Top Twenty PL/SQL Tips and Techniques

Steven Feuerstein

Steven Feuerstein is the author of *ORACLE PL/SQL Programming* (O'Reilly, 1995) and *Advanced ORACLE PL/SQL: Programming with Packages* (O'Reilly, 1996). He is Director of the Oracle Practice for SSC, a systems management consulting firm based in Chicago (www.sarswati.com). Steven is also a Senior Technology Officer for RevealNet and co-author of the *Reveal for PL/SQL* knowledge server (www.revealnet.com). Steven won the Outstanding Speaker Award at ECO'96.

PL/SQL has come a long way since its rather simplistic beginnings as a batch-processing language for SQL*Plus scripts in the Oracle Version 6 database. Since PL/SQL Version 2 was first released, for example, Oracle Corporation has added dynamic SQL, RDBMS-based job scheduling, file I/O, and cursor variables. PL/SQL is now a mature, advanced programming language packed with features and capabilities and one of the most significant enabling technologies in the Oracle universe of products.

PL/SQL has also become a much more complicated and complex language. There is so much more for all of us to understand and, eventually, master. At the same time, the demands on our expertise and productivity increase steadily. As a result, there has never been a better time to pick up some tips for improving our PL/SQL programming skills.

This article offers twenty of my favorite tips (organized into categories) for writing PL/SQL code which has fewer bugs, is more easily maintained, and takes advantage of some of the most powerful features of PL/SQL. Stepping back from the details, however, these suggestions generally express the following good advice:

Use structured methodology.

You may not be coding in Cobol, but the same principles of design which worked for 3GLs work just as well with PL/SQL. Don't throw away the baby with the bathwater! Methodologies developed over the past thirty years hold their legitimacy even with the non-procedural environments like Oracle Developer 2000.

Use common sense.

At the end of many of my presentations on PL/SQL, people will come up to me and say: "But everything you said was just common sense!" I agree wholeheartedly. Certainly PL/SQL breaks some new ground (for many of us) with features like packages, PL/SQL tables, and cursor variables. Most of our time is, however, spent coding the same old IF statements, loops and so forth. We shouldn't have to be reminded to avoid hard-coding literals, to name identifiers to reflect their purpose, to use consistent indentation, to avoid side-effects in functions.

Do it right the first time.

I have got to be the worst offender of this advice. I fit the classic hacker profile, impatiently chomping at the bit to get in there and code. Hold yourself back! Do the analysis up front so that you understand the requirements fully. Think through the implementation approach you will take. Uncover the pitfalls before you start coding. And when you start debugging, don't "try" things. Analyze the problem and craft a solution you can prove will solve the difficulty. You will always end up being more productive if you take the time to do it right the first time around.

Make full use of the appropriate, provided features of the programming language.

I don't think there is a single person in this world (including members of the PL/SQL development team) who truly knows about every single feature of the PL/SQL language, especially if you include all of the package-based extensions. Yet the more you know, the more you can use to your advantage in your programs. Make sure you remain current with the ever-flowing releases of PL/SQL. Take the time to familiarize yourself with new features so that you can fully leverage the capabilities of PL/SQL.

Effective Coding Strategies

1. Use UPPER-lower case method to make code more readable.

PL/SQL code is made up of many different components: PL/SQL data structures such as PL/SQL tables, bind variables like Oracle Forms items, procedures, functions, loops, declarations, control constructs, etc. All of these elements break down roughly into two types of text: reserved words and application-specific identifiers. The PL/SQL compiler treats these two kinds of text very differently. You can improve the readability of your code greatly by reflecting this difference in the way the text is displayed. I employ the following UPPER-lower rule to highlight the distinction:

Enter all reserved words in UPPERCASE and all application identifiers in lowercase.

Using a consistent mixture of upper- and lowercase words increases the readability of your code by giving a sense of dimension to the code. The eye can more easily cruise over the text and pick the different syntactical elements of each statement. The uppercase words act as signposts directing the activity in the code. You can focus quickly on the lowercase words for the application-specific content.

There are other widely-used standards for use of case in programming, most notably the Visual Basic style of UsingInitCapInLongNames. Whichever you approach you take, make sure it aids in readability -- and use it consistently!

2. Use consistent indentation.

You should use indentation -- in a ruthlessly consistent manner -- to reflect the logical structure of your program. All of the following statements should cause indentation to another "level": IF statements, loops, sections of PL/SQL Block, and exception handlers.

