off.

[5] aka index-by tables or PL/SQL tables

Overloading

- PL/SQL supports overloading. Please use it.

  If you have two routines that do very similar things, but take different parameters, use the same name between the two. Don't write "doThingA1" and "doThingA2". Instead write two versions of "doThing()" distinguished by the parameter list. This is similar to polymorphism in object-oriented programming. Take advantage of it.

Control Structures

- Use a simple LOOP or WHILE..LOOP when the number of iterations is unknown.
- Use a FOR loop when you already know, or can programmatically determine, the number of iterations.
- Make sure that conditions in IF-ELSIF statements are mutually exclusive and test all possibilities.

  Even if one of the branches is never expected, it is a best practice to show future maintainers of your code that you thought of, and tested, all the logic branches.

```
ELSIF (condition) THEN
   NULL; -- comment here to explain why this case is ignored
```

- Never EXIT from within any FOR loop.

  If you must use EXIT, your iterator should be a LOOP or WHILE loop instead.

- Never issue a RETURN statement inside a loop.

  Your module should have one point of exit. Use named labels if you have to, but always return at the end of execution, not before at various points within your routine body, and especially not within an uncompleted loop. The key to making this possible is a local variable that is assigned the result. Wait until the end of the function to return the local variable's value as the result.

- Make your decision points more self-documenting by replacing complex , multi-line expressions with simple Boolean variables or Boolean function calls.

Cursors

- When retrieving a guaranteed single-row result, e.g. (SUM, MAX, COUNT(*), etc.) feel free to use an implicit cursor, otherwise use explicit cursors.

  This includes cursor FOR loops. Declare the cursor explicitly and then call upon it in your loop. This is less of an issue since performance improvements were introduced in 9i. But it is still a good idea to declare more complex SQL statements as explicit cursors.

- Turn common queries on underlying tables & views into publicly accessible explicit cursors.

  With the advent of SYS_REFCURSOR, it is best to keep all your cursors protected in the body behind a routine that returns the cursor handle, or passes it back through an OUT parameter.

  This fulfills several best practices, including modularity, simplicity, maintainability and reusability.

- Use the elegant cursor FOR loop for all cursor access, unless you require high control over when the cursor closes.
- If you manually open your cursor, always explicitly close it, both in the body of your module and in any exception handling blocks or cleanup sections.
- Do not fetch into individual variables. Always fetch into a record whose type mirrors the cursor (defined with cursor%ROWTYPE).

  If you are using the cursor FOR loop, the declaration and definition of this cursor-based record is handled for you. The author often names this "lr" or "l_rec" for local record. Its scope is limited to the FOR loop, and the short identifier makes for easier typing.

Blocks and Labels

- Use of inner blocks should be limited to control exception handling and performance-sensitive code.

  Inner blocks are useful for trapping and handling expected exceptions, turning certain errors into non-events.

  Inner anonymous blocks (those with their own DECLARE section), are good to control when resource-intensive variables are allocated in memory, like large collections that are only needed during certain executions.

- You may use labels, but not in conjunction with GOTO. Do not use GOTO at all.

  If you want to use labels to assist in readability and organization, your routine is not yet modular. Break your routine up into several private routines that each does one thing well. Learn how to code in a modular manner such that GOTO is never needed. This yields cleaner, more robust and reusable code.

  If the now-compact routine still has several inner blocks or loops, label each inner block and loop. When ending a loop, tag the end with the loop label.

Actual Parameters

- Use positional notation for your actual parameter passing whenever possible.

  Named notation can be very useful for PL/SQL-based web applications and calls to routines that have DEFAULT NULL and NOT NULL formal parameters intermixed. Named notation is also very readable. However named notation isn't self-maintaining. If a parameter name changes, all named notation calls would also have to be changed. Positional notation does not have this problem.

Formal Parameters

- Explicitly declare each formal parameter's mode, i.e. IN, OUT and IN OUT.

