Achieving PL/SQL Excellence
Oracle PLSQL Best Practices and Tuning

Steven Feuerstein
steven@stevenfeuerstein.com
steven.feuerstein@quest.com
www.quest.com

QUEST SOFTWARE

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
  - Advanced PL/SQL Knowledge Base: contains PL/Vision Professional, the fully supported and enhanced version
- Demonstration scripts executed in the training can be found on the RevealNet PL/SQL Pipeline:
  - www.revealnet.com/Pipelines/PLSQL/index.htm
  - Archives, Miscellaneous, PL/SQL Seminar Files
  - See filedesc.doc for a description of many of the files
- A 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

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'
  - To do that, you have to know how to recognize what is *wrong* with a program

Training Outline

- Setting the stage
- Writing SQL in PL/SQL
- Package Construction
- Modularizing and Encapsulating Logic
- Deploy an Exception Handling Architecture
- Unit Test within a Framework
- Optimize Algorithms
- Use Data Structures Efficiently
- Manage Code in the Database and SGA
- Create Readable and Maintainable Code

PL/SQL Tuning & Best Practices
Setting the Stage

- What's wrong with this code?
- Setting expectations re: tuning
- Implementation strategies for best practices
- Analyzing performance
- Understanding the PL/SQL Architecture

"What is Wrong with this Code?"

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

Hard-Coding in PL/SQL

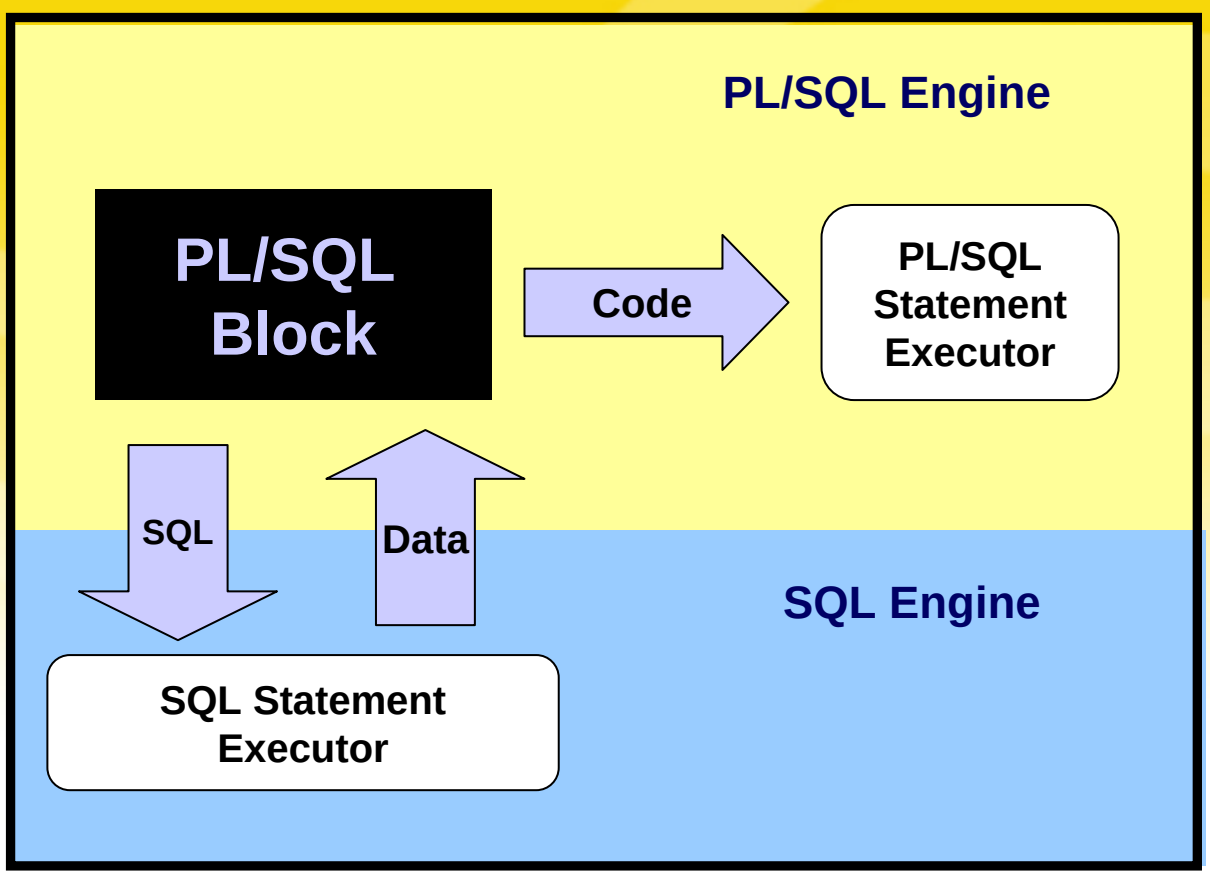
- Literal values IF num_requests > 100 THEN
- Date format in a call to TO_DATE 'MM/DD/YY'
- Language specificities '09-SEP-2001'
- Constrained declarations NUMBER(10,4)
- Variables declared using base datatypes my_name VARCHAR2(20)
- Every SQL statement you write, *especially* implicit cursors
- COMMIT and ROLLBACK statements
- Fetching into a list of individual variables
- Embedded (unencapsulated) business rules

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.

Scope of Tuning Training

- Focus is on PL/SQL -- not SQL
- In other words: tune your SQL first!
- Limited treatment of SQL tuning, SGA sizing and analysis, SQL-specific new features in Oracle, etc.



QUEST SOFTWARE

Expectations for a Training on Tuning

Improve the performance of your application 1000 fold for only $19.95 a month!*

***Plus shipping and handling and technical support. All performance degradation the responsibility of the user.**

QUEST SOFTWARE

Resources for PL/SQL Tuning

- Interested in "Oracle tuning"? The world is your oyster:
  - Oracle documentation
  - Numerous tuning books and Web sites
  - Many, many tools
- But PL/SQL tuning? Slim pickings...
  - PL/SQL books and general Oracle tuning books offer some coverage, but it is minimal and piecemeal
  - Code Complete by Steve McConnell (Microsoft Press)
    - Many tuning tips are not language-specific. This book offers an excellent treatment of tuning philosophies and issues you have to address in any programming language
- PL/SQL tuning is *tough*, compared to SQL tuning
  - You have to analyze and tune algorithms

Tuning Myths

- “A fast program is as important as a correct one”
  - Squeezing out microsecond performance improvements is much less important than meeting user requirements
- "Fewer lines of code improves speed"
  - There is no predictable relationship between lines of code and resulting performance. Think recursion...
  - But increasing code density definitely reduces readability
- “Certain operations are probably faster than others”
  - There is no room for “probably” in tuning. Either an operation is faster or it is not -- and that fact can change as you change versions and environments
- "Optimize as you write your code"
  - Don't knowingly write inefficient code, but don't agonize over code that may never contribute to a performance bottleneck. These are often only apparent when you run a completed application

PL/SQL Tuning & Best Practices

Strategies for Implementing Best Practices

QUEST SOFTWARE

Words Are Not Enough!

You have a long list of best practices and standards that you plan to deploy in your organization.

How can you get developers to "follow the rules"? Send out a memo? Hold a meeting?

- Need to develop a "culture of quality", so that all developers consciously seek to write higher quality code.
- Then move beyond words to software components and utilities that:
  - Implement the standards for developers, and simply present them as infrastructure components
  - Generate standards-based code
  - Analyze code for compliance with standards

Generator and Analyzer Utilities

- You have a number of options to generate code:
  - Build your own. It's not too hard to do something "quick and dirty" -- and very focused on your needs
  - Use the PL/Vision PLVgen package to create standard "code starters"
  - Many IDEs have templates and mini code libraries
  - Full-blown generation tools like Oracle Designer and PL/Generator
- You can also use SQL and the data dictionary to analyze your code
  - It's one of the big advantages of stored code

```
SELECT DISTINCT name
  FROM USER_SOURCE
 WHERE INSTR (UPPER (text), ' CHAR') > 0;
```

showchar.sql
showei.sql
showsrc.sql
valstd.pkg

QUEST SOFTWARE

Strategies for Implementation
Summary

- Involve the entire dev team in creating and adopting best practices.
  - Anything imposed risks backlash
- Provide tools that implement standards "automatically"
  - Developers won't even realize they're conforming
- Analyze stored code for compliance with standards
  - Use the power of SQL to manage your code, as well as your users' data

PL/SQL Tuning & Best Practices

Analyzing Performance

- Utilize Existing Tools
- The PL/SQL Code Profiler (Oracle8i)
- The DBMS_APPLICATION_INFO package
- Homegrown Timer Utility
- Compare Implementations
- Calculate Overhead of Various Operations

Measure Before You Tune

- Oracle PL/SQL is a different sort of animal from standard programming languages
  - It not only is designed to manipulate Oracle RDBMS information, but the code is stored in and executed from that database!
  - Accessing that code competes with other DB operations.
- Tuning PL/SQL code involves three main areas of tuning:
  - SQL statements
  - PL/SQL code (algorithms)
  - Memory utilization and management of code

- Critical to tune for concurrent users, not just single user.

But before you can tune, you must know what is running slowly. You must *measure*.

Utilize Existing Tools

- Oracle and other software vendors now provide a number of performance analysis tools
  - Unfortunately, almost all focus almost exclusively on SQL
- TKPROF
  - Lots of output, lots of work to analyze that output
- Later-generation tools embed and offer expertise
  - Sure, you should know the basics of SQL tuning, but it is increasingly possible to rely on SW-based knowledge to do the hard work for you
  - Oracle Performance Pack
  - Quest SQLab
  - Computer Associates SQL Station Plan Analyzer
  - Others, I am sure...

The PL/SQL Code Profiler

- In Oracle8i, the DBMS_PROFILER package offers an API to facilitate performance and code coverage analysis
- Easy to gather statistics:
  - Install the package and supporting elements
  - Start the profiler
  - Run your code
  - Stop the profiler
- Bigger challenge: Analyzing the results...

```
BEGIN
   DBMS_OUTPUT.PUT_LINE (
      DBMS_PROFILER.START_PROFILER (
         'showemps ' ||
         TO_CHAR (SYSDATE, 'YYYYMMDD HH24:MI:SS')
         )
      );
   showemps;
   DBMS_OUTPUT.PUT_LINE (
      DBMS_PROFILER.STOP_PROFILER);
END;
```

Installing the Code Profiler

- You will probably have to install the code yourself (in the SYS schema) Check the package specification file for documentation and guidelines
  - Specification: dbmspbp.sql Body: prvtpbp.plb
  - Files are located in Rdbms\Admin unless otherwise noted
- You must install the profile tables by running the proftab.sql script. Can define them in a shared schema or in separate schemas
- Creates three tables:
  - PLSQL_PROFILER_RUNS: parent table of runs
  - PLSQL_PROFILER_UNITS: program units executed in run
  - PLSQL_PROFILER_DATA: profiling data for each line in a program unit
- These tables, particularly *_DATA, can end up with *lots* of rows in them