Watch out for indentations which are too short to differentiate and too long to hold together related statements. I recommend an indentation of three spaces.

3. Avoid hard-coded literals of any kind in your programs.

Every application has its own set of special or "magic" values. These values might be configuration parameters or user information. They are constants throughout the application -- but that doesn't mean they never change. In fact, such values will almost always change eventually. To avoid getting burned by literals which have been littered throughout your code, follow these guidelines:

- Remove all literals (within reason) from your code. Instead, declare constants which hold those literal values.

4. Make sure your name describes the module accurately.

A name can make a big difference in the usability of your variables, modules and parameters. Your names should improve the self-documenting nature of your code. Follow these guidelines for procedures and functions:

- A procedure is an executable statement, a command to the PL/SQL compiler. Consequently, the grammar of the procedure name should be similar to a command: Verb_Subject.

- A function is used like an expression in an executable statement. It returns or "represents" a value, so the grammar of a function name should be a noun: Description_of_Returned_Value.

5. Test all of your assumptions as you debug your code.

Obvious, is it? Most of the time we aren't even aware of all the assumptions we make about our programs and the environment. I have often wasted hours trying to find the source of a problem because I muttered to myself "All right, I know that the value of that variable is set..." or something like that. And after I have tested everything else, I will finally say "Oh, what the heck! Might as check that variable." And lo and behold, that was my problem. Don't assume anything. Test everything.

Data Structures

6. Use anchored declarations whenever possible.

You can use the %TYPE and %ROWTYPE declaration attributes to anchor the datatype of one variable to that of a previously-existing variable or data structure. The anchoring data structure can be a column in a database table, the entire table itself, a programmer-defined record or a local PL/SQL variable. In the following example I declare a local variable with the same structure as the company name:

```
my_company company.name%TYPE;
```

In this second example I declare a record based on a cursor's structure:

```
CURSOR company_cur IS
 SELECT company_id, name, incorp_date
  FROM company;
company_rec company_cur%ROWTYPE;
```

Anchored types offer the following benefits:

1. Synchronization with database columns. Many PL/SQL variables "represent" database information inside the program. By using %TYPE, I am guaranteed that the local variable's data structure matches that in the database. If I instead hard-coded my declaration, the program could get "out of synch" with my data dictionary and generate errors. For example, if the previous declaration had declare my_company as VARCHAR2(30) and then I expanded the column size in the table to 60, my program is likely to cause VALUE_ERROR exceptions.

2. Normalization of local variables. You use PL/SQL variables to store calculated values used throughout the application. You can use %TYPE to base all declarations of these variables against a single, centralized variable datatype. If you need to modify that datatype, perhaps to expand the maximum size of a number to reflect higher revenue, you only need to change that single declaration. All anchored declarations will then pick up the new constraint when the programs are recompiled.

7. Streamline your code with records.

PL/SQL lets you create composite structures called records which can mimic the structure of a database table and a cursor. You can also define your own record structures with the record TYPE statement.

Records offer the following benefits:

- Write less code: instead of declaring, assigning and referencing individual variables, you can work the record as a whole.

- Protect your code from changes in database structures. Cursor and table records automatically adapt to the definitions of the columns in the underlying query.

Consider the program shown below. I wrote in while sitting in a coach airline seat. There was very little room to maneuver and so I wanted to minimize my typing. I was also unfamiliar with the details of the data structures, such as column names and datatypes. As a result, I worked almost exclusively with records and was able to focus on the logical structure of the program. I filled in the details when back on the ground.

```
FOR imp_rec IN imp_cur
LOOP
 /* Remove after compiles successfully! */
 IF imp_cur%ROWCOUNT > &1 THEN RETURN; END
IF;
 IF NVL (imp_rec.prft_ovrd_seq_nu, 0) = 0
 THEN
  get_site_prof_data (imp_rec, li_rec);
 ELSE
  get_provrd_data (imp_rec, li_rec);
 END IF;
 insert_line_001 (li_rec);
 insert_line_062 (li_rec);
 IF duplicate_pl_site (imp_rec)
 THEN
  create_new_override
   (imp_rec, li_rec, dup_count, new_pl_seq_nu);
 END IF;
END LOOP;
```