  Although it doesn't hurt anything, just as you should explicitly close your cursors and explicitly cast your data, you should always explicitly declare your parameter modes. The default is IN. But don't leave your reader guessing; explicitly specify it is an IN parameter. Use IN OUT wherever you would have a data structure that could be altered by a call to the routine. The more modular you make your code; the more frequently you will see IN OUT parameters.

- Give your formal parameters DEFAULT values where they are known. Place parameters with defaults at the end of the parameter list.

  By placing all the parameters with default values at the end, it allows the routine to be called with the minimum number of actual parameters possible. If those with default values are scattered among those which are required, either named notation must be used to "skip over" those parameters with defaults, or values must be supplied for every parameter every time when using positional notation.

- Use the NOCOPY parameter mode hint with caution.

  It is a compiler hint, not a directive. The compiler may ignore you, which means unpredictable behavior and performance. Speak with your technical lead before using this feature.

Handling the Infamous NULL

- Explicitly handle the NULL case in your logic control statements. Don't let NULL surprise you in production.
- Always initialize your numerical values to 0 to avoid mathematical expressions yielding NULL.
- Be very careful that your code anticipates the NULL element anomaly of the NOT IN evaluation.

  A NULL value in a NOT IN list yields NULL. This is not true of NULL within an IN list.

Be aware that DECODE is able to compare an expression to NULL. CASE expressions cannot. CASE statements can.

Transaction Control

- Leave the COMMIT/ROLLBACK decision to the top-level caller.

Generally commits should not be explicitly specified. Savepoints, locking, explicit isolation levels and serializable packages will be used rarely as well. Speak with the technical architect if you feel the need to use those features.

Our standard can be called a "poor man's" transaction monitor, where the top-level caller (be it a Java class or PL/SQL block) acts as the monitor. Based on the absence or existence of raised exceptions from the lower call tree, the caller determines whether to pass the errors up another level, handle the error itself, commit, or roll back.

One obvious exception is autonomous transactions that require commit/rollback. Any routines that load data into application tables (as part of data migration or application initialization) would also be in charge of their own transaction control.

Be very cautious when using a COMMIT at any level below the driving routine, especially within a cursor that is being fetched from within a loop. This will result in a runtime error. In general, using COMMIT anywhere but at the uppermost layer of the application will not pass code review.

Also remember that any DDL operations embedded in dynamic SQL statements perform implicit commits. These can really mess up an application that depends on commits only occurring at expected times, like applications that use global temporary tables which delete data on commit.

Bulk Processing

- Use bulk processing wherever possible and appropriate.

  Empirical, in-house research on bulk processing has typically shown at least 60 – 80% performance improvements over row-by-row processing. Sometimes the improvement is 4 to 6 times and more. Consider every loop and cursor as a candidate for bulk processing.

- Look for DML that uses a loop to access elements within a collection. Rewrite using FORALL.

- Use collections in SQL sparingly. Ensure it is performance tested if you do.

  Past experience has taught the author to be very careful using collection nesting or unnesting in SQL statements (CAST, TABLE, COLLECT, MULTISET, etc.). The CBO still has a hard time with most of these operations. In most cases, a simple rewrite using an equi-join, subquery, WITH clause, or temporary table will perform better.

- Most uses of MULTISET are overkill

  If you see an existing use of MULTISET, where a scalar result set is being cast into a scalar collection, use BULK COLLECT instead. If you are on 10g, look into using COLLECT() for your nesting requirement.

Invoker and Definer Rights

The schema model of most typical database applications does not need invoker rights. If you feel the need to use invoker rights (the default is definer rights), speak with the technical lead first.

One exception is the Starter Framework packages which contain generic utilities that should operate based on the caller's privileges and access.

SQL Optimization and Tips

The following is a quick "checklist" of things to consider when coding SQL statements inside PL/SQL stored objects. This will help to optimize the wording of SQL statements and ease maintenance. The items are arranged in no particular order.

