Achieving PL/SQL Excellence

Oracle PL/SQL Best Practices

Steven Feuerstein
Me - www.StevenFeuerstein.com
PL/Solutions - www.plsolutions.com
RevealNet - www.revealnet.com

Software Used in Training

- **PL/Vision:** a library of packages installed on top of PL/SQL.
  - PL/Vision Lite - use it, copy, change it for free -- unless you build software to be sold commercially.
  - PL/Vision Professional: fully supported and enhanced version.
  - PL/Vision Professional Trial Version: full-use version good for 60 days from date of installation.
  - Information about all three available at the RevealNet site.
- Demonstration scripts executed in the training can be found on the RevealNet PL/SQL Pipeline:
  - www.revealnet.com/plsql-pipeline
  - Archives surfboard, Miscellaneous, PL/SQL Seminar Files
  - See filedesc.doc for a listing of many of the files.
- The PL/SQL IDE (Integrated Development Environment).
  - You no longer have to use SQL*Plus and a crude editor! Choose from among the many listed in plsql_ides.txt.

Training Objectives

- Learn how to build code that:
  - Is readable, both by the author and others.
  - Is more easily and quickly maintained.
  - Works more efficiently.
  - You are proud of.
- Improve your ability to review code: yours and others'.
  - And to do that, you have to know how to recognize what is *wrong* with a program.

7/5/99 Copyright 1999 Steven Feuerstein

"What is Wrong with this Code?"

- Code repetition!
  - More to fix, more to maintain.
- Exposed implementation!
  - It is showing me *how* it is getting the job done.
- Hard-coding!
  - It assumes that something will never change -- and that is *never* going to not happen.

7/5/99 Copyright 1999 Steven Feuerstein

Hard-Coding in PL/SQL

- Before we dive into the class, here is a list of different examples of hard-coding in PL/SQL:
  - Literal values
  - Every SQL statement you wrote, *especially* implicit cursors
  - COMMIT and ROLLBACK statements
  - Comments that "explain" code
  - Variables declared using base datatypes like VARCHAR2
  - Cursors containing bind variables
  - Fetching into a list of individual variables

Scary, isn't it?

- When you are done with this seminar, you will know how to get rid of all these kinds of hard-coding.

Training Outline

- Strategies for Implementing Best Practices
- Creating Readable and Maintainable Code
- Developing an Exception Handling Architecture
- Writing SQL in PL/SQL
- Package Construction
- Modularizing and Encapsulating Logic

Focus on PL/SQL technology common to Oracle7 & Oracle8

Developing an Exception Handling Architecture

7/5/99 Copyright 1999 Steven Feuerstein

Exception Handling in PL/SQL

Execute Application Code

Handle Exceptions

- The PL/SQL language provides a powerful, flexible "event-driven" architecture to handle errors which arise in your programs.
  - No matter how an error occurs, it will be trapped by the corresponding handler.
- Is this good? Yes and no.
  - You have many choices in building exception handlers.
  - There is no one right answer for all situations, all applications.
  - This usually leads to an inconsistent, incomplete solution.

7/5/99 Copyright 1999 Steven Feuerstein

You Need Strategy & Architecture

- To build a robust PL/SQL application, you need to decide on your strategy for exception handling, and then build a code-based architecture for implementing that strategy.
- In this section, we will:
  - Explore the features of PL/SQL error handling to make sure we have common base of knowledge.
  - Examine the common problems developers encounter with exception handling.
  - Construct a prototype for an infrastructure component that enforces a standard, best practice-based approach to trapping, handling and reporting errors.

The PL/Vision PLVexc package is a more complete implementation.