8. Optimize foreign key lookups with PL/SQL tables.

Given the normalized nature of our databases, you will have to perform many foreign key lookups (given this key, get the entity's name) in your screens and reports. You can use PL/SQL tables to cache already-retrieved values and thereby avoid repetitive SQL access. This will speed up your programs. You build the PL/SQL table incrementally with each query against the database. On subsequent calls, however, you check the table first and use that value. Here is the pseudo-code which describes the logical flow behind this approach:

```
1  FUNCTION company_name
2   (id_in IN company.company_id%TYPE)
3  RETURN VARCHAR2
4  IS
5  BEGIN
6   get-data-from-table;
7   return-company-name;
8  EXCEPTION
9   WHEN NO_DATA_FOUND
10   THEN
11     get-data-from-database-table;
12     store-in-PL/SQL-table;
13     return-company-name;
14 END;
```

9. Use Booleans to improve readability.

You can use Boolean variables and functions to greatly improve the readability of your programs. When you have complex Boolean expressions in IF statements and loops (the WHILE loop condition and the simple loop's EXIT statement), hide that complexity behind either a local PL/SQL variable or a call to a Boolean function. The name of the variable/function will state clearly a summation of all the complex logic. You can also more easily re-use that logic by hiding behind this kind of interface.

To get a feel for the improvement, compare the following two IF statements. In the first conditional statement, a raise is triggered by a complex series of Boolean expressions. It is hard to make sense of this logic and, even worse, it exposes the formula directly in this code. What if the business rule changes?

```
IF total_sal BETWEEN 10000 AND 50000 AND
  emp_status (emp_rec.empno) = 'N' AND
  (MONTHS_BETWEEN (emp_rec.hiredate, SYSDATE)
> 10)
THEN
  give_raise (emp_rec.empno);
END IF;
```

In this second IF statement, the details are hidden behind the eligible_for_raise function. As a result, the code is more readable and the business rule is encapsulated within the module.

```
IF eligible_for_raise (emp_rec.empno)
THEN
  give_raise (emp_rec.empno);
END IF;
```

Built-in Functions and Packages

10. Leverage fully the built-in functions.

PL/SQL offers dozens of built-in functions to help you get your job done with the minimum amount of code and fuss possible. Some of them are straightforward, such as the LENGTH function, which returns the length of the specified string. Others offer subtle variations which will aid you greatly -- but only when you are aware of those variations.

Two of my favorites in this category of hidden talents are INSTR and SUBSTR, both character functions.

SUBSTR returns a sub-portion of a string. Most developers only use these functions to search forward through the strings. By passing a negative starting location, however, SUBSTR will count from the end of the string. The following expression returns the last character in a string:

```
SUBSTR (my_string, -1, 1)
```

INSTR returns the position in a string where a sub-string is found. INSTR will actually scan in reverse through the string for the Nth occurrence of a substring.

In addition, you can easily use INSTR to count the number of times a substring occurs in a string, using a loop of the following nature:

```
LOOP
substring_loc :=
      INSTR (string_in, substring_in, 1, return_value);

/* Terminate loop when no more occurrences are found.
*/
  EXIT WHEN substring_loc = 0;

/* Found match, so add to total and continue. */
return_value := return_value + 1;
END LOOP;
RETURN return_value - 1;
```

11.Get familiar with the new built-in packages.

In addition to the many built-in functions provided by PL/SQL, Oracle Corporation also offers many built-in packages. These packages of functions, procedures and data structures greatly expand the scope of the PL/SQL language.

It is no longer sufficient for a developer to become comfortable simply with the basic PL/SQL functions like TO_CHAR and ROUND and so forth. Those functions have now become only the inner-most layer of useful functionality. Oracle Corporation has built upon those functions, and you should do the same thing.

Just to give you a taste of what the built-in packages offer consider the following possiblities:

DBMS_UTILITY.GET_TIME Function

Returns the ealpsed time in 100ths of seconds since an arbitrary time. You can use it to measure sub-second response time -- and also analyze the impact of your coding practices on application performance.

DBMS_LOCK.SLEEP Procedure

Blocks the current program from continuing execution for the specified number of seconds.

DBMS_OUTPUT.PUT_LINE Procedure

Display information to the screen from within your PL/SQL program. Use for debugging, for linking World Wide Web Home Pages to Oracle, etc.

