Oracle Development - Part III: Coding Standards

By Cheetah Solutions

Editor's Note: *In this final of a three-white-paper series on Oracle® Custom Development, Cheetah Solutions tackles the issue of coding standards. In their concluding white paper, Cheetah rounds out the development topic by offering best practices advise for Explain Plans, variable usage conventions, and quality control, to name a few.*

Introduction

This document is intended to serve as a blue print for standards requirements for coding and setting up customizations and modifications in Oracle.

File Naming Conventions

All custom objects should start with a meaningful prefix plus underscore at the start and so will the filenames as shown in Table 1.

File Type	Extension	Example
Stored Packages	pkb	PREFIX_<package name>.pkg
Anonymous block	sql	PREFIX_<testtrg>.sql
Table creation statement	tab	PREFIX_<table_name>.tab

Table 1: File Naming Conventions

Identifier Naming Conventions

Think of all identifiers as consisting of 4 parts:

- <Scope>
- <Type>
- <Primary Identifier>
- <Suffix>

By default, all variables are local. If they are global, they will be prefixed with the letter "g" as shown in Table 2.

Scope
Scope is the locality of reference. Knowing this is invaluable to the maintenance programmer. Notice that "p" is added as the scope of a parameter. This is an excellent method for denoting that a variable is a parameter to the procedure.

Locality	Description	Example
G	Global	g_temp
L	Local	l_temp
P	Parameter	p_temp

Table 2: Identifier Naming Conventions

Type

In addition to scalar types, there are other data types that are supported by the PL/SQL language. They are aggregate data types, which are listed in Table 3.

Type	Description	Example
c	Cursor	gcur_employee or c_employee
vcr	Cursor(variable)	vc_employee
tbl	Table	gtbl_employee
Rec or r	Record	r_address

Table 3: Aggregate Data Types

Primary Identifier

The primary identifier is the most important part of a name. It can be a single word or a phrase. We will talk of lengths of names later but it is always a trade off between length for documentation and brevity for typing purposes. The name should be optimal meeting both requirements. The name should tell the reader the purpose of the identifier. Some examples are account, student, company, phone, etc. We will later on discuss some abbreviating rules.

Suffix

The suffix is used to qualify the identifier further to document the usage of the variable. For example, the suffix is used to denote the type of parameter, as in IN, OUT, or INOUT as shown in Table 4.

Type	Description	Example
i	Input only parameter	pv_num_items_i
o	Output only parameter	pv_sum_o
io	Both input and output	pv_sum_io

Table 4: Suffix Examples

Variable Usage Conventions

Now that some basic standards are defined, let us look at how some of these standards are used in practice.

Cursor Declarations

Cursors are usually named after the table or a view that is being processed. Use the letter “c” as the prefix for the variable as shown in Table 5. You would still specify the scope of the variable as usual. What happens if you pass the cursor in as a parameter? You would end up with two suffixes. Although this is unusual, it works fine.

Scope	Type	Primary Identifier	Modifier	Suffix	Example
Local	cur	Account	New	Null	c_new_account

Table 5: Cursor Declarations

Record Based on Table or Cursor

Records are defined from the structure of a table or cursor as shown in Table 6.

Scope	Type	Primary Identifier	Modifier	Suffix	Example
Local	rec	Account		Null	r_account
Parameter	rec	Account		IN	pr_account_i
Global	rec	Account			gr_account

Table 6: Cursor Declaration Records

FOR Loop Index

Typical format should be:

```
FOR     r_emp     IN      c_emp              -- Record in cursor
OR
FOR     v_people_idx      IN      1..12      -- follows variable naming
                                             standards
PL/SQL table TYPE
```

Whenever possible, do not use PL/SQL table data type.

Programmer Defined Subtype

In PL/SQL subtypes can be defined from base data types as shown in Table 7.