Flow of Exception Handling

```
PROCEDURE financial_review IS
BEGIN
   calc_profits (1996);

   calc_expenses (1996);

   DECLARE
      v_str VARCHAR2(1);
   BEGIN
      v_str := 'abc';
   EXCEPTION
      WHEN VALUE_ERROR THEN
         . . .
   END;
EXCEPTION
   WHEN OTHERS
   THEN
      ...
END;
```

```
PROCEDURE calc_profits (-) IS
BEGIN
   numeric_var := 'ABC';

EXCEPTION
   WHEN VALUE_ERROR THEN
      log_error;
      RAISE;
END;
```

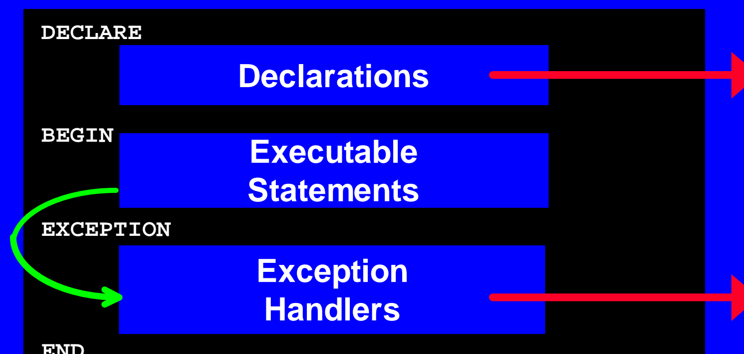
```
PROCEDURE calc_expenses (-) IS
BEGIN
   /* Do your thing! */
   SELECT x INTO y FROM …;

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      ...
END;
```

Scope and Propagation Reminders

- You can never go home.
  - Once an exception is raised in a block, that block's executable section closes. But *you* get to decide what constitutes a block.
- Once an exception is handled, there is no longer an exception (unless another exception is raised).
  - The next line in the enclosing block (or the first statement following the return point) will then execute.
- If an exception propagates out of the outermost block, then that exception goes *unhandled.*
  - In most environments, the host application stops.
  - In SQL*Plus and most other PL/SQL environments, an automatic ROLLBACK occurs.

What the Exception Section Covers

- The exception section only handles exceptions raised in the executable section of a block.
  - For a package, this means that the exception section only handles errors raised in the initialization section.



Continuing Past an Exception

- *Emulate* such behavior by enclosing code within its own block.

All or Nothing

```
PROCEDURE cleanup_details (id_in IN NUMBER) IS
BEGIN
   DELETE FROM details1 WHERE pky = id_in;
   DELETE FROM details2 WHERE pky = id_in;
END;
```

The "I Don't Care" Exception Handler

```
PROCEDURE cleanup_details (id_in IN NUMBER) IS
BEGIN
   BEGIN
      DELETE FROM details1 WHERE pky = id_in;
   EXCEPTION WHEN OTHERS THEN NULL;
   END;
   BEGIN
      DELETE FROM details2 WHERE pky = id_in;
   EXCEPTION WHEN OTHERS THEN NULL;
   END;
END;
```

defer.sql

Exceptions and DML

- DML statements are *not* rolled back by an exception unless it goes unhandled.
  – This gives you more control over your transaction, but it also can lead to complications.
  – What if you are logging errors to a database table? That log is then a part of your transaction.
- You may generally want to avoid "unqualified" ROLLBACKs and instead always specify a savepoint.

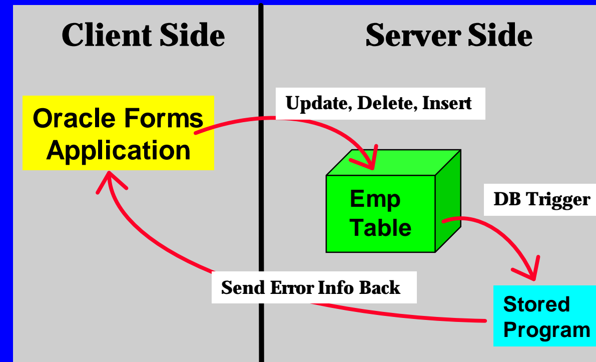
```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      ROLLBACK TO last_log_entry;
      INSERT INTO log VALUES (...);
      SAVEPOINT last_log_entry;
END;
```

But it can get complicated!

lostlog*.sql
rb.spp

Application-Specific Exceptions

- Raising and handling an exception specific to the application requires special treatment.
  - This is particularly true in a client-server environment with Oracle Developer.