DBMS_JOB.SUBMIT Procedure

Submit a job for scheduled execution by the Oracle Server. (PL/SQL Release 2.1 and above)

UTL_FILE.GET_LINE Procedure

Read a line of text from an operating system file. And use the PUT_LINE procedure to write text back out to a file. (PL/SQL Release 2.3 only)

The possibilities and capabilities aren't quite endless, but they are getting there! With each new release of the Oracle Server, we get new packages with which to improve our own programs.

Loops

12. Take advantage of the cursor FOR loop.

The cursor FOR loop is one of my favorite PL/SQL constructs. It leverages fully the tight and effective integration of the procedural aspects of the language with the power of the SQL database language. It reduces the volume of code you need to write to fetch data from a cursor. It greatly lessens the chance of introducing loop errors in your programming — and loops are one of the more error-prone parts of a program. Does this loop sound too good to be true? Well, it isn't — it's all true!

Suppose I need to update the bills for all pets staying in my pet hotel, the Share-a-Din-Din Inn. The example below contains an anonymous block that uses a cursor, occupancy_cur, to select the room number and pet ID number for all occupants at the Inn. The procedure update_bill adds any new changes to that pet's room charges.

```
1     DECLARE
2        CURSOR occupancy_cur IS
3           SELECT pet_id, room_number
4            FROM occupancy
             WHERE occupied_dt = SYSDATE;
5        occupancy_rec occupancy_cur%ROWTYPE;
6     BEGIN
7        OPEN occupancy_cur;
8        LOOP
9           FETCH occupancy_cur
               INTO occupancy_rec;
10          EXIT WHEN occupancy_cur%NOTFOUND;
11          update_bill
12             (occupancy_rec.pet_id,
                occupancy_rec.room_number);
13       END LOOP;
14       CLOSE occupancy_cur;
15 END;
```

This code leaves nothing to the imagination. In addition to defining the cursor (line 2), you must explicitly declare the record for the cursor (line 5), open the cursor (line 7), start up an infinite loop, fetch a row from the cursor set into the record (line 9), check for an end-of-data condition with the cursor attribute (line 10), and finally perform the update. When you are all done, you have to remember to close the cursor (line 14).

If I convert this PL/SQL block to use a cursor FOR loop, then I all I have is:

```
DECLARE
 CURSOR occupancy_cur IS
        SELECT pet_id, room_number
         FROM occupancy WHERE occupied_dt =
SYSDATE;
BEGIN
 FOR occupancy_rec IN occupancy_cur
 LOOP
        update_bill (occupancy_rec.pet_id,
occupancy_rec.room_number);
 END LOOP;
END;
```

Here you see the beautiful simplicity of the cursor FOR loop! Gone is the declaration of the record. Gone are the OPEN, FETCH, and CLOSE statements. Gone is need to check the %FOUND attribute. Gone are the worries of getting everything right. Instead, you say to PL/SQL, in effect:: "You and I both know that I want each row and I want to dump that row into a record that matches the cursor. Take care of that for me, will you?" And PL/SQL does take care of it, just the way any modern programming language integrated with SQL should.

13. Don't declare your FOR loop index.

Whether you use a numeric or cursor FOR loop, the loop index is declared for you and implicitly by PL/SQL. If you do declare a variable with the same name as the loop index, that will be a different variable. It will not be used by the loop and will likely introduce bugs into your program. Consider, for example, the anonymous block below. Even if I am the only person stored in the emp table, I will never get a raise!

```
DECLARE
  CURSOR emp_cur IS
   SELECT empno, ename FROM emp;
  emp_rec emp_cur%ROWTYPE;
BEGIN
  FOR emp_rec IN emp_cur
  LOOP
   display_emp (emp_rec.ename);
  END LOOP;
  IF emp_rec.ename = 'FEUERSTEIN'
  THEN
   give_raise
    (emp_rec.empno, 1000000);
  END IF;
END;
```

14. Avoid unstructured exits from loops.

You should follow these guidelines for terminating a loop:

- Let a FOR loop complete its specified number of iterations. Do not use an EXIT statement to leave prematurely.

- Always make sure you include an EXIT statement within a simple loop.

- Never use EXIT to leave a WHILE loop.

- Do not issue RETURNs directly from within a loop.