- Be aware of your tables' expected content, data volume and use within the application.

- Use bind variables (PL/SQL variables, constants and parameters). Do not hard-code values in a SQL statement, unless it can be show to improve performance in long queries on huge tables.
- Most IN subqueries can and should be rewritten as a join.
- Many IN and NOT IN subqueries will be much faster rewritten using EXISTS, as long as there is an index to support the correlated subquery in the EXISTS clause.
- Place the more selective tables and columns first in the FROM or WHERE clause.

  This doesn't make much difference to the CBO anymore, but it still makes your code easier to read and maintain.
- Avoid Native Dynamic SQL calls within performance-intensive routines.

  A soft parse is performed for every NDS call, which is kills performance in frequently-called routines.
- Avoid using DISTINCT.

  If a list is required, the DISTINCT performs better than the GROUP BY statement. The GROUP BY will add additional overhead to execution.
- Avoid calls to DUAL.

  This is less of an issue since the introduction of the optimized DUAL in 10g. If you are stuck with an older instance of Oracle, have your DBA create a faster DUAL substitute. There are various online articles on how to do this.
- Concatenated indexes should contain the columns most often used in common queries.
- Use LIKE "%…" with caution.

  Do not begin the pattern with the wildcard character "%". Realize if this is used you will probably harm the query's performance. The query may work great in the DEV environment where there is only one record, but will not scale when used in production.
- BETWEEN is more efficient than LIKE in the WHERE clause.

  <To be completed: Double-check this. This may have changed in 9i and 10g.>
- The most simple, robust method of determining DML success is by using the built-in cursor attributes SQL%FOUND and SQL%NOTFOUND.

  You may also use SQL%ROWCOUNT and the RETURNING clause to get further info about each INSERT, UPDATE and DELETE.
- Use the CASE expression or searched CASE statements wherever you used to use DECODE. DECODE is still perfectly acceptable, but not as elegant as CASE.

  CASE tends to be easier to read and more flexible than DECODE. So far, I have not found a reason to avoid CASE.
- Use the new MERGE statement instead of the traditional INSERT-DUP_VAL_ON_INDEX-UPDATE, or UPDATE-NO_DATA_FOUND-INSERT solutions.
- Use RETURNING to obtain information about modified rows. Do not re-query the same data.

Aliases

- Create an intuitive table alias for each table in a join. Avoid single-letter, unintuitive aliases.

All queries and DML that touch more than one table must use a short alias for each table. To promote consistency and instant familiarity, use the short table code found in the table comment (assuming the table was created per the standard). For example, instead of a, b, and c, use prop for PROPERTY, struct for STRUCTURES, and als for ALLOCATED_LCN_SPACE.

Accidentally Disabling Indexes

- Avoid full table scans, UNLESS your query is meant to access a large subset of the rows in a table

A full table scan is the better choice when retrieving all, or a large subset, of the rows from a table; an index lookup is the better choice when retrieving one row, or a small subset of the rows. Between these extremes, it is difficult to say when to use one method over another. However, full table scans can be faster and more efficient in certain cases, especially when accessing more than 15-30% of the table data.

To avoid an accidental full table scan, ensure that your WHERE clause is free of:

- Searches for NULL, e.g. WHERE column IS NULL.
- NOT EQUALS operators, e.g. != and <>
- DISTINCT
- Functions that operate on columns[6].

Full Outer Joins

In 8i and earlier, a full outer join of two result sets could be accomplished by using UNION to join the original outer-joined result with the reverse outer-joined result, allowing NULLs to appear on both sides of the join. The union operation eliminated the duplicate records from the overall resultset.

However, with the support for ANSI-standard joins in 9i and higher, I recommend, as does Oracle, the use of the FULL OUTER JOIN syntax. It is simpler to code, read, maintain AND it lifts certain restrictions imposed on Oracle's "(+)" outer join operations. See the 9i docs for further information.