Communicating an Application Error

- Use the RAISE_APPLICATION_ERROR built-in procedure to communicate an error number and message across the client-server divide.
  - Oracle sets aside the error codes between -20000 and -20999 for your application to use. RAISE_APPLICATION_ERROR can only be used those error numbers.

```
RAISE_APPLICATION_ERROR
   (num binary_integer,
    msg varchar2,
    keeperrorstack boolean default FALSE);
```

- The following code from a database triggers shows a typical usage of RAISE_APPLICATION_ERROR.

```
IF birthdate > ADD_MONTHS (SYSDATE, -216)
THEN
   RAISE_APPLICATION_ERROR
      (-20070, 'Employee must be 18.');
END IF;
```

Handling App. Specific Exceptions

- Handle in OTHERS with check against SQLCODE...

```
BEGIN
   INSERT INTO emp (empno, deptno, birthdate)
      VALUES (100, 200, SYSDATE);
EXCEPTION
   WHEN OTHERS THEN
      IF SQLCODE = -20070 ...
END;
```

Server-side Database

- Or handle with named exception, declared on client side ...

```
DECLARE
   emp_too_young EXCEPTION;
   PRAGMA EXCEPTION_INIT (emp_too_young, -20070);
BEGIN
   INSERT INTO emp (empno, deptno, birthdate)
      VALUES (100, 200, SYSDATE);
EXCEPTION
   WHEN emp_too_young THEN ...
END;
```

Server-side Database

The Ideal But Unavailable Solution

- Declare the exception in one place (server) and reference it (the error number or name) throughout your application.

```
CREATE OR REPLACE PACKAGE emp_rules IS
   emp_too_young EXCEPTION;
END;
```

Server side pkg defines exception.

```
IF birthdate > ADD_MONTHS (SYSDATE, -216) THEN
   RAISE emp_rules.emp_too_young;
END IF;
```

Database trigger raises exception.

```
BEGIN
   INSERT INTO emp VALUES (100, 200, SYSDATE);
EXCEPTION
   WHEN emp_rules.emp_too_young THEN ...
END;
```

Client side block handles exception.

- But this won't work with Oracle Developer! If it's got a dot and is defined on the server, it can only be a function or procedure, not an exception or constant or variable...

Blocks within Blocks I

- What information is displayed on your screen when you execute this block?

```
DECLARE
   aname VARCHAR2(5);
BEGIN
   BEGIN
      aname := 'Justice';
      DBMS_OUTPUT.PUT_LINE (aname);
   EXCEPTION
      WHEN VALUE_ERROR
      THEN
         DBMS_OUTPUT.PUT_LINE ('Inner block');
   END;
   DBMS_OUTPUT.PUT_LINE ('What error?');
EXCEPTION
   WHEN VALUE_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE ('Outer block');
END;
```

excquiz1.sql

Blocks within Blocks II

- What information is displayed on your screen when you execute this block?

```
DECLARE
   aname VARCHAR2(5);
BEGIN
   DECLARE
      aname VARCHAR2(5) := 'Justice';
   BEGIN
      DBMS_OUTPUT.PUT_LINE (aname);

   EXCEPTION
      WHEN VALUE_ERROR THEN
         DBMS_OUTPUT.PUT_LINE ('Inner block');
   END;
   DBMS_OUTPUT.PUT_LINE ('What error?');
EXCEPTION
   WHEN VALUE_ERROR THEN
      DBMS_OUTPUT.PUT_LINE ('Outer block');
END;
```