Modular Code

15. Construct abstract data types with PL/SQL packages.

The term "abstract data type" is about as dry and technical-sounding as you can get. Yet the concept of an abstract data type, or ADT, is something we apply -- or should apply -- in every single one of our application efforts, sometimes without even realizing that we are doing it.

An abstract data type is a collection of information and operations which act on that information. When you create an ADT, you work with objects as opposed to variables, columns, and other computer-science items. You perform an abstraction from the implementation details to the "thing in itself" and work on a higher level. In PL/SQL, this is best done with a package.

To give you a feel for the ADT, consider the following "before and after" examples of a thermometer implemented in Oracle Forms. The "before" shows how to manage the thermometer as a sequence of separate statements. The "after" shows the interface provided by the progress package.

1. Set the thermometer to 20% completion.

Before:

```
:B_PROGRESS.PERCENT_DONE := '20 % Complete.';
:B_PROGRESS.THERMOMETER := 'nn';
SHOW_VIEW ('cv_progress');
SYNCHRONIZE;
```

After:

```
progress.bar (20, 2);
```

2. Hide the progress box when the program completed and control was returned back to the user.

Before:

```
HIDE_VIEW ('cv_progress');
SET_WINDOW_PROPERTY ('PROGRESS_WINDOW',
VISIBLE, PROPERTY_OFF);
SYNCHRONIZE;
```

After:

```
progress.hide;
```

By treating the progress box as an object with rules governing its use and appearance, I can reduce greatly the volume of code required. The resulting statements are also easier to understand and maintain.

Here are some guidelines you should follow when designing an ADT:

1. Maintain a consistent level of abstraction. This is probably the most important aspect of your ADT implementation. All the modules you build to represent your abstract data structure should operate with the same level of data.

2. Provide a comprehensive interface to the abstract data type. Make sure that the user (a programmer, in this case) can perform all necessary operations on the ADT without having to go around the interface you build to the ADT. Hide all the implementational details of your ADT behind calls to procedures and modules -- without exception.

3. Use the package structure, the most natural repository for ADT code. The ADT represents a thing by presenting a layer of code which allows you to perform operations on that thing as a whole, rather than its individual components. The package joins related objects together and so corresponds closely to the ADT. The package clearly distinguishes between the public and private parts of the code. The public objects make up the interface to the ADT. The private objects contain and hide the implementational details for the ADT.

16. Enhance scope control with nested blocks.

```
PROCEDURE delete_details
IS
BEGIN
  BEGIN
   DELETE FROM child1 WHERE ...;
  EXCEPTION
   WHEN OTHERS THEN NULL;
  END;
  BEGIN
   DELETE FROM child2 WHERE ...;
  EXCEPTION
   WHEN OTHERS THEN NULL;
  END;
END;
```

I can in this way use my nested blocks to allow my PL/SQL program to continue past exceptions.

17. Overload modules to make your software smarter.

Within a package and within the declaration section of a PL/SQL block, you can define more than module with the same name! The name is, in other words, overloaded. In the following example, I have overloaded the value_ok function in the body of my check package, which is used to validate or check values used in my application:

```
PACKAGE BODY check
IS
/* First version takes a DATE parameter. */
FUNCTION value_ok (date_in IN DATE)
  RETURN BOOLEAN IS
BEGIN
      RETURN date_in <= SYSDATE;
END;
/* Second version takes a NUMBER parameter. */
FUNCTION value_ok (number_in IN NUMBER)
  RETURN BOOLEAN IS
BEGIN
      RETURN number_in > 0;
END;
END;
```

Now I can put both versions of value_ok to work in my code as follows:

```
IF check.value_ok (hiredate) AND
  check.value_ok (salary)
THEN
...
END IF;
```

I have found overloading to be extremely useful when I am building a layer of code which will be used by other developers (my PL/SQL toolbox). I use module overloading to hide complexities of the programmatic interface from my users (other programmers). Instead of having to know the six different names of procedures used to, for example, display various kinds of data, a developer can rely on a single module name. In this fashion, overloading transfers the burden of knowledge from the developer to the software.

18. Use local modules to reduce code volume and improve readability.

A local module is a procedure or function which is defined in the declaration section of a PL/SQL block (anonymous or named). This module is considered local because it is only defined within the parent PL/SQL block. It cannot be called by any other PL/SQL blocks defined outside of that enclosing block.