Example:

```
SELECT t1.column1
      ,t2.column2
  FROM table1 t1 FULL OUTER JOIN table2 t2
    ON t1.join_column = t2.join_column;
```

Cursor Expressions

<To be completed>

Table Functions

<To be completed>

Dynamic SQL

- Wherever possible, bind variables to NDS strings. Do not concatenate values unless needed to identify schema or table/view.
- Do not use NDS in performance-critical routines.
- Format your NDS SQL strings as nicely as you would your regular SQL.

The various formatting tools treat NDS strings as normal string literals. They won't touch the format of the SQL inside these strings. Thus it becomes your job to ensure the SQL inside NDS strings is readable.

In general, using native dynamic SQL (NDS) will suit most instances where dynamic SQL is called for. Be aware that NDS involves hard parsing. So it is fine for run-once statements, but not for performance-intensive or

[6] There are two ways to avoid using a function on a column: store the data in a consistent format (e.g. upper case) or use function-based indexes.

frequently used code. Use regular dynamic SQL (DBMS_SQL) where performance is critical or where you run into NDS limitations.

Combining NDS with bulk processing and record-based DML is especially powerful. Study it carefully and attempt to apply it where it makes sense. It can lead to very flexible and re-usable modules that will greatly cut down on redundant and one-off code.

Exception Handling and Logging

- Use WHEN OTHERS only as a last resort.

  It is best to in most cases to let unexpected exceptions go unhandled. You get automatic rollback, better error point reporting and the same SQLCODE and SQLERRM that you would have obtained had you trapped it and handled it yourself.

  The catch-all OTHERS exception does have a place, but make sure to use SQLCODE and SQLERRM (DBMS_UTILITY.format_error_stack from 10g up) to capture what happened and re-raise, otherwise ALL errors, no matter how minor or severe, are masked and hidden from discovery and fix.

  Remember that the OTHERS exception MUST be the last one in the exception block.

- Always call cleanup code and logging code, if any, before allowing exceptions to raise.

  Remember that once an exception is raised, the normal flow of the program is interrupted. The exception bubbles up to the next exception handler (if any). If you placed cleanup or logging calls only in the main body of the block, it will be bypassed, as control will exit before it arrives at that point.

- Don't use OUT parameters or function return values to communicate success or failure codes/status to PL/SQL callers. Rely on the built-in exception handling mechanisms of PL/SQL.

- NEVER determine your own error message IDs (in -20999 to -20000 range) or use RAISE_APPLICATION_ERROR explicitly.

  Use the Starter Framework's logs.err, logs.warn, logs.info instead. I used to try to maintain a map of messages to their allowed ID in the -20999 to -20000 range. I no longer do. Instead, the framework just uses -20000 all the time, keeping all the truly relevant info in the error message.

- Use the Starter framework's logging features to automatically include error location metadata (caller, line number, timestamp, etc.) within error and debug logs. See the Starter's User Guide doc for more information on this feature.

Here are a few guidelines and PL/SQL rules and limitations you should keep in mind:

Read the Oracle PL/SQL User Guide, Error Handling section for a good explanation, examples and predefined exceptions.

Exceptions are built-in or user-defined. Exceptions are handled by name, not by error number. If it is an Oracle error, there may already be a predefined name for the error. If not, it is a good practice to use PRAGMA EXCEPTION_INIT to give the Oracle error a name, which makes your exception section much more readable and obvious what is being trapped.

Remember that if you are dealing with calling PL/SQL routines over database links, that PL/SQL cannot catch an exception raised by a remote PL/SQL routine. You will need to come up with another mechanism to let the caller know that something has errored out (probably an error OUT parameter this standard encouraged you to avoid earlier).

Since the exception block is just like a block of ELSIFs, it stops upon the first match. Therefore a named exception can only appear once in the exception block.

The maximum Oracle error message length (from SQLERRM) is 512 bytes. The maximum error message in RAISE_APPLICATION_ERROR is 2048 bytes. Use the newer DBMS_UTILITY.format_error_stack to obtain the full error message for display or logging.