Interpreting Code Profiler Results

- To make it easier to analyze the data produced by the profiler, Oracle offers the following files in the Ora81\plsql\demo directory:
  - profrep.sql: Creates a number of views and a package named prof_report_utilities to help extract data from profiling tables
  - profsum.sql: series of canned queries and programs using prof_report_utilities Don't run them all; pick the ones that look most useful

```
EXECUTE prof_report_utilities.rollup_all_runs;

/* Total time */
SELECT TO_CHAR (grand_total / 1000000000, '999999.99') AS grand_total
  FROM plsql_profiler_grand_total;

/* Total time spent on each run */
SELECT runid, SUBSTR (run_comment, 1, 30) AS run_comment,
       run_total_time / 1000000000 AS seconds
  FROM plsql_profiler_runs
 WHERE run_total_time > 0
 ORDER BY runid asc;
```

slowest.sql
slowest.txt

Homegrown Timer Utility

- PL/SQL developers often need a very granular timing mechanism: What is the elapsed time for this one function?
- Oracle offers several mechanisms to get this information
  - DBMS_UTILITY.GET_TIME function
  - V$TIMER table and other V$ data sources
  - SQL*Plus SET TIMING ON
- In most cases, you will be best off building an encapsulation around the lower-level functionality

Package Implementation

```
BEGIN
   PLVtmr.capture;
   showemps;
   PLVtmr.show_elapsed
      ('Showemps');
END;
```

Object Implementation

```
DECLARE
   se_tmr tmr_t :=
      tmr_t.make ('Showemps');
BEGIN
   se_tmr.go;
   showemps;
   se_tmr.stop;
END;
```

plvtmr.pkg
tmr81.ot
vsesstat.pkg

Compare Implementations

- You write a program and it just doesn't seem fast enough
- Time to try a different approach -- and then you need to compare them
  - The timer utility offers an easy way to do this

```
DECLARE
   v_user VARCHAR2(30);
   once_tmr tmr_t :=
      tmr_t.make ('Packaged',
&1);
   every_tmr tmr_t :=
      tmr_t.make ('USER', &1);
BEGIN
   once_tmr.go;
   FOR indx IN 1 .. &1
   LOOP
      v_user := thisuser.name;
   END LOOP;
   once_tmr.stop;

   every_tmr.go;
   FOR indx IN 1 .. &1
   LOOP
      v_user := USER;
   END LOOP;
   every_tmr.stop;
END;
```

thisuser.*

Calculate Overhead

- We often wonder how expensive is a particular operation, or when Oracle actually performs a requested task
- Common questions:
  - Does Oracle identify the result of a query with the OPEN or the first FETCH?
  - What is the overhead of a procedure or function call?

```
DECLARE
   otmr tmr_t :=
      tmr_t.make ('OPEN?');
   ftmr tmr_t :=
      tmr_t.make ('FETCH?');

   CURSOR lots_stuff IS
      SELECT *
        from plsql_profiler_data
        Order BY total_time DESC;

   lots_rec lots_stuff%ROWTYPE;
BEGIN
   otmr.go;
   OPEN lots_stuff;
   otmr.stop;

   ftmr.go;
   FETCH lots_stuff INTO lots_rec;
   ftmr.stop;
END;
```

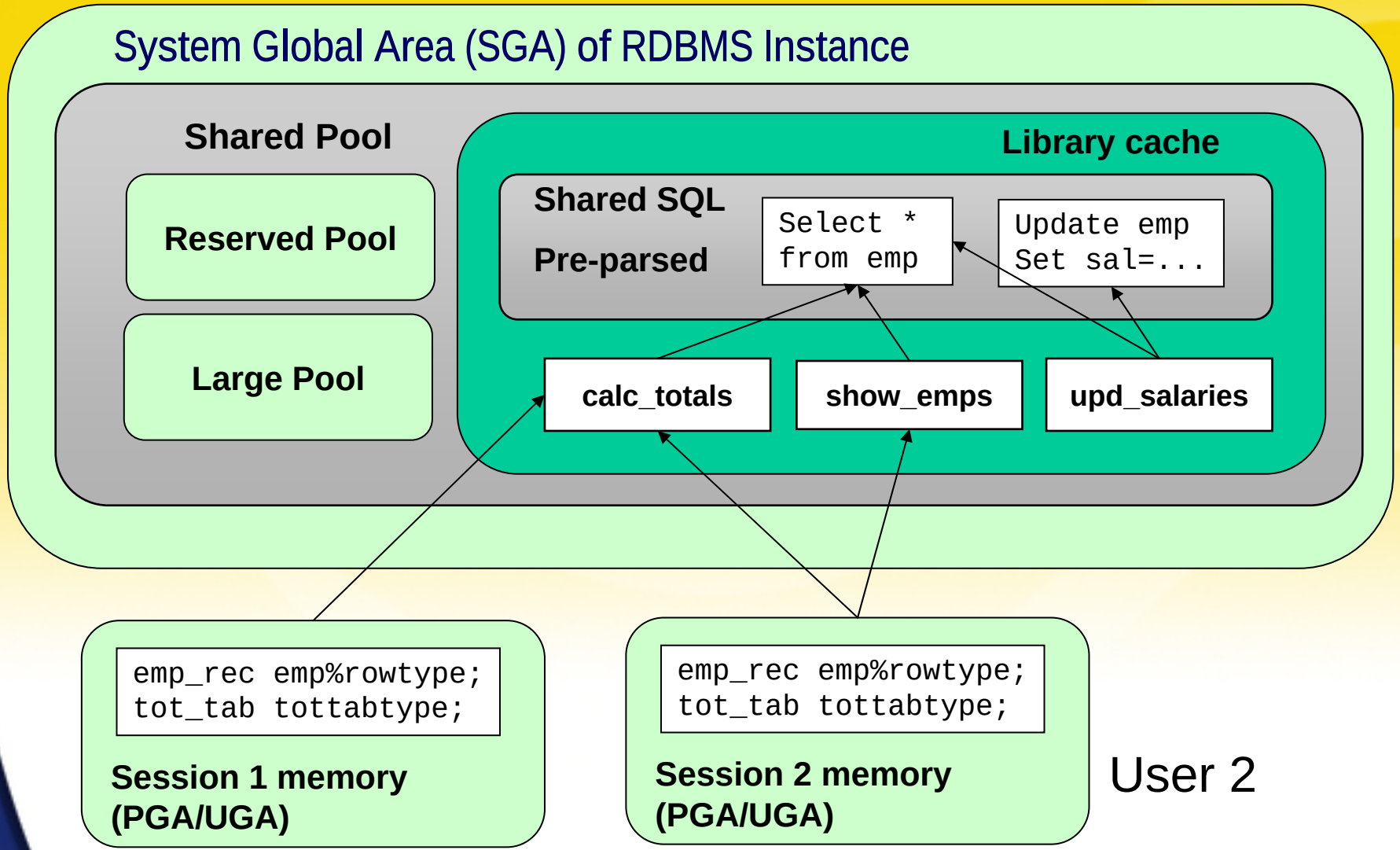
```
SQL> @open_fetch
Elapsed for "OPEN" = 0 seconds.
Elapsed for "FETCH?" = 1490.1
seconds.
```

open_fetch.sql
curperf*.sql
ovrhead.sql

PL/SQL Tuning & Best Practices

Understanding PL/SQL Architecture

PL/SQL in Shared Memory



QUEST SOFTWARE

Code in Shared Memory

- PL/SQL is an interpretative language. The source code is "partially compiled" into an intermediate form ("p-code").
  - The p-code is loaded into the shared pool when any element of that code (package or stand-alone program) is referenced.
- The partially-compiled code is shared among all users who have EXECUTE authority on the program/package.
  - Prior to Oracle7.3, contiguous memory was required for program units. That is now relaxed, but still preferable.
- Each user (Oracle session) has its own copy of any data structures defined within the program/package.
  - Separate sets of in-memory data (not shared among different users) are stored in the PGA.

PL/SQL Tuning and Best Practices

Writing SQL in PL/SQL

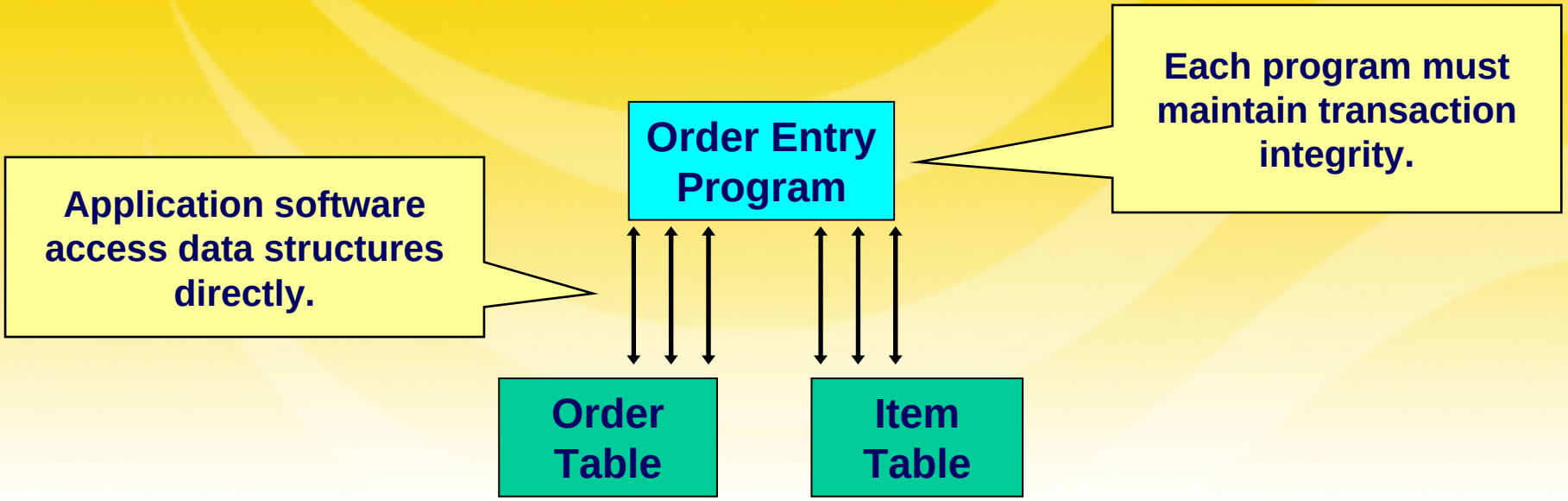
- What's the Big Deal?
- Some Rules to Follow
- Synchronize Code with Data Structures
- Avoid Repetition of SQL
- Optimize the way we write SQL in PL/SQL

Why We Write PL/SQL Code

- The predominant reason you write PL/SQL programs is to interact with the database, which:
  - Is the repository of information that shapes your business
  - Is always changing
- The layer of PL/SQL code should *support* the data model
  - It should *not* disrupt your ability to maintain and work with that model
  - But common coding practices tend to do just that: make it extremely difficult to modify and enhance your code as the data structures change
- The difficulties surface in two different areas:
  - Transaction integrity
  - Poor coding practices

Transaction Integrity the Hard Way

Typical Data Access Method



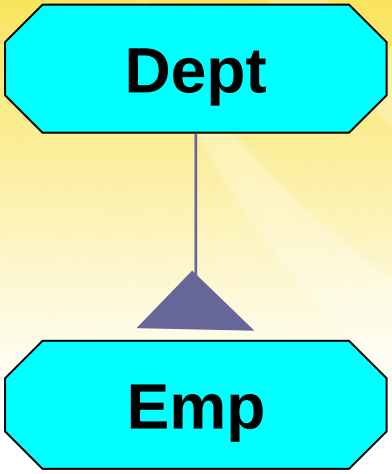
- When a transaction consists of three updates, two inserts, a delete and six queries, how do you guarantee that each developer is going to get it right?

QUEST SOFTWARE

The Dangers of Poor Coding Practices

- If you are not very careful, it is very easy to write your code in ways that cause your code to *break* whenever a change occurs in the underlying structures

Data Structures



Code

```
PROCEDURE calc_totals
IS
   v_dname VARCHAR2(20);
   v_ename CHAR(30);
BEGIN
   SELECT dname, ename
     INTO v_dname, v_ename
     FROM emp, dept
    WHERE ...;
   ...
END;
```

This program is a ticking time bomb in my application...