There are two key reasons to create local modules:

- Reduce the size of the module by stripping it of repetitive code. This is the most common motivation to create a local module. The code reduction leads to higher-quality code since you have fewer lines to test and fewer potential bugs. It takes less effort to maintain the code, since there is simply less to maintain. When you do have to make a change, you make it in one place in the local module and the effects are felt immediately throughout the parent module.

- Improve the readability of your code. Even if you do not repeat sections of code within a module, you still may want to pull out a set of related statements and package them into a local module to make it easier to follow the logic of the main body of the parent module.

Consider the following example, in which I calculate the net present value for various categories and then format the result for display purposes.

```
PROCEDURE display_values
(proj_sales_in IN NUMBER, year_in IN INTEGER)
IS
BEGIN
DBMS_OUTPUT.PUT_LINE
      ('Total Sales: ' ||
       TO_CHAR((npv ('total_sales', year_in) /
                          proj_sales_in * 100),'999.99'));
DBMS_OUTPUT.PUT_LINE
      ('Employee Labor: ' ||
       TO_CHAR((npv ('labor_costs', year_in) /
                          proj_sales_in * 100),'999.99'));
END;
```

In this approach, I have exposed the way I perform the calculation and formatting. As a result, whenever a change is required (different display format, different formula, etc.) I must upgrade each distinct calculation. If, on the other hand, I hide the calculation behind the interface of a callable module, then the calculation is coded only once.With the help of a local module, the display_values procedure is transformed as shown below.

```
PROCEDURE display_values
(proj_sales_in IN NUMBER, year_in IN INTEGER)
IS
/*------------------- Local Module -----------------*/
  PROCEDURE display_npv (column_in IN VARCHAR2)
IS
  BEGIN
       DBMS_OUTPUT.PUT_LINE
                (INITCAP (REPLACE (column_in, '_', ' '))
|| ': ' ||
                 TO_CHAR((npv (column_in, year_in) /
                                                  proj_sales_in
* 100), '999.99'));
  END;
BEGIN
 display_npv ('total_cost');
 display_npv ('employee_labor');
END;
```

I have found that few developers are aware of the ability to create local modules. I have also found that modules-within-a-module play an important role in allowing me to write well-structured, even elegant programs. Take a look at any of your more complex programs and I guarantee you will quickly identify segments of the code which would serve you better bundled into a local module.

19. Code a single RETURN for successful function execution.

The sole purpose of a function should be to return a single value (this could be a PL/SQL table or other composite structure, of course). To reinforce this single-mindedness, you should code your function so that it has only one RETURN statement. This RETURN should also be the last statement in the function. I use the following template to achieve this effect:

```
FUNCTION fname () RETURN datatype
IS
 return_value datatype;
BEGIN
 RETURN return_value;
END fname;
```

20. Use Self-identifying Parameters (Avoid Boolean Values)

A common parameter for a module is a flag which relies on two-valued logic (TRUE or FALSE) to pass information to the module. The temptation in such a case is to declare the parameter to be type Boolean (or to simulate a Boolean with a VARCHAR2 'Y' or 'N' value). With this approach, when you call the program, you will pass either TRUE or FALSE. You will find in these circumstances that while the specification for your program is very readable, the way you call it in other programs will be difficult to understand.

Consider the parameters for the following procedure, which is used to generate reports. It offers flags to control report printing and file clean-up.

```
PROCEDURE gen_report
(report_id_in IN NUMBER,
 clean_up_in IN BOOLEAN,
 print_in IN VARCHAR2);
```

In order to make gen_report more flexible, the developer has provided two flag parameters:

clean_up_in	TRUE if the procedure should clean up log files, FALSE to keep the files in place, usually for purposes of debugging.
print_in	Pass 'Y' if the output file of the report should be printed, 'N' to skip the print step.

When one glances over the procedure's specification, the purpose and usage of each parameter seems clear enough. But take a look at how I would call this procedure:

```
gen_report (report_id, TRUE, 'Y');
```

As you can see, these arguments are not very descriptive. Without the context provided by their names, actual Boolean parameters cannot "self-document" their effect. A maintainer of the code must go back to the source to understand the impact of a particular value. That defeats completely the information hiding principle of modular programming.