All Oracle errors are negative except for no data found, which is 100 and user-defined errors, which are treated as a positive 1.

If a call to a stored object fails with an unhandled exception, no OUT parameters are passed back. An exception to this rule is the use of NOCOPY parameters. If a call fails with an exception, it would be best to throw away any results stored in NOCOPY parameters since they are probably incomplete or incorrect.

When to Handle Exceptions

The primary purposes of adding exception handling to PL/SQL programs are:

- To automate notification to support personnel when data, software or hardware needs fixing.
- To log context around errors or possible errors to aid in investigation and solution.
- To clean up opened/locked resources.
- To trap certain conditions and allow processing to continue instead of halting.

If you don't need to do one of those, don't add exception handling.

How to Handle Exceptions

When you, as the designer, have decided to add an exception handler, determine:

- Whether the exception you are catching is built-in (pre-defined) or whether you need to create your own (user-defined).
- What the exception's message level is.
- If there is any cleanup necessary (cursor closing, file closing, table-based state to reset, etc.)
- Whether you wish your process to continue after handling the exception, or to halt.

You should be familiar with Oracle's pre-defined exceptions to help you through the first decision. Also become familiar with the common exceptions defined in the Starter Framework's EXCP package specification. If they don't have the exception you need, create one in one of your application's package specifications.

The third decision is fairly simple. If you have resources you've tied-up, opened, locked or changed the state of, set them back or close them before re-raising the error.

The last decision is entirely up to the application designer. Some errors should halt the program; others should allow the program to continue. All Warning exceptions, however, should allow the program to continue. If you desire to continue processing, you will need to use inner blocks, wrapping the exception-generating call with its own BEGIN/END/EXCEPTION block. This will usually be found within a loop so that when the error is caught and logged, the processing may continue with the next iteration.

Message Levels

There are four types of messages used within our Starter Framework-based applications:

1. Error – for the most catastrophic errors where processing must cease immediately
2. Warning – for detected conditions that may or may not be an error, usually requiring human investigation to determine whether it is a problem or not.
3. Informational – for process logging and messaging to the end user, letting them know about detected conditions of interest, timings, etc.
4. Debug – for low-level debugging messages that provide context around each program step and call

These four message levels are enumerated in the Core CNST package as global constants. They are easily and transparently handled for you when calling the Starter Framework LOGS routines: logs.err, logs.warn, logs.info, and logs.dbg.

Assertions

- Use assertions to ensure assumptions are met and callers pass what you are expecting.

This method of programming is sometimes called "code by contract." When you code a routine, you provide an interface to consumers of the program's service. In order for the routine to function, certain parameters are required and must contain the right kind of data. In order for callers to successfully use your routine, they have to understand its requirements. There are all kinds of assumptions going on here, assumptions which could bring production to its knees if they are incorrect. If things aren't well documented, or if incoming data is dirty, or if the underlying implementation changed without notifying anyone, the caller or receiver is going to be sorely disappointed.

Thus it is a best practice to programmatically check assumptions through what are known as "assertions." The "assert()" routine is found in the Core EXCP package. Please read its documentation found in the package spec and

Starter's User Guide. Try it out. You will create code that is three times more robust that you've ever experienced before.

PL/SQL Debugging

PL/SQL has had a debugging API since 1999. Most PL/SQL IDEs have a GUI debugger. Get to know your debugger. It can save hours over the traditional trial-and-error investigation using added DBMS_OUTPUT and message logging. Triggers can be debugged with the IDE as well. Ideally your code will make heavy use of the Core LOGS.dbg() routine. If you have used it well and consistently, you won't even need to use your debugger. You will simply turn on debugging for a particular user, PL/SQL unit, process or stored object, and then monitor the debug data flowing into the APP_LOG table. See the Starter User Guide for more details on how to set up debugging and include it in new and refactored code work.