QUEST SOFTWARE

The View from 30,000 Feet

High-Level Best Practices

- Never repeat an SQL statement in application code
- Encapsulate all SQL statements behind a procedural interface, usually a package
- Write your code assuming that the underlying data structures will change
- Take advantage of PL/SQL-specific enhancements for SQL

Never Repeat SQL

- Take the "acid test" of SQL in PL/SQL: Can you say "sure" to the following question?

> Do you know all the places in your code where an INSERT (or DELETE or UPDATE) occurs for your table(s)?

  - If the answer is "not really", then you have essentially lost control of your application code base
- It is crucial that you avoid repetition of the same logical SQL statement...
  - With repetition, comes variation, and with it excessive parsing
  - Potentially significant impact on performance and maintainability

And sometimes you have to worry about more than *logical* variations!

Give W/One Hand,Take W/the Other

- Oracle sometimes improves things in ways that make it very difficult for us to take advantage of them
- When are two SQL statements the same and yet different?

twoblocks.sql

Column A		Column B	
`SELECT COUNT(*) FROM after_deforestation;`	=	`select count(*) from after_deforestation;`	?
`BEGIN UPDATE favorites SET flavor = 'CHOCOLATE' WHERE name = 'STEVEN'; END;`	=	`BEGIN update favorites set flavor = 'CHOCOLATE' where name = 'STEVEN'; END;`	?
`BEGIN UPDATE ceo_compensation SET stock_options = 1000000, salary = salary * 2.0 WHERE layoffs > 10000; END;`	=	`BEGIN update ceo_compensation set stock_options = 1000000, salary = salary * 2 where layoffs > 10000; END;`	?

Crossing the Physical-Logical Divide

- When you write SQL, you must be aware of the *physical* representation of your code
  - Pre-parsed cursors are only used for byte-wise equal statements (analyzed using a hash of the SQL string)
  - White space (blanks, tabs, line breaks) make a difference – except for SQL inside PL/SQL blocks
  - PL/SQL reformats SQL to avoid nuisance redundancy
- Not much can be done, however, about these kinds of logical duplications:

```
BEGIN
   UPDATE ceo_compensation
      SET stock_options = 1000000,
       salary = salary * 2
    WHERE layoffs > 10000;
```

```
BEGIN
   update ceo_compensation
      set salary = salary * 2,
          stock_options = 1000000
    where layoffs > 10000;
```

How to Avoid SQL Repetition

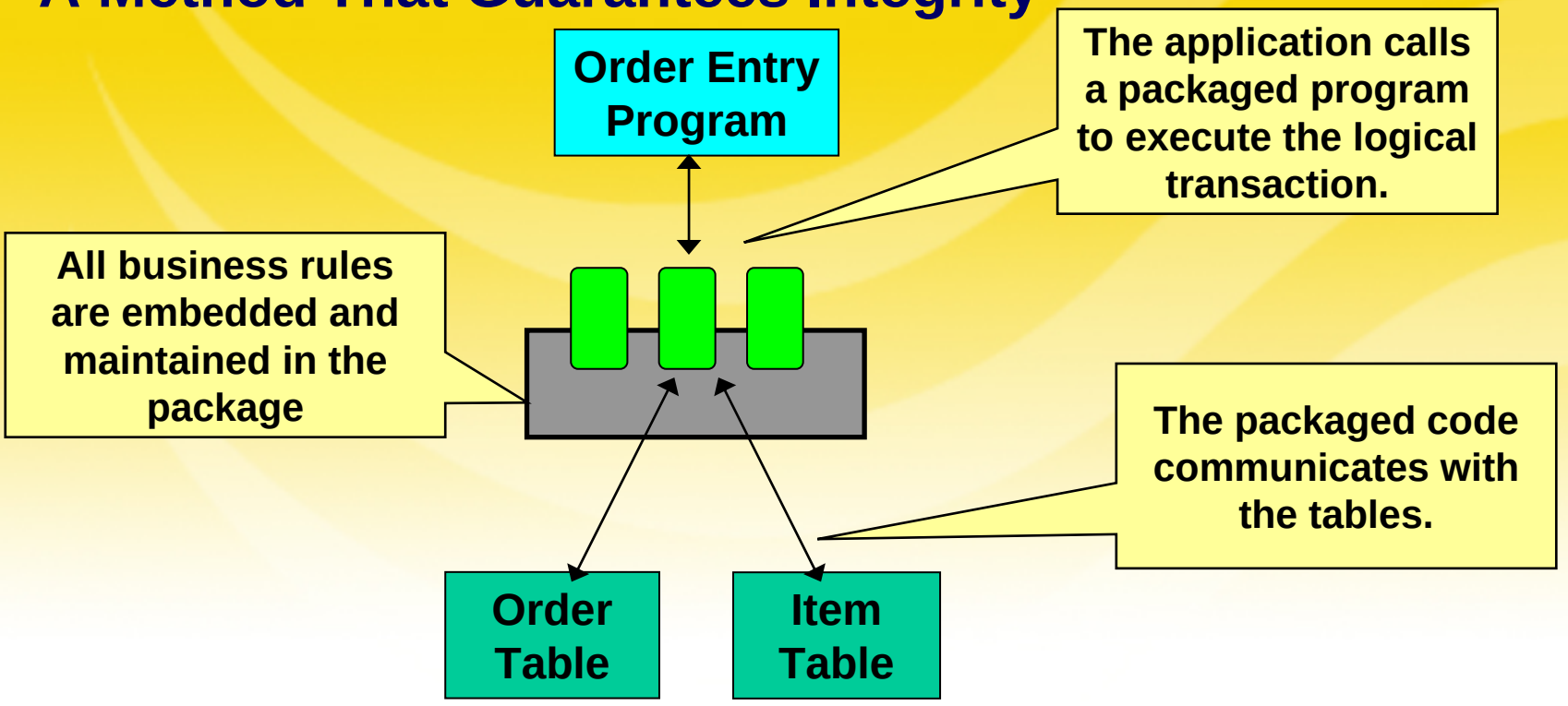
- You should, as a rule, not even *write* SQL in your PL/SQL programs
  - You can't repeat it if you don't write it
- Instead, rely on pre-built, pre-tested, write-once, use-often PL/SQL programs.
  - "Hide" both individual SQL statements and entire transactions.

Guaranteed transaction integrity!

QUEST SOFTWARE

Transaction Integrity with PL/SQL

A Method That Guarantees Integrity



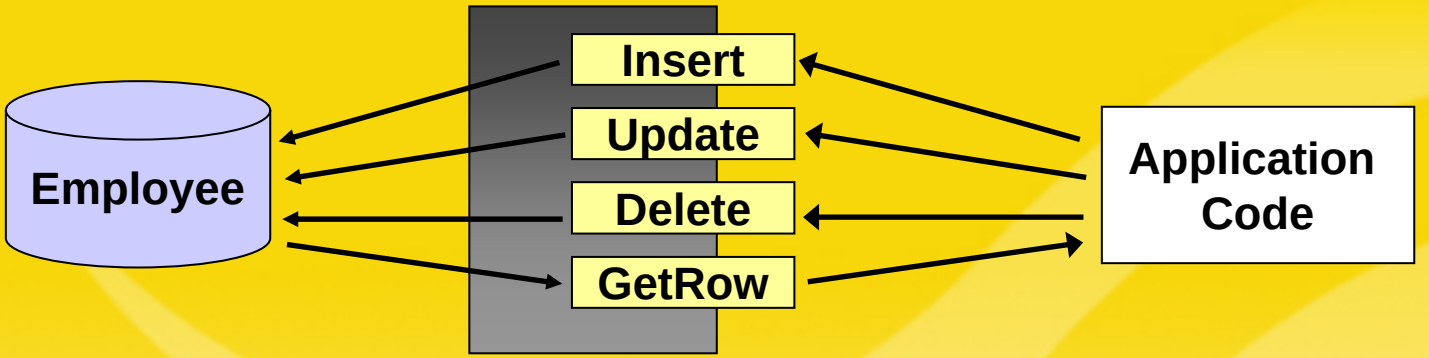
- This is why Oracle originally developed the PL/SQL language!

QUEST SOFTWARE

Hide all SQL Behind Interface

- A team lead can't watch over everybody's shoulders to "police" the construction of every SQL statement
  - Instead, a group needs to set policies and provide code so that everyone can follow the rules – and write better code
- Here are some recommendations:
  - Build and use table encapsulation packages
  - Hide all single row queries behind function interfaces
  - In particular, don't expose the dual table
  - Move multi-row cursors into packages

The Beauty of Table Encapsulation



- Move all SQL inside packages: one per table or "business object"
  - All DML statements written by an expert, behind a procedural interface, with standardized exception handling
  - Commonly-needed cursors and functions to return variety of data (by primary key, foreign key, etc.)
  - If the encapsulation package doesn't have what you need, add the new element, so that everyone can take advantage of it
  - Separate packages for query-only and change-related functionality (for added security)

te_employee.*
givebonus*.sp

Allow No Exceptions!

- Instead of this:

```
INSERT INTO employee
   (employee_id, department_id, salary, hire_date)
VALUES
   (1005, 10, 10000, SYSDATE);
```

- Do this:

```
te_employee.insert (
   employee_id_in => 1005, department_id_in => 10,
   salary_in => 10000, hire_date_in => SYSDATE);
```

- Check dependency information to identify programs that rely directly on tables

TRUE STORY!
"I forced all programmers to use the encapsulated INSERT, instead of writing their own. Using SQLab, we determined that this one insert statement was parsed 1 time and executed over a million times! It has been in the SGA for over 2 weeks, never aging out because it is called so frequently."

```
SELECT owner || '.' || name refs_table,
       REFERENCED_owner || '.' || REFERENCED_name table_referenced
  FROM ALL_DEPENDENCIES
 WHERE type IN ('PACKAGE', 'PACKAGE BODY', 'PROCEDURE', 'FUNCTION')
   AND REFERENCED_type IN ('TABLE', 'VIEW');
```

reftabs.sql

Minimal Encapsulation a Must!

- At an absolute minimum, hide every single row query behind the header of a function
  - If you hide the query, you can choose (and change) the implementation for optimal performance
- No need to argue about implicit vs explicit; the main thing is to encapsulate
- Best approach: put the function in a package, so you can take advantage of package-level data
  - Very useful for data caching mechanisms; by hiding the way you retrieve the data, you allow yourself the freedom to change the retrieval implementation without affecting any usages

Get Me the Name for an ID...

Don't "expose" any SQL...

```
DECLARE
   l_name VARCHAR2(100);
BEGIN
   SELECT last_name || ',' ||
          first_name
     INTO l_name
     FROM employee
    WHERE employee_id =
            employee_id_in;
    ...
END;
```

Instead, hide the query

```
CREATE OR REPLACE PACKAGE te_employee
AS
   SUBTYPE fullname_t IS VARCHAR2
(200);

   FUNCTION fullname (
      l employee.last_name%TYPE,
      f employee.first_name%TYPE
   )
      RETURN fullname_t;

   FUNCTION name (
      employee_id_in IN
         employee.employee_id%TYPE
   )
      RETURN fullname_t;
END;
/
```

And call the function...

```
DECLARE
   l_name te_employee.fullname_t;
BEGIN
   l_name :=
      te_employee.name (
         employee_id_in);
   ...
END;
```

explimpl.pkg

And Never, *Ever* Expose the Dual Table

- The dual table is 100% kluge. It is astonishing that Oracle still relies on it within the STANDARD PL/SQL package
- Always hide queries against the dual table inside a function
  - We need to be optimistic: perhaps in Oracle12*i* the dual table will no longer be necessary

Instead of this...