Scope	Type	Primary Identifier	Modifier	Suffix	Example
Local	stp	Primary_key			stp_primary_key
Global	stp	Large_string			gstp_large_string

Table 7: Sub Types

```
SUBTYPE        lstp_primary_key     IS BINARY_INTEGER;
SUBTYPE        gstp_large_string    IS VARCHAR2;
```

Use sensible abbreviations for table and column aliases

Instead of a code segment such as:

```
SELECT  … select list …
  FROM employee A, Company B,  History C,  Bonus D, Profile E,  Sales F
 WHERE   A.company_id = B.company_id
   AND      A.employee_id = C.employee_id
   AND      B.company_id = F.company_id
   AND      A.employee_id = D.employee_id
   AND      B.company_id = E.company_id;
```

Use a code segment such as:

```
SELECT  … select list …
  FROM employee EMP, Company CO, History HIST, Bonus, Profile PROF, Sales
 WHERE   EMP.company_id = CO.company_id
   AND      EMP.employee_id = HIST.employee_id
   AND      CO.company_id = SALES.company_id
   AND      EMP.employee_id = BONUS.employee_id
   AND      CO.company_id = PROF.company_id;
```

Code Format

How you format your code in your source code is an intensely personal issue. Most people use conventions that are imposed by corporate standards. But when there is no standard available, then most programmers feel lost. They end up using a mish-mash of techniques that makes the resulting code hard to read. So it is important that every programmer develop a consistent and cohesive coding style that is easy to read and maintain.

There are two points of view to formatting. One is the developer's view. The other is the maintainer's view. A good standard should meet the needs of both views. There is really one fundamental reason for formatting your code: ***Reveal and reinforce the logical structure of your program.*** Writing code to please the eye is a waste of time. Code never stays that way for long. What is more important is to show the structure and the intent of the program. We truly believe that the machine should do this for the programmer. So if you follow the rules set forth here, there will be a tool in the future that will magically transform your program into a listing that could be framed as a work of art.

Indentation

Indentation is one of the most common and effective ways to display a program's logical structure. Programs that are indented are lot easier to read than those that are not. Be aware that indentation is a double-edged sword. It is very easy to mislead with inconsistent indentation.

General Indentation Rules

- Indent and align nested control structures, continuation lines, and embedded units consistently.
- Distinguish between indentation for nested control structures and for continuation lines.
- Use spaces for indentation, not the tab character.

Commenting Style

Functional comments always appear in code. They should appear in the pseudo-code in the header and should have corresponding comments in the blocks below. Custom object commenting should be MANDATORY and the reason for the custom object should be documented in the code.

Comment As You Code

- Explain Why - Not the How
- Maintain Indentation

Syntax Guidelines

1. Do not use “gotos”
2. Do not use conditional statements
3. Avoid unnecessary nested Ifs
4. Never Declare the FOR Loop Index
5. Avoid Unstructured Exits from Loops
6. Do not EXIT or RETURN out of a FOR loop.

The following statements are equivalent. The flat structure expresses the logic more clearly and with less code.

Nested

```
IF <condition1>
THEN
  ...
ELSE
  IF  <Condition2>
  THEN
     …
   ELSE
     IF  <Condition3>
     THEN
        …
      ELSE
        IF  <Condition4>
        THEN
           …
         END IF;
      END IF;
  END IF;
END IF;
```

Flat

```
IF <condition1>
THEN
  …
ELSIF  <Condition2>
THEN
   …
ELSIF <Condition3>
THEN
   …
ELSIF <Condition4>
THEN
   …
END IF;
```

If you have to use Nested IF..THEN..ELSE statements, then they should have identifiers that clearly establish the END IF and ELSE statements corresponding to each IF.

For example:

```
IF          <cond>                --  Check for existence condition 1
ELSE                              --            Else for condition 1
            IF          <cond2>   -- Check for existence condition 2
            END IF                --            End IF for condition 2
END IF                  --        END IF for condition 1
REPETITION
```

PL/SQL Programming Guidelines

Now that naming standards are defined, here are some general guidelines for good programming practices. Most of them are universal and would apply to any type of a programming effort. But we are only speaking in terms of PL/SQL here.

- Use named constants to avoid hard-coding values
- Convert variables into named constants
- Name subtypes to self-document code
- Remove unused variables from programs
- Use %TYPE when a variable represents a column
- SQL Guidelines
- Use sensible abbreviations for table and column aliases
- Add hints after you are done with the code and optimize to a reasonable degree

Naming Standards

- All value sets and flexfields, and custom programs should be prefixed.
- ALL custom tables should be created within a custom schema.
- All procedures should be created using the user id of apps.
- Access rights should be granted for the programs to be executed from apps.