PL/SQL Tuning

A few cardinal rules of performance tuning are worthwhile to memorize and live by:

- If it ain't broke, don't fix it

  Focus on the code that is causing the greatest amount of pain. The only exception to this rule is code that is horrid, but the users have learned to accept its sluggishness. This is like being constipated and not knowing it. Surprise your users and remove that bottleneck.

- The fastest way to do something is to not do it at all.

  The least effective method of tuning is to tune the hardware. The next least effective is to tune the PL/SQL and SQL, which is where we typically spend the most time since managers often hope for a quick fix. The best method of performance tuning is to redesign the data model, business logic and/or application such that a performance bottleneck is altogether avoided or done in a completely different way. So the first question to ask when faced with a performance hotspot is "Do I really need to do this at all? Is there a way to accomplish this in another way that doesn't involve me tweaking the SQL?"

- The second fastest way to do something is in one fell swoop with a single SQL statement.

  Once one knows how to look for them, it is surprising how much sequential and row-by-row PL/SQL processing can be rewritten as a single SQL statement using joins, subqueries, correlation, analytics, vectors, CASE/DECODE, etc.

- The final option for quicker code is to do something in large chunks, taking advantage of multiple CPUs, massive memory and modern disk architecture.

  This rule can also be thought of as the "If you do it at all, do it big" rule. Generally, this means using array, or bulk, processing wherever possible. This can also include the use of parallel processing, table functions, partitioning and temporary tables where they make sense.

Conventions

Dates, Timezone and Timestamps

- Use explicit date/time formatting when using TO_CHAR on DATE values.

The author prefers this format for the date portion: YYYYMonDD as it clearly separates month from day or year, whereas formats like MMDDYY or YYYYMMDD are a just a bunch of numbers jammed together where it is unclear which two digits are the month and which are the day, e.g. 2010Oct09 is much clearer than 20101009 or 20100910.

Some tools like TOAD and SQL Navigator try to be too intelligent, allowing lazy developers to count on a default date format when they forget to use one. It is a good idea to set the NLS_DATE_FORMAT parameter to something useless like 'MONTH' so that developers who forget this rule are quickly schooled by the bugs that pop out when the database's default NLS_DATE_FORMAT is modified.

Appendix A: Templates

Package Specification

```
CREATE OR REPLACE PACKAGE mypkg
AS
/*****************************************************************************
                   Copyright YYYY by <Company Name>
                             All Rights Reserved.

<Brief description as to why the package was created, the category of routines
it contains, etc.>

<Optional design notes. Can include requirements, algorithms, alternatives
rejected, uses, dependencies, caveats, future plans etc.>

Artisan      Date      Comments
============ ========= ============================================================
devnameorid  YYYYMonDD Creation

*****************************************************************************/

-------------------------------------------------------------------------------
--                                PUBLIC CURSORS
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
--                                 PUBLIC TYPES
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
--                 PUBLIC CONSTANTS, VARIABLES, EXCEPTIONS, ETC.
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
--                               PUBLIC FUNCTIONS
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
--                               PUBLIC PROCEDURES
-------------------------------------------------------------------------------

END mypkg;
```

Package Body

```
CREATE OR REPLACE PACKAGE BODY mypkg
AS
/*****************************************************************************
                   Copyright YYYY by <Company Name>
                             All Rights Reserved.
*****************************************************************************/

-------------------------------------------------------------------------------
--                 PACKAGE CONSTANTS, VARIABLES, TYPES, EXCEPTIONS
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
--                         PRIVATE FUNCTIONS AND PROCEDURES
-------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
--                         PUBLIC FUNCTIONS AND PROCEDURES
--------------------------------------------------------------------------------

END mypkg;
```

Public or Private Routine Comment Block

```
/**----------------------------------------------------------------------------
myproc :
<Short synopsis of routine's purpose. Required.>
<Optional design notes. Can include requirements, algorithms, alternatives
rejected, uses, dependencies, caveats, example usage, etc.>

<Document unintuitive parameters here if inline space is not sufficient.>

----------------------------------------------------------------------------*/
```