A much better approach replaces Boolean and pseudo-Boolean parameters with character parameters whose acceptable values are descriptions of the action or situation. With this change, a call to gen_report states clearly its intentions:

```
gen_report (report_id, 'CLEANUP', 'PRINT');
or:
gen_report (report_id, 'NO_CLEANUP', 'PRINT');
```

As you can see from these examples, I write my procedures to accept an unambiguous affirmative value ("CLEANUP") and the most clearly-defined negative form ("NO_CLEANUP"). One complication to this style is that you need to validate the parameter values; if you were using a Boolean, the strong datatyping in PL/SQL would guarantee that a legal value is passed. An even better solution than either of these approaches is to use constants which are stored in a package. In this way, you avoid hard-coding values, get compile-time verification of correctness and make your code more readable.

The package might look like this:

```
PACKAGE vals
IS
  c_print CONSTANT VARCHAR2(5) := 'print';
  c_noprint CONSTANT VARCHAR2(7) := 'noprint';
  c_cleanup CONSTANT VARCHAR2(5) := 'cleanup';
  c_nocleanup CONSTANT VARCHAR2(7) :=
'nocleanup';
END vals;
```

And the calls to gen_report would now look like this:

```
gen_report (report_id, vals.c_clean, vals.c_print);
```

This paper should provide you with lots of ideas on how to improve your code. Of course, it is just a tip (no pun intended) of the iceberg of features and functionality in the PL/SQL language. Oracle PL/SQL Programming, published by O'Reilly and Associates, explores all of the tips in this article, plus many, many more, in almost overwhelming detail. If you have any questions, you can reach me at my Compuserve address: 72053,441.

e·c·o 97 ALL-STAR Training
LEARNING FROM THE BEST

Registration

Please print clearly!

NAME: _____

COMPANY: _____

ADDRESS: _____

CITY: _____ STATE/PROVINCE: _____

ZIP/POSTAL CODE: _____ COUNTRY: _____

TELEPHONE: _____ FAX: _____

E-MAIL ADDRESS: _____

(CLASS SIZE IS LIMITED. DON'T DELAY.)		(✓)	**FEE PER PERSON**	Please make a copy of this form and submit a completed form for each individual
Registration Fee for Both Days **January 30 & 31**	User Group Member		$450	
	Non-User Group Member		$475	
Registration Fee for One Day Only **January 30 or January 31** **(Web Server Training & Designer/2000 Methods and Methodology are 2-day classes)**	User Group Member		$225	
	Non-User Group Member		$250	

User Group Affiliations: _____

Participants paying for both days of the training classes should choose one class for each day.
Participants paying for only one day should make one selection for the day they plan to attend.
Participants selecting *Developing Applications Using Oracle WebServer* or *Designer/2000 Methods and Methodology* will receive two days of training. (This class will be your only training class selection).

Thursday and Friday, January 30 and 31, 1997

THESE ARE TWO-DAY CLASSES

☐ Developing Applications Using Oracle WebServer – Jeff Bernknopf
☐ Designer/2000 Methods and Methodology – Paul Dorsey and Peter Koletzke

Thursday, January 30, 1997

THESE ARE ONE-DAY CLASSES

☐ Best Practices and Advanced Techniques in PL/SQL – Steven Feuerstein
☐ Advanced Techniques for Oracle7 Database Administration – Noorali Sonawalla

Friday, January 31, 1997

THESE ARE ONE-DAY CLASSES

☐ Leveraging PL/SQL Packages – Steven Feuerstein
☐ Generating WebServer Applications from Designer/2000 – Paul Rodgers
☐ Multidimensional Modeling and Designing a Data Warehouse – Mark Whitney

On-Site Registration is $300 per day for ALL participants. (both members and non-members)

All-Star Training Cancellation Refund Policy:

Cancellations must be received prior to January 27, 1997, for a full refund less $100.
If you don't cancel and don't attend, no refunds will be given.

Payment Method

Payment must accompany this form. Payment may be made by check (payable to Oracle User Resource), purchase order, or credit card authorization.