excquiz2.sql
excquiz2a.sql

Blocks within Blocks III

- What do you see when you execute this block?

```
DECLARE
   aname VARCHAR2(5);
BEGIN
   <<inner>>
   BEGIN
      aname := 'Justice';
   EXCEPTION
      WHEN VALUE_ERROR THEN
         RAISE NO_DATA_FOUND;

      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE ('Inner block');
   END inner;

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Outer block');
END;
```

excquiz3.sql

Blocks within Blocks IV

- What do you see when you execute this block?
  - Assume that there are no rows in emp where deptno equals -15.

```
DECLARE
   v_totsal NUMBER;
   v_ename emp.ename%TYPE;
BEGIN
   SELECT SUM (sal) INTO v_totsal FROM emp WHERE deptno = -15;

   p.l ('Total salary', v_totsal);

   SELECT ename INTO v_ename
     FROM emp
    WHERE sal =
       (SELECT MAX (sal) FROM emp WHERE deptno = -15);

   p.l ('The winner is', v_ename);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      p.l ('Outer block');
END;
```

excquiz4.sql

Taking Exception to My Exceptions

- What do you see when you execute this block?

```
DECLARE
   d VARCHAR2(1);
   no_data_found EXCEPTION;
BEGIN
   SELECT dummy INTO d
     FROM dual
    WHERE 1=2;

   IF d IS NULL
   THEN
      RAISE no_data_found;
   END IF;

EXCEPTION
   WHEN no_data_found
   THEN
      DBMS_OUTPUT.PUT_LINE ('No dummy!');
END;
```

excquiz5.sql

What, No GOTO?

- It gets the job done…but does the end justify the means? What should this procedure look like if I want to create a "wrapper" around calc_totals?

```
FUNCTION totalsales (year IN INTEGER) RETURN NUMBER
IS
   return_nothing EXCEPTION;
   return_the_value EXCEPTION;
   retval NUMBER;
BEGIN
   retval := calc_totals (year);

   IF retval = 0 THEN
      RAISE return_nothing;
   ELSE
      RAISE return_the_value;
   END IF;

EXCEPTION
   WHEN return_the_value THEN RETURN retval;
   WHEN return_nothing THEN RETURN 0;
END;
```

isvalinlis.sql

An Exceptional Package

```
PACKAGE valerr
IS
  FUNCTION
    get RETURN VARCHAR2;
END valerr;
```

```
PACKAGE BODY valerr IS
   v VARCHAR2(1) := 'abc';
   FUNCTION get RETURN VARCHAR2 IS
   BEGIN
      RETURN v;
   END;
BEGIN
   p.l ('Before I show you v...');
EXCEPTION
  WHEN OTHERS THEN
    p.l ('Trapped the error!');
END valerr;
```

- So I create the valerr package, compile it, and then execute the following command. What is displayed on the screen? And what is displayed when I immediately run it a second time?

```
SQL> EXECUTE p.l (valerr.get);
```

valerr.spp
valerr2.spp

7/5/99 Copyright 1999 Steven Feuerstein

Desperately Seeking Clarity

- Hopefully everyone now feels *more* confident in their understanding of how exception handling in PL/SQL works.
- Let's move on to an examination of the challenges you face as you build an application and seek to build *into* it consistent error handling.
- After that, we take a look at how you might build a generic, reusable infrastructure component to handle the complexities of exception handling.

7/5/99 Copyright 1999 Steven Feuerstein

Challenges of Exception Handling

- Exception handling is one of the more complicated aspects of your application.
  - You have to understand how it all works.
  - You want exceptions handled consistently throughout your application.
- Some specific issues you will face:
  - What kind of action do you take in a given situation?
  - Do you maintain a log of errors? If so, what form does the log take?
  - How do you avoid overlapping usage of the -20,NNN error numbers?
  - How do you consolidate and access message text?