Trigger Comment block

```
CREATE OR REPLACE TRIGGER trg_mytab_bi
BEFORE INSERT ON mytab
FOR EACH ROW
DECLARE
  -- local variables here
BEGIN

END trg_mytab_bi
/*****************************************************************************
                    Copyright YYYY by <Company Name>
                                All Rights Reserved.
<Short synopsis of trigger's purpose. Required.>

<Optional design notes. Can include requirements, algorithms, alternatives
rejected, uses, dependencies, caveats, etc.>
*****************************************************************************/
;
```

View Comment Block

```
SET SQLBLANKLINES ON

CREATE OR REPLACE FORCE VIEW myview
AS
SELECT
...
FROM mytab
/*****************************************************************************
                    Copyright YYYY by <Company Name>
                                All Rights Reserved.
<Short synopsis of view's purpose. Required.>

<Optional design notes. Can include requirements, algorithms, alternatives
rejected, uses, dependencies, caveats, etc.>
*****************************************************************************/
;
```

Appendix B: TOAD Options

For those with TOAD, the following settings will lead the reader through the standard setup in each screen of the formatter in order to match the formatting guidelines in this document. To see the options, from TOAD's main menu, select View | Formatting Options.

Formatter Options

General Layout

Tabs

Input Tab Size: 3

Insert spaces: selected

Output Tab Size: 3

Margins

Right Margin: 80

Indenting

Indent Size 3

Position the THEN: checked

Position the LOOP: checked

Position the IS: checked

Special Procedure Indenting: project specific; we leave this section alone.

Linefeeds

Every checkbox should be checked.

Header

Disable: selected

Case

Keywords

Uppercase: selected

User Dictionary: Currently working with Quest to get their default list to include all the missing keywords and reserved words as outlined in 10g docs.

Built-ins

Uppercase: selected

Built-in packages

Lowercase: selected

Other Identifiers

Lowercase: selected

User Dictionary: accept default list here for PL/SQL Web Toolkit coding. Add more as needed for special CamelCase treatment.

Lists and Operators

Variable declarations

Dynamic: selected

Parameter Declarations

Stack only on line overflow: selected

Dynamic: selected

Minimum space between names, modes and types: 3

Parameters

Stack only on line overflow: selected

Dynamic: selected

Minimum space between the association operator and the expressions: 1

Left align the association operator: unchecked

Parentheses

Space before parameter or column lists: selected

Leave a space between consecutive parentheses: unchecked

In stacked lists, the open parenthesis is on the first item's line: unchecked

In stacked lists, the closing parenthesis is on the last item's line: unchecked

Commas

Trailing: selected

AND – OR

Stack only on line overflow: selected

Operators left, all conditions aligned when stacked: selected

Remaining three checkboxes: all checked

Plus-Minus-Mul-Div-Concat

Stack only on line overflow: selected

Operators left, all expressions aligned when stacked: selected

Specific Statements

Assignments

Compact: selected

Wrap right hand side to new line rather than folding it: selected

SELECT/FETCH/EXECUTE

Keywords Right aligned: selected

Stacked (SELECT/INTO lists): selected

Stacked (FROM/GROUP BY/ORDER BY/RETURNING): selected

INSERT

Keywords Right aligned: selected

Stacked (COLUMN and VALUES lists): selected

Stacked (RETURNING and INTO lists): selected

UPDATE

Keywords Right aligned: selected

Stacked (RETURNING and INTO lists): selected

DELETE

Keywords Right aligned: selected

Stacked (RETURNING and INTO lists): selected

Comments

All boxes checked.

Comment Generation

7/8 Dependencies

Disable: selected

Recommendations

Disable: selected

Code Review Options

Leave defaults as-is for now.

Profiler/Dependency Options

Leave text fields as is.