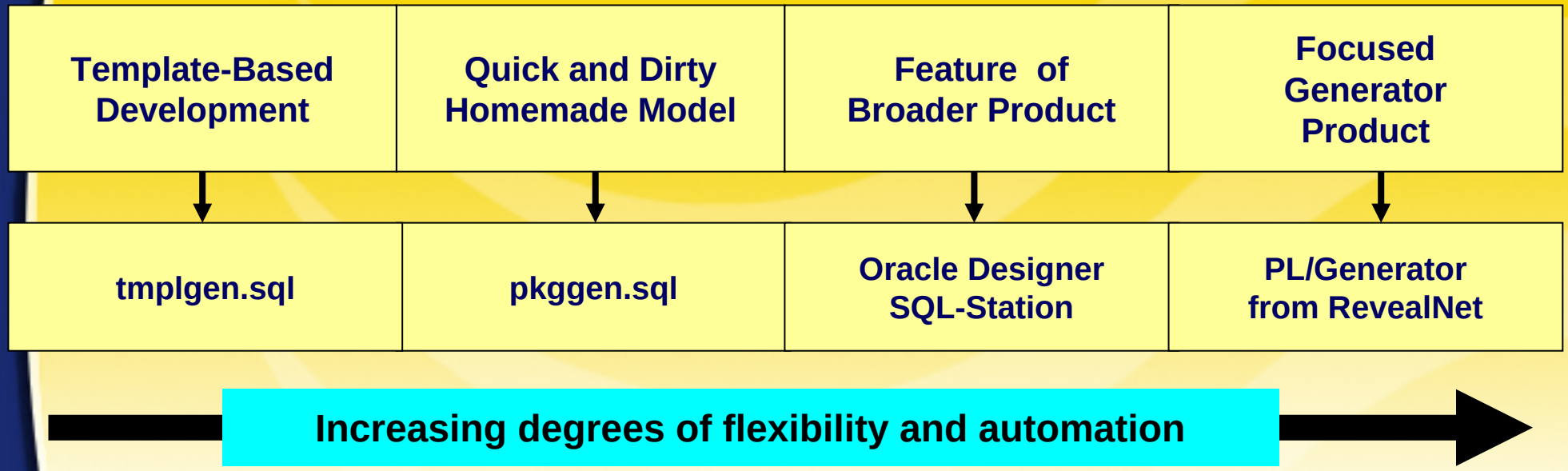
```
BEGIN
   SELECT employee_id_seq.NEXTVAL
     INTO l_employee_id
      FROM dual;
```

Write this:

```
BEGIN
   l_employee_id :=
      te_employee.next_pkey;
```

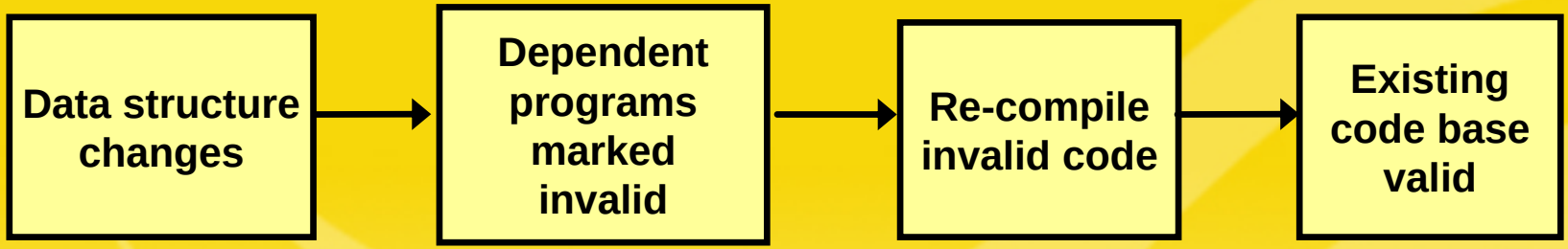
Encapsulation Creation Options

- You can pursue one of the following four models:



- Many challenges to successful encapsulation, including:
  - Write large volumes of high quality code.
  - Train developers to understand and use the API.

Write Code Assuming Change



- Use anchoring to tightly link code to underlying data structures
- Rely on bind variables inside SQL statements
- Fetch into cursor records
- Qualify all references to PL/SQL variables inside SQL statements

Anchor Declarations of Variables

- You have two choices when you declare a variable:
  - Hard-coding the datatype
  - Anchoring the datatype to another structure

Whenever possible, use anchored declarations rather than explicit datatype references

- %TYPE for scalar structures
- %ROWTYPE for composite structures

Hard-Coded Declarations

```
ename VARCHAR2(30);
totsales NUMBER (10,2);
```

Anchored Declarations

```
v_ename emp.ename%TYPE;
totsales pkg.sales_amt%TYPE;

emp_rec emp%ROWTYPE;
tot_rec tot_cur%ROWTYPE;
```

Examples of Anchoring

```
DECLARE
   v_ename emp.ename%TYPE;
   v_totsal config.dollar_amt%TYPE;
   newid config.primary_key;
BEGIN
   . . .
END;
```

The emp table	
ename	VARCHAR2(60)
empno	NUMBER
hiredate	DATE
sal	NUMBER

```
PACKAGE config
IS
   dollar_amt NUMBER (10, 2);

   pkey_var NUMBER(6);

   SUBTYPE primary_key
   IS
      pkey_var%TYPE;

END config;
```

- Use %TYPE and %ROWTYPE when anchoring to database elements
- Use SUBTYPEs for programmatically-defined types
- SUBTYPEs can also be used to mask dependencies that are revealed by %TYPE/%ROWTYPE

PLV.sps
aq.pkg

Benefits of Anchoring

- Synchronize PL/SQL variables with database columns and rows
  - If a variable or parameter does represent database information in your program, always use %TYPE or %ROWTYPE
  - Keeps your programs in synch with database structures without having to make code changes
- Normalize/consolidate declarations of derived variables throughout your programs
  - Make sure that all declarations of dollar amounts or entity names are consistent
  - Change one declaration and upgrade all others with recompilation

Remember: Never Repeat Code!

Quiz: Where's the Hard Coding?

```
1     name VARCHAR2 (30);
2     minbal NUMBER(10,2);
3  BEGIN
4     OPEN company_pkg.allrows (1507);

5     FETCH company_pkg.allrows INTO name, minbal;

6     IF name = 'ACME' THEN ...
```

- Better question for this code: which of these six lines of code do *not* contain an example of hard-coding?

Always Fetch into Cursor Records

wrong

```
   name VARCHAR2 (30);
   minbal NUMBER(10,2);
BEGIN
  OPEN company_pkg.allrows;
  FETCH company_pkg.allrows
     INTO name, minbal;

  IF name = 'ACME' THEN ...

  CLOSE company_pkg.allrows;
```

Fetching into individual variables hard-codes number of items in select list.

right

```
   rec company_pkg.allrows%ROWTYPE;
BEGIN
  OPEN company_pkg.allrows;
  FETCH company_pkg.allrows INTO rec;

  IF rec.name = 'ACME' THEN ...

  CLOSE company_pkg.allrows;
```

Fetching into a record means writing less code.

If the cursor select list changes, it doesn't necessarily affect your code.

Avoid Hard Coding inside SQL

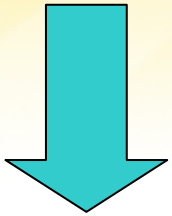
- Don't bury hard-coded values in your SQL statements
  - Instead, move your cursors to a shared area and then rely on that version in all instances
- Here is some inefficient, hard to maintain code:

```
DECLARE
   CURSOR r_and_d_cur IS
      SELECT last_name FROM employee
       WHERE department_id = 10;
```

```
DECLARE
   CURSOR marketing_cur IS
      SELECT last_name FROM employee
       WHERE department_id = 20;
BEGIN
   OPEN marketing_cur;
```

- And what it should be:

```
CREATE OR REPLACE PACKAGE bydept
IS
   CURSOR name_cur (dept IN INTEGER) IS
      SELECT last_name FROM employee
       WHERE department_id = dept;
```



```
BEGIN
   OPEN bydept.name_cur (20);
```

Local variables also avoid multiple parses.

bindvar.sql

QUEST SOFTWARE

Avoiding SQL-PL/SQL Naming Conflicts

- One rule: make sure that you never define variables with same name as database elements
  - OK, you can be sure today, but what about tomorrow?
  - Naming conventions simply cannot offer any guarantee
- Better approach: always qualify references to PL/SQL variables inside SQL statements
  - Remember: you can use labels to give names to anonymous blocks

```
PROCEDURE del_scenario
IS
   reg_cd VARCHAR2(100) := :GLOBAL.reg_cd;
BEGIN
   DELETE FROM scenarios
    WHERE reg_cd = del_scenario.reg_cd
      AND scenario_id = :scenario.scenario_id;
END;
```

No problem!

delscen.sql
delscen1.sql
delscen2.sql

Write SQL Efficiently in PL/SQL

- It's one thing to tune your SQL statements; it is quite another to write your SQL *inside PL/SQL* so that it executes as efficiently as possible

- Use native SQL whenever possible
- Use PL/SQL to replace IO-intensive SQL
- Use the RETURNING Clause
- Use WHERE CURRENT OF
- BULK BIND and COLLECT (Oracle8i)

Use Native SQL Whenever Appropriate

- If you can replace a PL/SQL loop that executes a SQL statement repetitively with a "pure" SQL statement, you are likely to get improved performance
- Instead of this:

```
FOR rec IN (SELECT ename, sal FROM emp)
LOOP
   UPDATE emp SET sal = rec.sal * 1.01
    WHERE ename = rec.ename;
END LOOP;
```

- Do this:

```
UPDATE emp SET sal = sal * 1.01;
```

allsql.tst
allsql2.tst

Use PL/SQL to Avoid IO-Intensive SQL

- 100% Pure SQL -- it's *almost always* the right way to go, but sometimes it results in excessive processing
  - Correlated sub-queries compute same values repeatedly
  - Multiple joins require lots of SORT/MERGE operations

```
SELECT 'Top employee in ' || department_id || ' is ' ||
         E.last_name || ', ' || E.first_name str
  FROM employee E
 WHERE E.salary = (SELECT MAX (salary) FROM employee E2
             WHERE E2.department_id = E.department_id)
```

Instead of one big query...

```
CURSOR dept_cur IS
   SELECT department_id, MAX (salary) max_salary
     FROM employee E GROUP BY department_id;

CURSOR emp_cur (dept IN PLS_INTEGER,maxsal IN NUMBER) IS
   SELECT last_name || ', ' || first_name emp_name
     FROM employee
    WHERE department_id = dept AND salary = maxsal;
```

Break it up and run within PL/SQL loops...

useplsql.tst

QUEST SOFTWARE

Use the RETURNING Clause

- Oracle8 offers a new clause FOR INSERT and UPDATE statements: the RETURNING clause.
  - Retrieve information from DML statement w/o a separate query.
- Instead of this:

```
BEGIN
   INSERT INTO UnionBuster VALUES (ub_seq.NEXTVAL, 'Prison', 5);
   SELECT ub_id, hourly_wage INTO v_latest_bustID, v_hard_to_beat
     FROM UnionBuster
    WHERE labor_type = 'Prison';
END;
```

- Do this:

```
BEGIN
   INSERT INTO UnionBuster VALUES (ub_seq.NEXTVAL, 'Prison', 5)
      RETURNING ub_id, hourly_wage
           INTO v_latest_bustID, v_hard_to_beat;
END;
```