All-Too-Common Handler Code

```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      v_msg := 'No company for id ' || TO_CHAR (v_id);
      v_err := SQLCODE;
      v_prog := 'fixdebt';
      INSERT INTO errlog VALUES
         (v_err, v_msg, v_prog, SYSDATE, USER);
   WHEN OTHERS THEN
      v_err := SQLCODE;
      v_msg := SQLERRM;
      v_prog := 'fixdebt';
      INSERT INTO errlog VALUES
         (v_err, v_msg, v_prog, SYSDATE, USER);
      RAISE;
```

I am showing everyone how the log is kept.

Lots of redundant code.

- If every developer writes exception handler code on their own, you end up with an unmanageable situation.
  - Different logging mechanisms, no standards for error message text, inconsistent handling of the same errors, etc.

Some Dos and Don'ts

- Make decisions about exception handling *before* starting your application development.
- Developers should never:
  - Use RAISE_APPLICATION_ERROR.
  - Use PRAGMA EXCEPTION_INIT.
  - Code explicit -20NNN error numbers.
  - Hard-code error messages.
  - Expose the implementation of exception handling logic.
- Developers should always:
  - Handle errors by calling the pre-defined handler programs.
  - Raise errors (particularly -20NNN errors) by calling a pre-defined program, instead of the native program.
  - Retrieve error messages through the function interface.
  - Work with generated, standards-based code instead of *writing* the code again and again.

7/5/99 Copyright 1999 Steven Feuerstein

Checking Standards Compliance

- Whenever possible, try to move beyond document-based standards.
  - Instead, build code to both help people deploy standards and create tools to help verify that they have complied with standards.

```
CREATE OR REPLACE PROCEDURE progwith (str IN VARCHAR2)
IS
   CURSOR objwith_cur (str IN VARCHAR2)
   IS
      SELECT DISTINCT name
        FROM USER_SOURCE
       WHERE UPPER (text) LIKE '%' || UPPER (str) || '%';

BEGIN
   FOR prog_rec IN objwith_cur (str)
   LOOP
      p.l (prog_rec.name);
   END LOOP;
END;
```

valstd.spp

7/5/99 Copyright 1999 Steven Feuerstein

Pre-Defined -20,NNN Errors

```
PACKAGE errnums
IS
   en_general_error CONSTANT NUMBER := -20000;
   exc_general_error EXCEPTION;
   PRAGMA EXCEPTION_INIT
     (exc_general_error, -20000);

   en_must_be_18 CONSTANT NUMBER := -20001;
   exc_must_be_18 EXCEPTION;
   PRAGMA EXCEPTION_INIT
     (exc_must_be_18, -20001);

   en_sal_too_low CONSTANT NUMBER := -20002;
   exc_sal_too_low EXCEPTION;
   PRAGMA EXCEPTION_INIT
     (exc_sal_too_low , -20002);

   max_error_used CONSTANT NUMBER := -20002;

END errnums;
```

Assign Error Number

Declare Named Exception

Associate Number w/Name

But don't write this code manually!

msginfo.spp

Reusable Exception Handler Package

```
PACKAGE errpkg
IS
   PROCEDURE raise (err_in IN INTEGER);

   PROCEDURE recNstop (err_in IN INTEGER := SQLCODE,
      msg_in IN VARCHAR2 := NULL);

   PROCEDURE recNgo (err_in IN INTEGER := SQLCODE,
      msg_in IN VARCHAR2 := NULL);

   FUNCTION errtext (err_in IN INTEGER := SQLCODE)
     RETURN VARCHAR2;

END errpkg;
```