The following GRANTS should be packaged along with the SQL:

- GRANT ALL to apps
- GRANT SELECT to IAMATOAD
- GRANT SELECT, INSERT, UPDATE, DELETE to EXPDEVL. This is not valid in all cases, depending upon if the table needs to be touched in production.

Packaging the Code

After the object is created in the custom schema, the appropriate synonyms for the object should be created in apps. Also, userids and passwords should be "accept"- ed in the same script. This would ensure proper packaging of SQL code.

QA Process

Commenting Example

Header
This should exist after the CREATE OR REPLACE command and not before the command.

Procedure:	get_users
Purpose:	This procedure populates the global variables g_transactionuserid and g_witnessuserid.
Tables Accessed:	users

Tables Modified:

Passed Variables:

p_transactionusername	The user performing the transaction.
p_witnessusername	The user witnessing the transaction.
p_callingprogramname	Program calling this program.
p_status	Status message to check for errors.
p_error_message	The actual error message.
p_debug	Debug flag for debug mode.

Modification History

Date	By	Reason and description of modification
05/16/02	John Doe	User wanted to add a new sales region automatically

Pseudo code:

```
{ Select userid for witnessname. }
{ Select userid for transactionusername. }
{ Exception }
```

Other things to consider

Please, no DBMS_OUTPUT.put_line statements; replace them with FND_FILES.

No Block comments – The old code should be stored as a backup and comments eliminated as much as possible.

ALL programs must have a header comment.

Exception handlers should exist for all blocks of code.

No SYSDATE or constants being returned from functions. Inoptimal code must be eliminated wherever possible.

EXPLAIN PLAN for all SQLs.

Look particularly for FULL TABLE scans on HUGE tables. Optimize as much as reasonable for user acceptance and use the guidelines mentioned in the Guidelines document. The reasonableness of code optimization will be determined as a part of the code review process.

Package the code. For example, if you have to log out as one user and log in as another user, then you should have accept commands in the script. In other words, the statements should be rightly packaged.

Eliminate using a lot of flags. Flags must have a proper reason for being used (useful for readers of your code – does not have any major performance impact).

Check if registration process is right.

Full table “Delete” statements should not exist; replace with a “Truncate” or “execute immediate ‘truncate …’ ” statement.

If there are multiple SQL statements that do INSERT, UPDATE, or DELETE statements with a COMMIT at the end of a transaction block, then the EXCEPTION for the transaction block MUST have a ROLLBACK.

For example:

```
BEGIN
        INSERT …
        UPDATE..
        INSERT
        COMMIT;
EXCEPTION
        ROLLBACK;          --- This is necessary
END;
```

All comments should have good functional description, in case the query needs more tuning.

Back Out Procedures

Always provide a description of what the procedure does and the objects that could get impacted

When doing explain plans, always convert the variable that is in the package to a bind variable. Oracle can evaluate a bind variable and a hard coded value differently.

While dropping tables inside a table creation package, CASCADE CONSTRAINTS should be used.

A grant script must exist with the code. This should not have GRANT ALL to any specific responsibility.

All reports must use the views and not the _all tables.

Inserts, updates, and deletes can use the _all tables depending on the requirement.

Temporary objects (tables, views, procedures, packages, functions, etc.) should be dropped after use.

The methodology to change a RULE hinted query back to a COST based query is as follows:
Perform an explain plan on the query with the RULE based hint.

Order the FROM clause according to the order of execution of the RULE based hint.
Insert the ORDERED hint into the SELECT clause.

Insert the INDEX hint for the innermost table and index of the RULE based explain plan.

Insert the USE_NL hint for all the other tables in the FROM clause.

All custom indexes should be created on separate tablespace.

Make sure that all CURSORs are closed both in the logic as well as in the EXCEPTION block. Also, make sure that transactions are handled properly both in the block as well as in the

EXCEPTION block. All files must be closed in the code and EXCEPTION block. For example: COMMIT in a loop, must have a corresponding ROLLBACK in the EXCEPTION block.

Conclusion

There's no getting around it and that is a good thing. If you are going to take the time to develop custom Oracle Application enhancements, then it is in your best interest to develop them for readability, maintenance and efficiency.