Use WHERE CURRENT OF

- When using SELECT FOR UPDATE, use the WHERE CURRENT OF clause in UPDATE and DELETE to avoid coding a possibly complex and slower WHERE clause.
- Instead of this:

```
LOOP
   FETCH cur INTO rec;
   EXIT WHEN cur%NOTFOUND;

   UPDATE employee SET last_name = UPPER (last_name)
    WHERE employee_id = rec.employee_id;
END LOOP;
```

- Do This:

```
LOOP
   FETCH cur INTO rec;
   EXIT WHEN cur%NOTFOUND;

   UPDATE employee SET last_name = UPPER (last_name)
    WHERE CURRENT OF cur;
END LOOP;
```

wco.sql

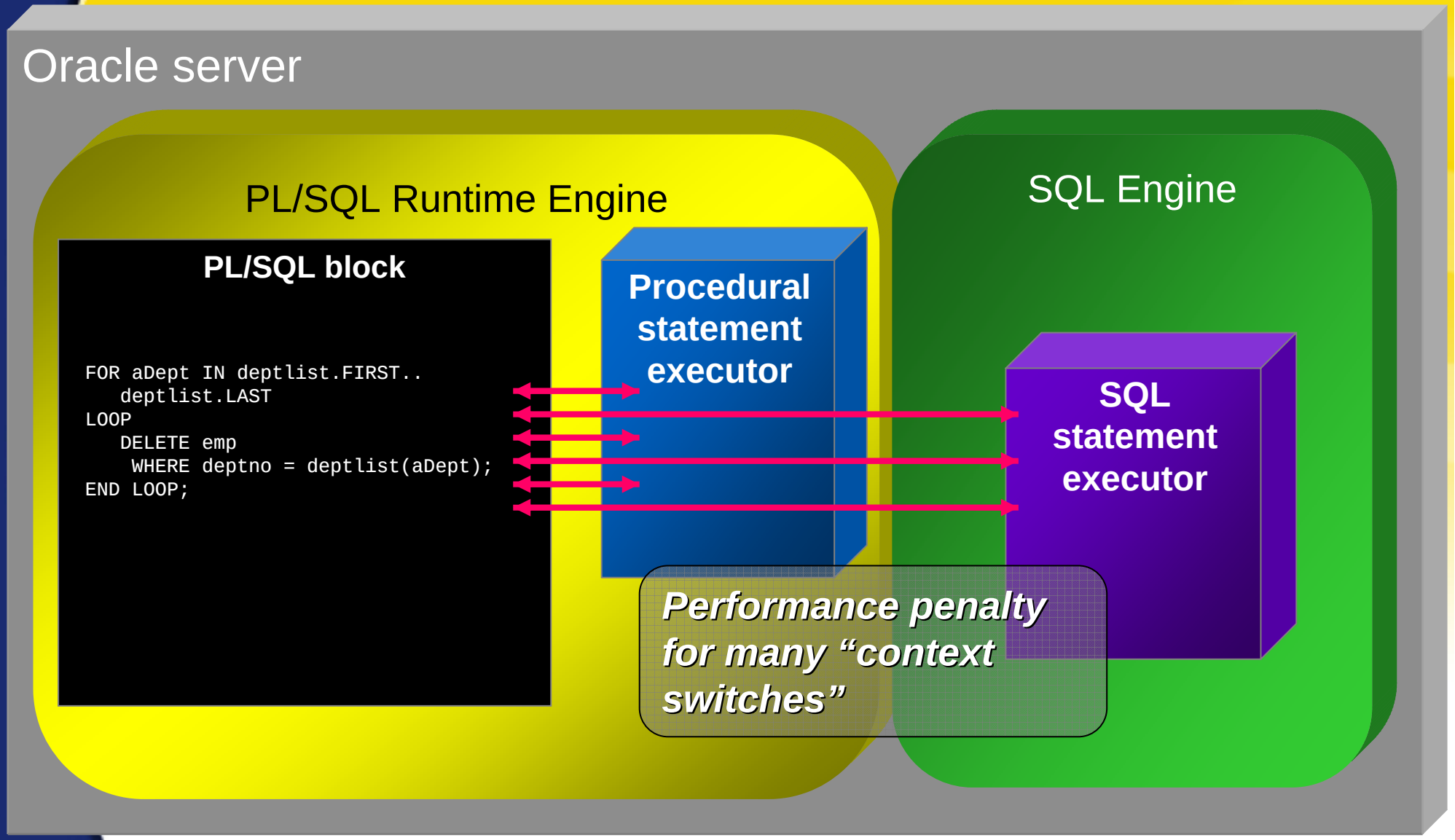
Use Bulk Binding and COLLECT

- Oracle8i offers new syntax to improve the performance of both DML and queries. In Oracle8, updating from a collection (or, in general, performing multi-row DML) meant writing code like this:

```
CREATE TYPE dlist_t AS TABLE OF INTEGER;
/

PROCEDURE whack_emps_by_dept (deptlist dlist_t)
IS
BEGIN
   FOR aDept IN deptlist.FIRST..deptlist.LAST
   LOOP
      DELETE emp WHERE deptno = deptlist(aDept);
   END LOOP;
END;
```

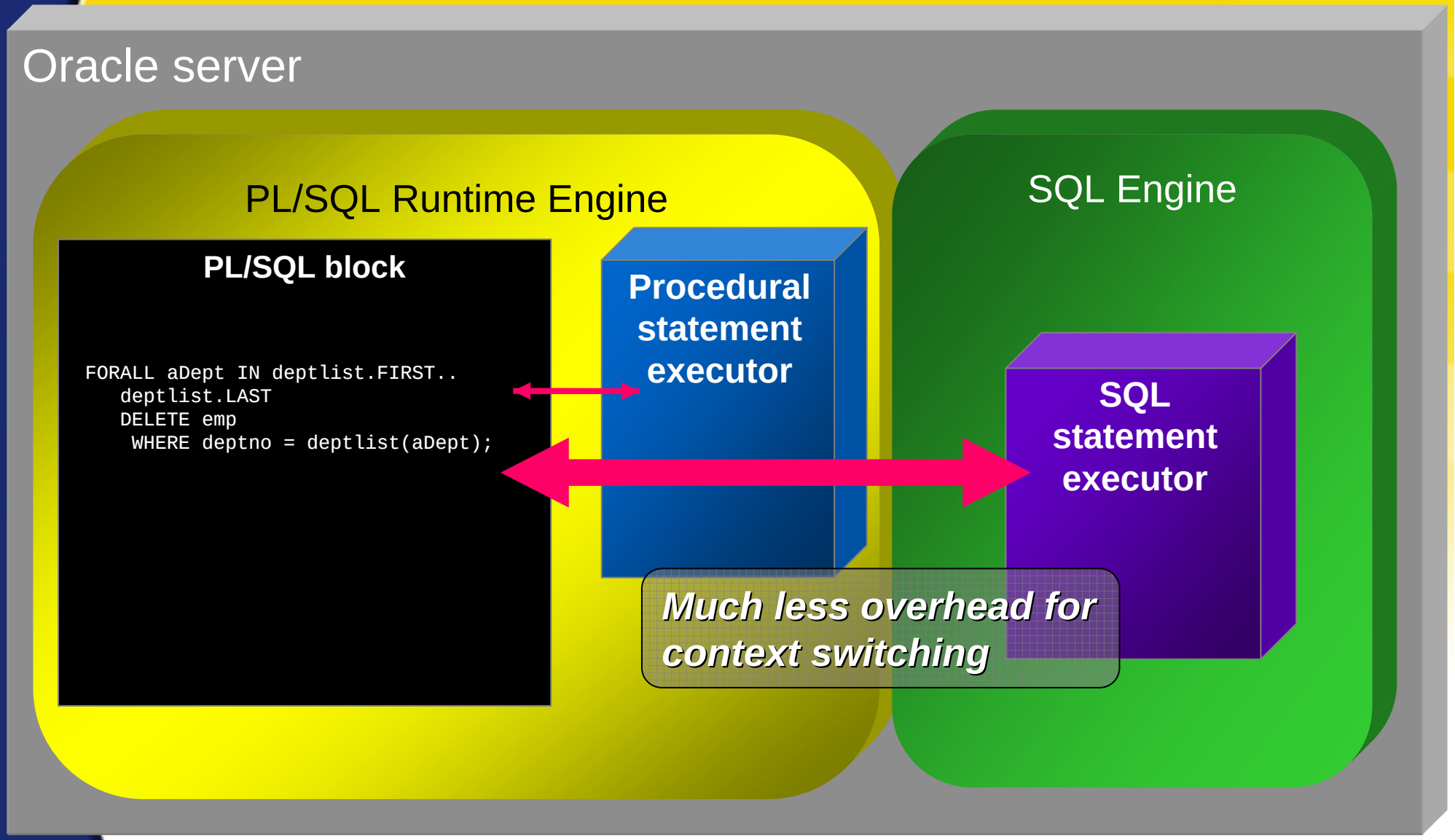
"Conventional bind" (and lots of them!)

Conventional Bind



QUEST SOFTWARE

Enter the “Bulk Bind”



QUEST SOFTWARE

Use the FORALL Bulk Bind Statement

- Instead of the individual DML operations, you can do this:

```
PROCEDURE whack_emps_by_dept (deptlist dlist_t)
IS
BEGIN
   FORALL aDept IN deptlist.FIRST..deptlist.LAST
      DELETE FROM emp WHERE deptno = deptlist(aDept);
END;
```

- Some restrictions:
  - Only the single DML statement is allowed. If you want to INSERT and then UPDATE, two different FORALL statements
  - Cannot put an exception handler on the DML statement

Use BULK COLLECT for Queries

- BULK COLLECT performs bulk bind of results from SQL select statement
  - Returns each selected expression in a table of scalars

```
CREATE OR REPLACE FUNCTION get_a_mess_o_emps
   (deptno_in IN dept.depno%TYPE)
RETURN emplist_t
IS
   emplist emplist_t := emplist_t();
   TYPE numTab IS TABLE OF NUMBER;
   TYPE charTab IS TABLE OF VARCHAR2(12);
   TYPE dateTab IS TABLE OF DATE;
   enos numTab;
   names charTab;
   hdates dateTab;
BEGIN
   SELECT empno, ename, hiredate
      BULK COLLECT INTO enos, names, hdates
      FROM emp
     WHERE deptno = deptno_in;
   emplist.EXTEND(enos.COUNT);
   FOR i IN enos.FIRST..enos.LAST
   LOOP
      emplist(i) := emp_t(enos(i),
         names(i), hiredates(i));
   END LOOP;
   RETURN emplist;
END;
```

Combining FORALL & BULK COLLECT

- Use the RETURNING clause to obtain information about each of the DML statements executed in the FORALL
  - Since you are executing multiple DML statements, you need to BULK COLLECT the RETURNING results into one or more collections

```
FUNCTION whack_emps_by_dept (deptlist dlist_t)
   RETURN enolist_t
IS
   enolist enolist_t;
BEGIN
   FORALL aDept IN deptlist.FIRST..deptlist.LAST
      DELETE FROM emp WHERE deptno IN deptlist(aDept)
         RETURNING empno BULK COLLECT INTO enolist;
   RETURN enolist;
END;
```

bulkcoll.sql
bulktiming.sql.
bulkcollect91.sql

Tips and Fine Points

- Use bulk binds if you write code with these characteristics:
  - Recurring SQL statement in PL/SQL loop
  - Use of a collection as the bind variable, or code that could be transformed to use a collection containing the bind variable information
- Bulk bind rules:
  - Can be used with any kind of collection
  - Collection subscripts cannot be expressions
  - The collections must be densely filled
  - If error occurs, prior successful DML statements are NOT ROLLED BACK
- Bulk collects:
  - Can be used with implicit and explicit cursors
  - Collection is filled starting at row 1