Generic Raise

Record and Stop

Record and Continue

Message Text Consolidator

errpkg.spp

Implementing a Generic RAISE

- Hides as much as possible the decision of whether to do a normal RAISE or call RAISE_APPLICATION_ERROR.
  - Also forces developers to rely on predefined message text.

```
PROCEDURE raise (err_in IN INTEGER) IS
BEGIN
   IF err_in BETWEEN -20999 AND -20000
   THEN
      RAISE_APPLICATION_ERROR (err_in, errtext (err_in));
   ELSIF err_in IN (100, -1403)
   THEN
      RAISE NO_DATA_FOUND;
   ELSE
      PLVdyn.plsql (
        'DECLARE myexc EXCEPTION; ' ||
        '    PRAGMA EXCEPTION_INIT (myexc, ' ||
              TO_CHAR (err_in) || ');' ||
        'BEGIN  RAISE myexc; END;');
   END IF;
END;
```

Enforce use of standard message

Re-raise *almost* any exception using Dynamic PL/SQL!

7/5/99 Copyright 1999 Steven Feuerstein

Raising Application Specific Errors

- With the generic raise procedure and the pre-defined error numbers, you can write high-level, readable, maintainable code.
  - The individual developers make fewer decisions, write less code, and rely on pre-built standard elements.
- Let's revisit that trigger logic using the infrastructure elements...

```
PROCEDURE validate_emp (birthdate_in IN DATE) IS
BEGIN
   IF ADD_MONTHS (SYSDATE, 18 * 12 * -1) < birthdate_in
   THEN
      errpkg.raise (errnums.en_must_be_18);
   END IF;
END;
```

No more hard-coded strings or numbers.

7/5/99 Copyright 1999 Steven Feuerstein

Deploying Standard Handlers

- The rule: developers should *only* call a pre-defined handler inside an exception section.
  - Make it impossible for developers to *not* build in a consistent, high-quality way.
  - They don't have to make decisions about the form of the log and how the process should be stopped.

```
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      errpkg.recNgo (
         SQLCODE,
         ' No company for id ' || TO_CHAR (v_id));

   WHEN OTHERS
   THEN
      errpkg.recNstop;
END;
```

The developer simply *describes* the desired action.

Implementing a Generic Handler

- Hides all details of writing to the log, executing the handle action requested, etc.

```
PACKAGE BODY errpkg
IS
   PROCEDURE recNgo (err_in IN INTEGER := SQLCODE,
      msg_in IN VARCHAR2 := NULL)
   IS
   BEGIN
      log.put (err_in, NVL (msg_in, errtext (err_in)));
   END;

   PROCEDURE recNstop (err_in IN INTEGER := SQLCODE,
      msg_in IN VARCHAR2 := NULL)
   IS
   BEGIN
      recNgo (err_in, msg_in);
      errpkg.raise (err_in);
   END;

END errpkg;
```

Pre-existing package elements are re-used.

Retrieving Consolidated Message Text

```
FUNCTION errtext (err_in IN INTEGER := SQLCODE) RETURN VARCHAR2
IS
   CURSOR txt_cur IS
      SELECT text FROM message_text
       WHERE texttype = 'EXCEPTION' AND code = err_in;
   txt_rec txt_cur%ROWTYPE;
BEGIN
   OPEN txt_cur;
   FETCH txt_cur INTO txt_rec;
   IF txt_cur%NOTFOUND THEN
      txt_rec.text := SQLERRM (err_in);
   END IF;
   RETURN txt_rec.text;
END;
```

You can selectively override default, cryptic Oracle messages.

- Or, as shown in the errpkg.spp file, you can call the underlying msginfo packaged function to retrieve the text from that standardized component.

Added Value of a Handler Package

- Once you have all of your developers using the handler package, you can add value in a number of directions:
  – Store templates and perform runtime substitutions.
  – Offer the ability to "bail out" of a program, no matter how many layers of nesting lie between it and the host application.

exc.tst
exc2.tst
exc3.tst

An Exception Handling Architecture

- Make Sure You Understand How it All Works
  - Exception handling is tricky stuff.
- Set Standards Before You Start Coding
  - It's not the kind of thing you can easily add in later.
- Use Standard Infrastructure Components
  - Everyone and all programs need to handle errors the same way.