Check/Money Order (U.S. Dollars only) ☐ Purchase Order # _____

Credit Card: AMEX ☐ Visa ☐ MasterCard ☐

Card # _____

Billing Zip Code _____ Exp. mm/yy: _____ / _____

_____
CARD HOLDER'S SIGNATURE

_____
PRINT CARD HOLDER'S NAME AS IT APPEARS ON CARD

MAIL PAYMENT TO: **ECO'97**
1417 S.College Road · Wilmington, NC 28403
Phone: (910) 452-0006 · Fax: (910) 452-7834 · E-mail: 71172.644@compuserve.com

APRIL 6-9, 1997

Boston Park Plaza Hotel
Boston
Massachusetts

PLEASE TYPE OR PRINT CLEARLY

NAME: _____
(FIRST) (LAST)

COMPANY: _____

ADDRESS: _____

CITY: _____

STATE/PROVINCE: _____ ZIP/POSTAL CODE: _____

COUNTRY: _____

E-MAIL ADDRESS: _____

TELEPHONE: _____ FAX: _____

CONFERENCE REGISTRATION

Full payment must be included with all registrations to obtain advance prices. Payment may be made by check (payable to Oracle User Resource, Inc.), purchase order, or credit card authorization.

All participants are guaranteed a Proceedings Book upon check-in; however, we cannot guarantee additional conference materials and giveaways to those registering after March 3, 1997.

Conference Cancellation Refund Policy

Full conference fee will be charged for all reserved space. A full refund less $100 will be given to all cancellations requested in writing and received on or before March 3, 1997. After March 3, 1997, no refunds will be given.

		(✓)	FEE PER PERSON	
Advance registration fee (received on or before March 3, 1997)	User Group Member		$495	Please make a copy of this form and submit a completed form for each individual.
	Non-User Group Member		525	
Late registration fee (received after March 3, 1997)	User Group Member		545	
	Non-User Group Member		595	
	TOTAL			

Level Of Expertise
☐ Introductory ☐ Intermediate ☐ Advanced

User Group Affiliation _____

How many ECO Conferences have you attended in the past? _____

PAYMENT METHOD

Check/Money Order (U.S. Dollars only) ☐

Purchase Order # _____

Credit Card: Visa ☐ MasterCard ☐

CARD # _____

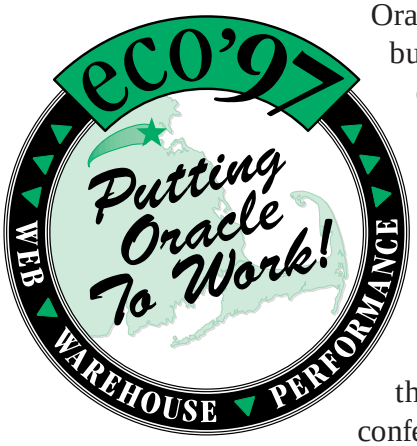
EXP. MM/YY: _____/BILLING ZIP CODE: _____

_____
CARD HOLDER'S SIGNATURE

_____
PRINT CARD HOLDER'S NAME AS IT APPEARS ON CARD.

Please make checks payable to Oracle User Resource, Inc.
1417 South College Road • Wilmington, NC 28403
Call: (910) 452-0006 • Fax: (910) 452-7834
E-mail: 71172.644@compuserve.com
Web address: www.oracle-users.com
PAYMENT MUST ACCOMPANY THIS FORM.

ECO'97's theme, "Putting Oracle to Work", emphasizes Oracle7 on the Web, Data Warehousing, Applications Development, and Oracle Performance Tuning. ECO continues to provide the best in-depth, technical sessions for applications developers and DBAs.



Oracle users responsible for building and maintaining enterprise-wide business critical applications and databases need ECO'97.

Every year ECO gets better because we take our participants' comments seriously and work to give them the technical conference that they want and need.

participants said:
"NO GENERAL SESSIONS!"

result:
THEY'RE GONE!
More technical sessions than ever

participants said:
"GIVE US MORE UNCONTESTED TIME WITH VENDORS!"

result:
ANOTHER INNOVATION:
The Corporate Campground with technical demonstrations and dedicated conference time

participants said:
"DESIGNER/2000 IS BECOMING MORE IMPORTANT TO OUR APPLICATIONS DEVELOPMENT."

result:
INCORPORATE CASE DAY INTO ECO'97
Gives attendees more value.

participants said:
"IT'S HARD TO KEEP UP WITH ORACLE'S NEW TECHNOLOGY."

result:
MINI-LESSON TRACK

INTRODUCTORY TRACK FOR NEW TECHNOLOGIES

ADDITIONAL HALF-DAY OF QUICK START TRAINING

bottom line:
ECO LISTENS!