Optimizing DBMS_SQL Usage

- DBMS_SQL implements dynamic SQL in PL/SQL
  - It is a very complex and difficult to use package
- Keep in mind the following tips when working with DBMS_SQL:
  - Re-parse only when absolutely necessary
  - Allocate new cursors only when necessary
  - Close dynamic SQL cursors when done, and also in exception sections
  - Always choose binding over concatenation
  - Use bulk processing features of Oracle8

effdsql.tst

Manage Dynamic SQL Cursors

- If you have already allocated memory for a cursor and assigned the value to a PL/SQL variable, you can parse multiple statements against that same cursor
  - Don't open and close cursors unnecessarily
- Make sure you close DBMS_SQL cursors when done and also in exception sections
  - They are *not* closed automatically when the block terminates
  - You will often want to close cursors on exceptions in the parse phase
- Best solution: encapsulate calls to DBMS_SQL.OPEN_CURSOR and also DBMS_SQL.PARSE

openprse.pkg

QUEST SOFTWARE

Choose Binding over Concatenation

- You *can* concatenate rather than bind, but binding is almost always preferable. Two key reasons:
  - Simpler code to build and maintain
  - Improved application performance
- Simpler code to build and maintain
  - Concatenation results in much more complicated and error-prone code unless you are doing a very simple operation
- Improved application performance
  - Concatenates requires an additional call to DBMS_SQL.PARSE and also increases the likelihood that the SQL statement will be physically different, requiring an actual re-parsing and unnecessary SGA utilization
- Note: you cannot bind schema elements, like table names

effdsql.tst
updnval2.sp
updnval3.sp

Use Bulk Processing Features

- Similar to the FORALL and COLLECT features of Oracle8i, DBMS_SQL as of Oracle8 allows you to specify the use of "arrays", i.e., index tables, when you perform updates, inserts, deletes and fetches
- Instead of providing a scalar value for an operation, you specify an index table. DBMS_SQL then repeats your action for every row in the table
- It really isn't "array processing"
  - In actuality, DBMS_SQL is executing the specified SQL statement N times, where N is the number of rows in the table
- This technique still, however, can offer a significant performance boost over Oracle7 dynamic SQL
  - And in some cases give you better performance than static SQL

Array-Oriented Actions

- You will take these steps when working with arrays or collections in DBMS_SQL:
  - Bind arrays of values with DBMS_SQL.BIND_ARRAY
  - Define a column as an array with DBMS_SQL.DEFINE_COLUMN
  - Fetch multiple rows with one call with DBMS_SQL.FETCH_ROWS
  - Retrieve multiple column values with DBMS_SQL.COLUMN_VALUE and DBMS_SQL.VARIABLE_VALUE
- We will look at performing updates, inserts and deletes, then finish up with queries
- These examples are explained and explored thoroughly in chapter 3 of Oracle Built-in Packages (O'Reilly & Associates)

An Example with Bulk Deletes

- When you delete using arrays, you will specify the values in the WHERE clause for all rows to be deleted

```
CREATE OR REPLACE PROCEDURE delemps
   (enametab IN DBMS_SQL.VARCHAR2_TABLE)
IS
   cur PLS_INTEGER := PLVdyn.open_and_parse (
     'DELETE FROM emp WHERE ename LIKE UPPER (:ename)');

   fdbk PLS_INTEGER;
BEGIN
   DBMS_SQL.BIND_ARRAY (cur, 'ename', enametab);

   fdbk := DBMS_SQL.EXECUTE (cur);

   p.l ('Rows deleted', fdbk);

   DBMS_SQL.CLOSE_CURSOR (cur);
END;
```

dyndel.sp
dyndel.tst

Returning Values After Execution

- With Oracle8, you can include a RETURNING clause
  - Obtain info about rows just modified without issuing extra query

```
CREATE OR REPLACE PROCEDURE delemps (
   enametab IN DBMS_SQL.VARCHAR2_TABLE)
IS
   cur PLS_INTEGER := PLVdyn.open_and_parse (
     'DELETE FROM emp WHERE ename LIKE UPPER (:ename) ' ||
     ' RETURNING empno INTO :empnos');
   empnotab DBMS_SQL.NUMBER_TABLE;
   fdbk PLS_INTEGER;
BEGIN
   DBMS_SQL.BIND_ARRAY (cur, 'ename', enametab);
   DBMS_SQL.BIND_ARRAY (cur, 'empnos', empnotab);
   fdbk := DBMS_SQL.EXECUTE (cur);
   DBMS_SQL.VARIABLE_VALUE (cur, 'empnos', empnotab);

   FOR indx IN empnotab.FIRST .. empnotab.LAST LOOP
      p.l (empnotab(indx));
   END LOOP;
   DBMS_SQL.CLOSE_CURSOR (cur);
END;
```

Notice use of VARIABLE_VALUE

dyndel2.sp
dyndel2.tst

Example of Bulk Fetch -- One Pass

- I know the table is small, so I grab all the rows at once

```
CREATE OR REPLACE PROCEDURE showall IS
   cur PLS_INTEGER;
   fdbk PLS_ INTEGER;
   empno_tab DBMS_SQL.NUMBER_TABLE;
   hiredate_tab DBMS_SQL.DATE_TABLE;
BEGIN
   cur := PLVdyn.oap ('SELECT empno, hiredate FROM emp');

   DBMS_SQL.DEFINE_ARRAY (cur, 1, empno_tab, 100, 1);
   DBMS_SQL.DEFINE_ARRAY (cur, 2, hiredate_tab, 100, 1);

   fdbk := DBMS_SQL.EXECUTE_AND_FETCH (cur);

   DBMS_SQL.COLUMN_VALUE (cur, 1, empno_tab);
   DBMS_SQL.COLUMN_VALUE (cur, 2, hiredate_tab);

   FOR rowind IN empno_tab.FIRST .. empno_tab.LAST
   LOOP
      p.l (empno_tab(rowind)); p.l (hiredate_tab(rowind));
   END LOOP;
   DBMS_SQL.CLOSE_CURSOR (cur);
END;
```

arrayemp.sp

Fetching N Records at a Time

```
DBMS_SQL.DEFINE_ARRAY (cur, 1, empno_tab, numrows, 1);
DBMS_SQL.DEFINE_ARRAY (cur, 2, hiredate_tab, numrows, 1);

fdbk := DBMS_SQL.EXECUTE (cur);
LOOP
   fdbk := DBMS_SQL.FETCH_ROWS (cur);

   DBMS_SQL.COLUMN_VALUE (cur, 1, empno_tab);
   DBMS_SQL.COLUMN_VALUE (cur, 2, hiredate_tab);

   EXIT WHEN fdbk < numrows;
END LOOP;
```

arrayempN.sp

- What if you do not know how many rows will be fetched?
  - In this case, you fetch N rows at a time inside a loop and then exit when there are no more to fetch. The above example shows the kind of loop you would write
- Note: rows are *appended* to the index tables. You must DELETE in between fetches to clear them out (unless you want to fill the table and process the data later)

Update Rows En Masse From File

- This examples shows how to do a bulk load of data from a file into SQL
  - Data file contains key (employee number) and new salary
  - Use UTL_FILE to read the contents of the file and load it into arrays
  - Use Oracle8 DBMS_SQL to update in a single pass

```
CREATE OR REPLACE PROCEDURE upd_from_file
   (loc IN VARCHAR2, file IN VARCHAR2)
IS
   cur PLS_INTEGER := DBMS_SQL.OPEN_CURSOR;
   fdbk PLS_INTEGER;
   empnos DBMS_SQL.NUMBER_TABLE;
   sals DBMS_SQL.NUMBER_TABLE;

   fid UTL_FILE.FILE_TYPE;
   v_line VARCHAR2(2000);
   v_space PLS_INTEGER;
```

Declare DBM_SQL arrays and UTL_FILE pointer.

fileupd.sp
fileupd.dat

Quiz!

- Some people get rich by laying off other people
  - Write code to show the CEO(s) who laid off the most people, and the CEO(s) who laid off the second-highest number of people
  - You *could* do it all in pure SQL...but would you want to?

```
SELECT name || ' of ' || company Slasher1
  FROM ceo_compensation
 WHERE layoffs =
       (SELECT MAX (layoffs)
          FROM ceo_compensation);

SELECT name || ' of ' || company Slasher2
  FROM ceo_compensation
 WHERE layoffs =
       (SELECT MAX (layoffs)
          FROM ceo_compensation
         WHERE layoffs !=
                  (SELECT MAX (layoffs)
                     FROM ceo_compensation));0
```

slowsql_q1.sql
slowsql_a1.sql
slowsql_a1.tst

Quiz!

- This program is running slowly. How can I improve it?
  - This is a test of analyzing algorithms for unnecessary and/or slow program performance, *and* tuning of DBMS_SQL code

```
CREATE OR REPLACE PROCEDURE insert_many_emps
IS
   cur INTEGER := DBMS_SQL.open_cursor;
   rows_inserted INTEGER;

BEGIN
   DBMS_SQL.parse (cur,
      'INSERT INTO emp (empno, deptno, ename)
         VALUES (:empno, :deptno, :ename)',
      DBMS_SQL.native);

   FOR rowind IN 1 .. 1000
   LOOP
      DBMS_SQL.bind_variable (cur, 'empno', rowind);
      DBMS_SQL.bind_variable (cur, 'deptno', rowind * deptno);
      DBMS_SQL.bind_variable (cur, 'ename', 'Steven' || rowind);
      rows_inserted := DBMS_SQL.execute (cur);
   END LOOP;

   DBMS_SQL.close_cursor (cur);
END;
```

loadlots*.*

QUEST SOFTWARE

Writing SQL in PL/SQL
Summary

- Never Repeat SQL
  - Maximize performance, minimize impact of change
- Anchor Variables Whenever Possible
  - You're almost never doing something for the first time
  - If you are, let that be the prototype for all others
- Take a Serious Look at Table Encapsulation
  - It's a worthwhile investment for the future of your applications

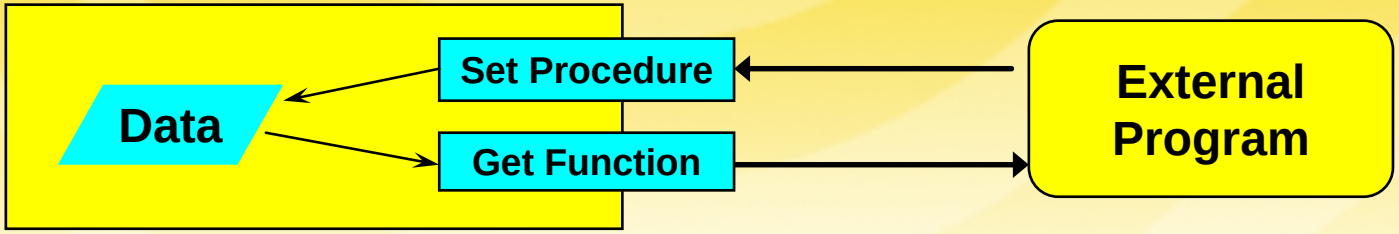
PL/SQL Tuning & Best Practices

Package Construction

- Hide package-level data inside the package body
- Build toggles and windows to increase usability and flexibility of your packages
- Tips for managing large packages

Hide Package Data!

- *Never* put your variables and other data structures in the package specification
  - Always put it in the body. Then build publicly-available "get and set" programs to change values in the data structures and retrieve the current values



- The benefits include:
  - Tighter control over data structures. If your data is public, it can be changed by any program with EXECUTE authority
  - Flexibility to change implementation of data structure
  - Ability to track access to the data

Public vs. Private Data

```
PACKAGE P_and_L
IS
   last_stmt_dt DATE;
END P_and_L;
```

Public data, declared in package specification, directly accessible.

Tip: Use PLVgen.gas to generate code like this.

Now in the body, with get-and-set routines.

And a business rule: date cannot be in the future.

```
PACKAGE BODY P_and_L
IS
   last_stmt_dt DATE;

   FUNCTION last_date RETURN DATE IS
   BEGIN
      RETURN last_stmt_dt;
   END;

   PROCEDURE set_last_date (date_in IN DATE)
   IS
   BEGIN
      last_stmt_dt := LEAST (date_in, SYSDATE);
   END;
END P_and_L;
```

Different Access Paths to Data

The public approach exposes fully the variable and allows for violation of the business rule. Below I move the last statement date into the future.

```
IF P_and_L.last_stmt_dt <
   ADD_MONTHS (SYSDATE, -3)
THEN
   P_and_L.last_stmt_dt := SYSDATE + 12;
END IF;
```

And a business rule is violated!

The private version of the same code hides the variable completely and protects against violations of the rules.

```
IF P_and_L.last_date <
   ADD_MONTHS (SYSDATE, -3)
THEN
   P_and_L.set_last_date (SYSDATE + 12);
END IF;
```

Tracking Variable Reads and Writes

- When you use get-and-sets to control access to your data, good things just naturally start coming your way
  - For example, you can easily add a trace to your code to show when, where and how a specific variable's value is read and/or changed. Wow!
  - If you have exposed the variable in the specification of the package, it would be very difficult to perform this kind of trace
  - A PL/SQL debugger might someday do this
  - Or maybe you could examine the source code stored in the USER_SOURCE view. A fairly tedious process
- But with get-and-sets, you have given yourself a "hook" on which to hang your trace
  - Since the variable can only be accessed through these modules, you can place your trace inside the procedure and function
  - The package structure then guarantees that you will have caught every access

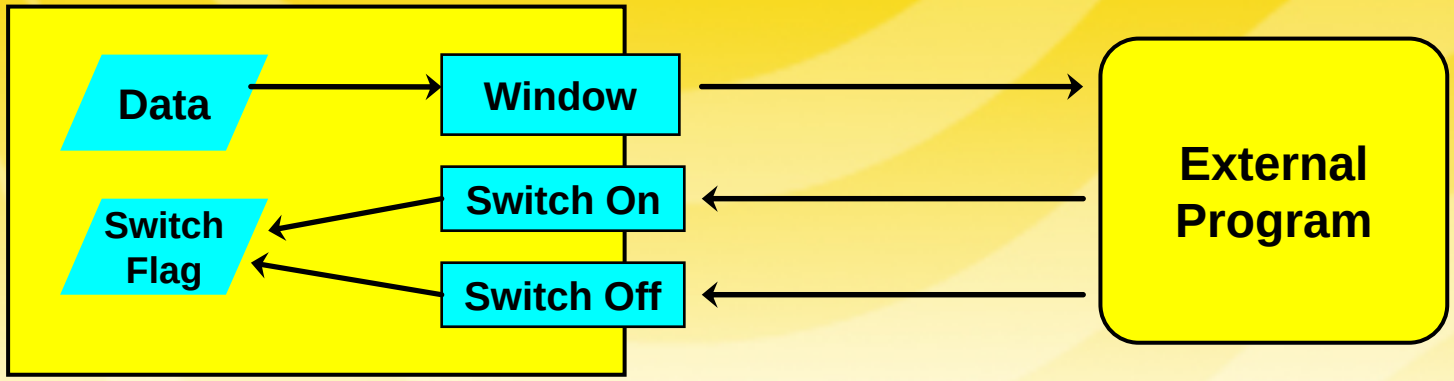
Tracking Changes to P&L Date

- With just a few, quick changes, the p_and_l package now uses PLVxmn to show any attempts to access the variable
- In this case, the PLVxmn package itself offers the ability to display the module which called the package
- We will learn how to build a trace package in the next section

```
PACKAGE BODY P_and_L
IS
   last_stmt_dt DATE;

   FUNCTION last_date RETURN DATE IS
   BEGIN
      PLVxmn.trace
         ('last_date', 1, last_stmt_dt);

      RETURN last_stmt_dt;
   END;

   PROCEDURE set_last_date (date_in IN DATE)
   IS
   BEGIN
      PLVxmn.trace
         ('set_last_date', 2, date_in);

      last_stmt_dt :=
         LEAST (date_in, SYSDATE);
   END;
END P_and_L;
```

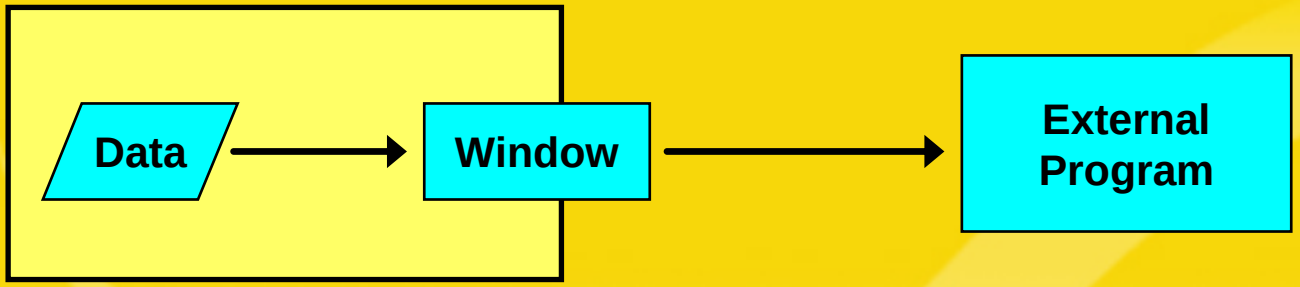
Build Toggles and Windows

- You will greatly increase the usability and flexibility of your packages if you build toggles and windows into the package interface (specification)



- Toggles are...
  - On/off switches that allow you to modify the behavior of the package without changing its code
- Windows offer...
  - Read-only access to the inner workings of a package

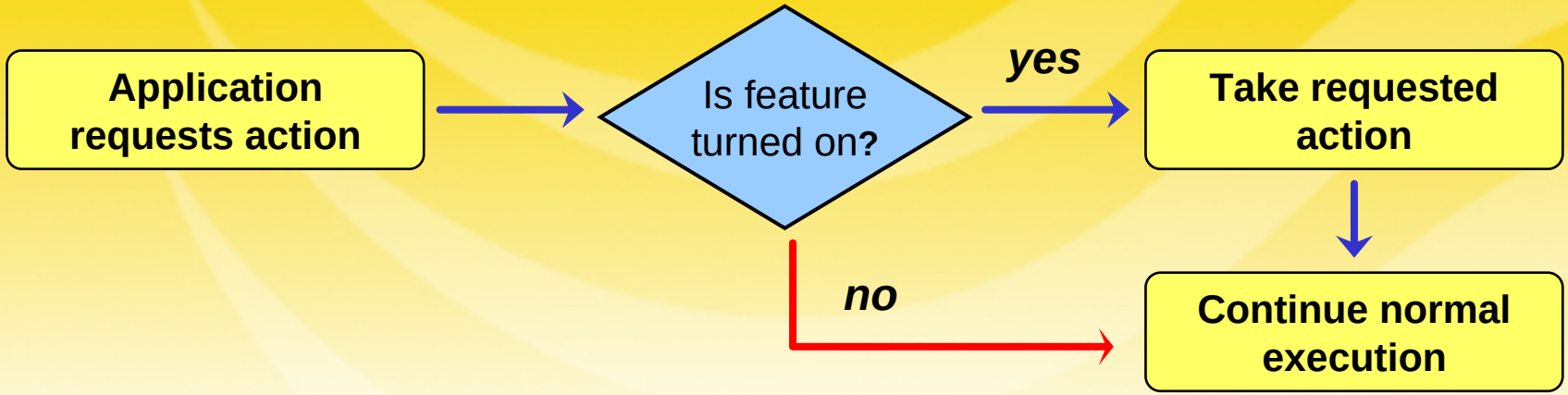
A Window Offers Safe Viewing



- Normally, the contents of the package body (private data, implementation of code) are completely hidden from a user of the package
  - This "black box" protects the integrity of the data and encourages top-down design and problem-solving, but it can leave you in the dark when you run and test your code
- Maintain package integrity and shed light on internal package functioning by providing *read-only* glimpses off package activity
  - Developers open and close the window, but they can't "break and enter"

A Toggle Offers Flexibility

- A toggle is an on-off switch that allows a user to modifies the behavior of programs inside the package
  - Without making any code changes to that package



- Toggles are often used in conjunction with Windows, as the next example will demonstrate

Toggle & Window Example: Variable Trace

- A variable trace offers a fine example of a toggle and a window

```
BEGIN
   P_and_L.set_stmt_date (SYSDATE + 12);

   IF P_and_L.stmt_date > SYSDATE - 4
   THEN
      P_and_L.set_stmt_date (SYSDATE - 1);
   END IF;
END;
```

A dozen programs touch your data. Which one is causing the problem?

- I want to be able to watch both reads from and writes to a variable, and see:
  - The current value of the variable
  - The new value of the variable if it is being changed
  - What program is touching the variable
- What are my options?

Adding Trace Before Each Touch

```
BEGIN
   DBMS_OUTPUT.PUT_LINE (SYSDATE + 12);
   P_and_L.set_stmt_date (SYSDATE + 12);

   DBMS_OUTPUT.PUT_LINE (P_and_L.stmt_date);
   IF P_and_L.stmt_date > SYSDATE - 4
   THEN
      DBMS_OUTPUT.PUT_LINE (SYSDATE - 1);
      P_and_L.set_stmt_date (SYSDATE - 1);
   END IF;
END;
```

- This approach is full of drawbacks, including:
  - Reduces productivity and code quality. You must write the same code repeatedly (and remember to do it). You must then take out those calls when done testing, etc.
  - DBMS_OUTPUT.PUT_LINE is, in and of itself, problematic. For example, what if the SQL string contains more than 255 characters? What if you want to write to a pipe?

A Better Idea: Put Trace Inside "Touch"

- Bundle the trace into each program that reads from or writes to the variable
  - Requires no additional coding for users, except to turn on the trace when needed
- Combine the window with a toggle to turn it on and off
  - After all, you don't *always* want to see this information

```
CREATE OR REPLACE PACKAGE p_and_l
IS
   PROCEDURE set_stmt_date (date_in IN DATE);
   FUNCTION stmt_date RETURN DATE;

   PROCEDURE trc;
   PROCEDURE notrc;
   FUNCTION tracing RETURN BOOLEAN;
END;
```

Get and Set programs for hidden date variable.

Turn trace on or off, return current setting.

p_and_l.pkg
watch.pkg

Trace Window & Toggle Implementation

```
CREATE OR REPLACE PACKAGE BODY p_and_l
IS
   g_trace BOOLEAN := FALSE;

   PROCEDURE trc IS
   BEGIN
      g_trace := TRUE;
   END;

   PROCEDURE notrc IS
   BEGIN
      g_trace := FALSE;
   END;

   FUNCTION tracing RETURN BOOLEAN IS
   BEGIN
      RETURN g_trace;
   END;
                  /* MORE ON NEXT PAGE
*/
END dynsql;
```

Global Boolean Flag

Procedures set the flag's value, while the function returns the current value.

Trace Window & Toggle Implementation

```
CREATE OR REPLACE PACKAGE BODY p_and_l
IS
 PROCEDURE set_stmt_date (date_in IN DATE)
   IS
   BEGIN
      IF tracing
      THEN
         watch.action ('set_stmt_date', date_in);
      END IF;

      g_stmt_date := date_in;
   END;
END p_and_l;
```

Check status of toggle.

Rely on separate watch package.

- By creating a separate package to do the watching, you have a reusable utility that also hides the implementational details, giving you lots of flexibility

The Watch Package Specification

```
CREATE OR REPLACE PACKAGE watch
IS
   PROCEDURE toscreen;
   PROCEDURE topipe;

   PROCEDURE action (prog IN VARCHAR2, val IN BOOLEAN);
   PROCEDURE action (prog IN VARCHAR2, val IN DATE);
   PROCEDURE action (prog IN VARCHAR2, val IN NUMBER);
   PROCEDURE action (prog IN VARCHAR2, val IN VARCHAR2);

   PROCEDURE show;
END;
```

Select the output type.

Overloaded watch programs.

- You have to know when to draw the line
  - Watch does not try to determine automatically the program which asked for the watch (check out PLVcs for a package that can do this)
  - Instead, it will just call DBMS_UTILITY.FORMAT_CALL_STACK

watch.pkg

The Watch Package Body

```
CREATE OR REPLACE PACKAGE BODY watch
IS
   c_pipe_name CONSTANT VARCHAR2(3) := 'watch$trc';

   c_screen CONSTANT INTEGER := 0;
   c_pipe   CONSTANT INTEGER := 1;
   g_target INTEGER := 0;

   PROCEDURE toscreen
   IS
   BEGIN
      g_target := c_screen;
   END;

   PROCEDURE topipe
   IS
   BEGIN
      g_target := c_pipe;
   END;
```

Data to manage output type.

Request a write to screen.

Request a write to pipe.

The Watch Package Body

```
PROCEDURE action (prog IN VARCHAR2, val IN VARCHAR2)
IS
   stat INTEGER;
   msg PLV.dbmaxvc2;
BEGIN
   msg := ' Time: ' || PLV.now || ' Context: ' || prog ||
      ' Message: ' || val ||
      ' Callstack ' || DBMS_UTILITY.FORMAT_CALL_STACK;

   IF g_target = c_screen
   THEN
      p.l (msg);

   ELSIF g_target = c_pipe
   THEN
      DBMS_PIPE.RESET_BUFFER;
      DBMS_PIPE.PACK_MESSAGE (msg);
      stat := DBMS_PIPE.SEND_MESSAGE (c_pipe_name, timeout => 0);
   END IF;
END;
```

Put together the watch string.

Write to screen.

Write to pipe.

Managing Package Code

- Modularize and re-use internal package elements to reduce code volume and improve maintainability
- Split up large packages into separate, smaller packages -- and hide the split

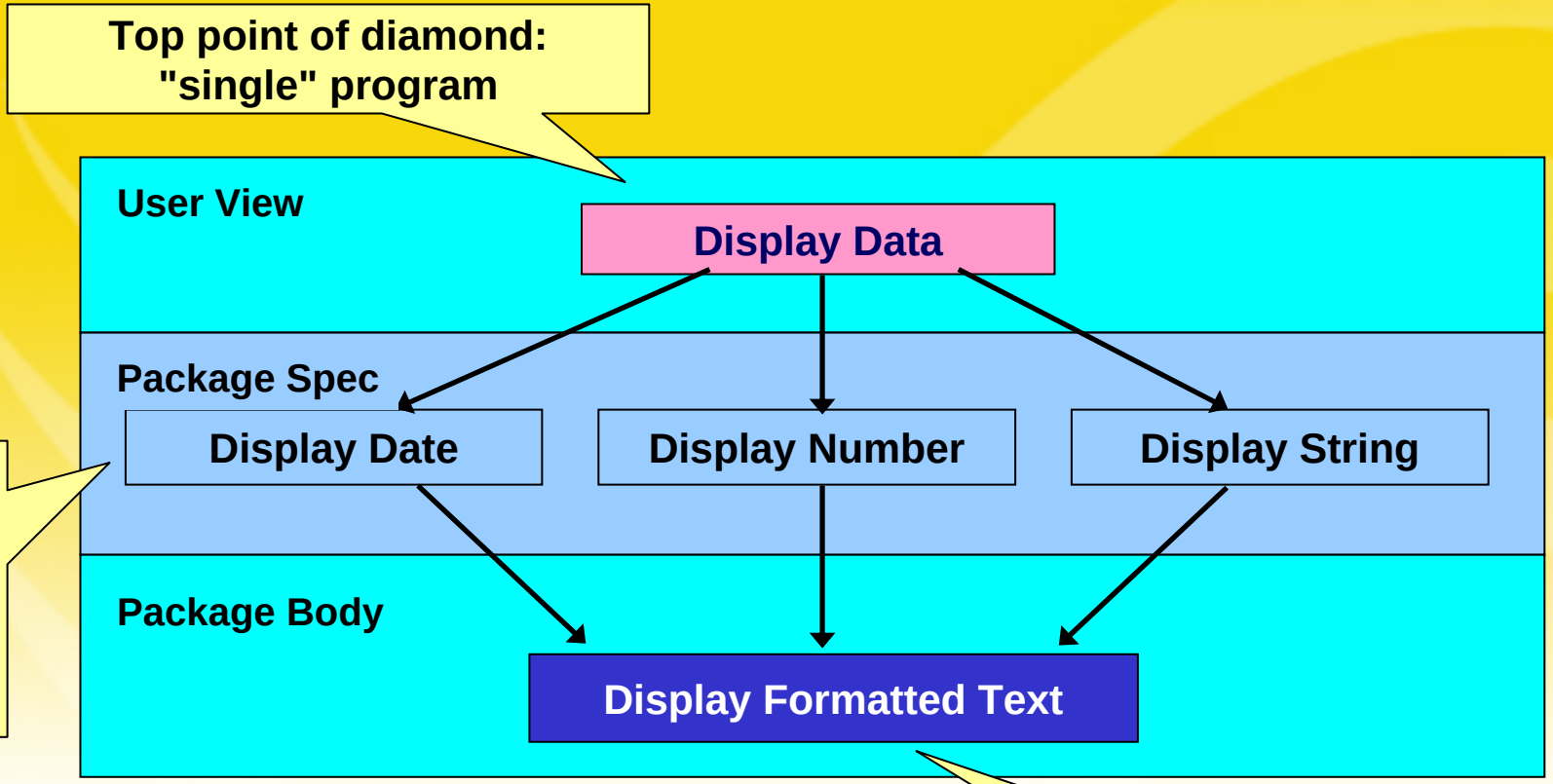
Modularize Within Your Package

- Modularizing your code is *always* important, but inside package bodies modularization is more crucial than ever before
  - And it is also often neglected
  - Ugly package bodies can be hidden behind pretty (well-designed) interfaces
  - That works (if you're lucky) for the first production roll-out. Maintenance is, on the other hand, an unqualified nightmare

Modularize Within Your Package (Cont.)

- Package bodies can be very hard to control and maintain, particularly because of the primitive nature of PL/SQL editors
  - Overloading, in particular, carries the potential for greatly increased code volume and lots of code redundancy
  - As you deepen your PL/SQL skills and take on more complex application development challenges, you must also deepen your commitment to properly modularizing your code
  - Perhaps you should remember that "diamonds are forever"...

The Overloading Diamond



Your challenge: to pull all the facets back to a point, a single implementation.

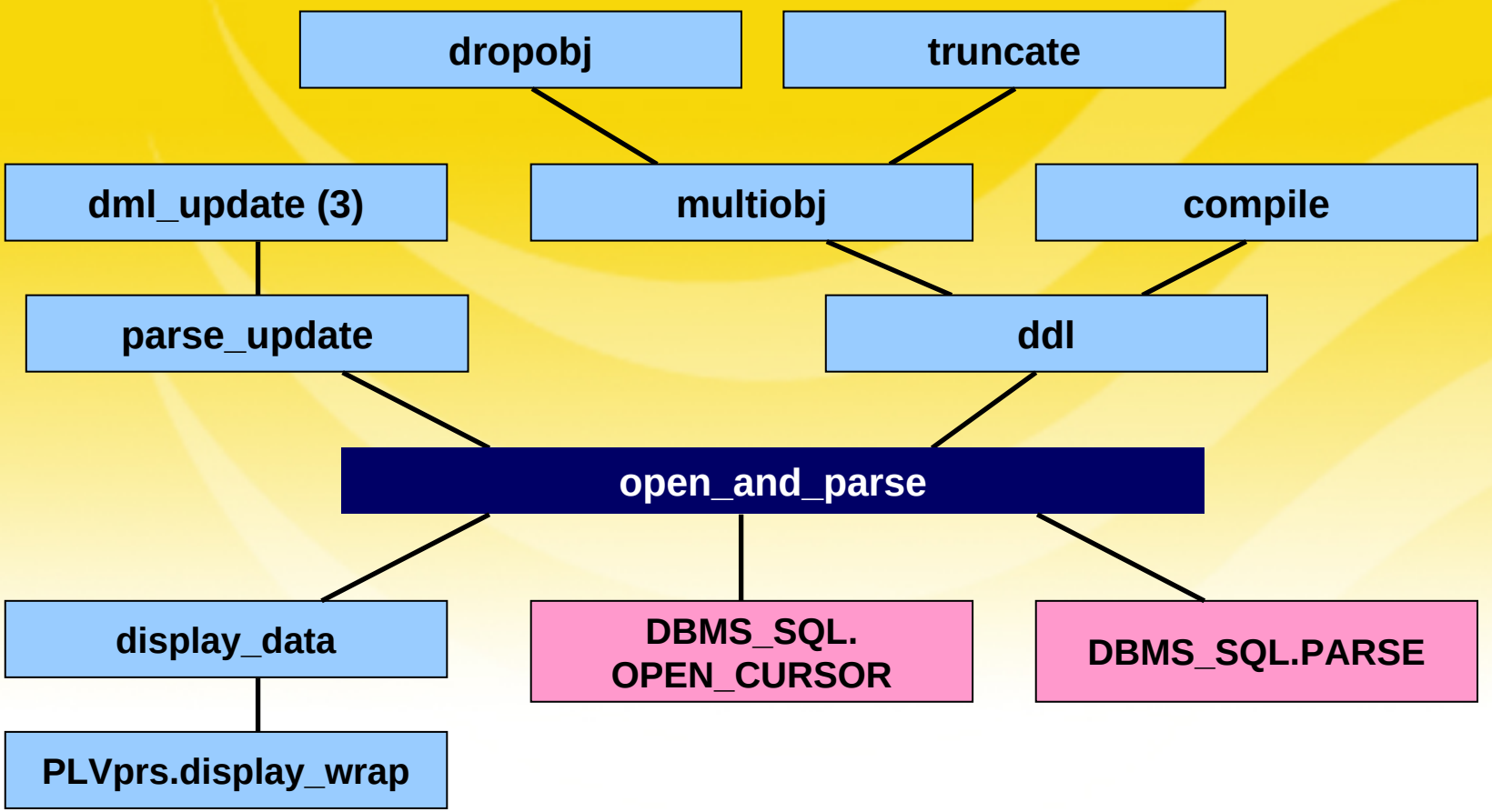
p.spb

QUEST SOFTWARE

Avoiding Redundancy In the Package

- Sometimes the need to modularize is obvious
  - Large-scale overloading literally begs for a consolidated implementation
  - In other situations the pattern and need is less clear, but the long-term implications if you do not bother to modularize can be severe
- You should be an unabashed fanatic when it comes to avoiding redundant code in your package body
  - It might seem like extra work at first, but as scope creeps and enhancement requests arrive, you will find yourself very thankful for the initial effort.
  - Follow that deceptively simple rule: never repeat a line of code. Instead, build a private (body-only) function or procedure to encapsulate the logic

Code Reuse in the PLVdyn Package



QUEST SOFTWARE

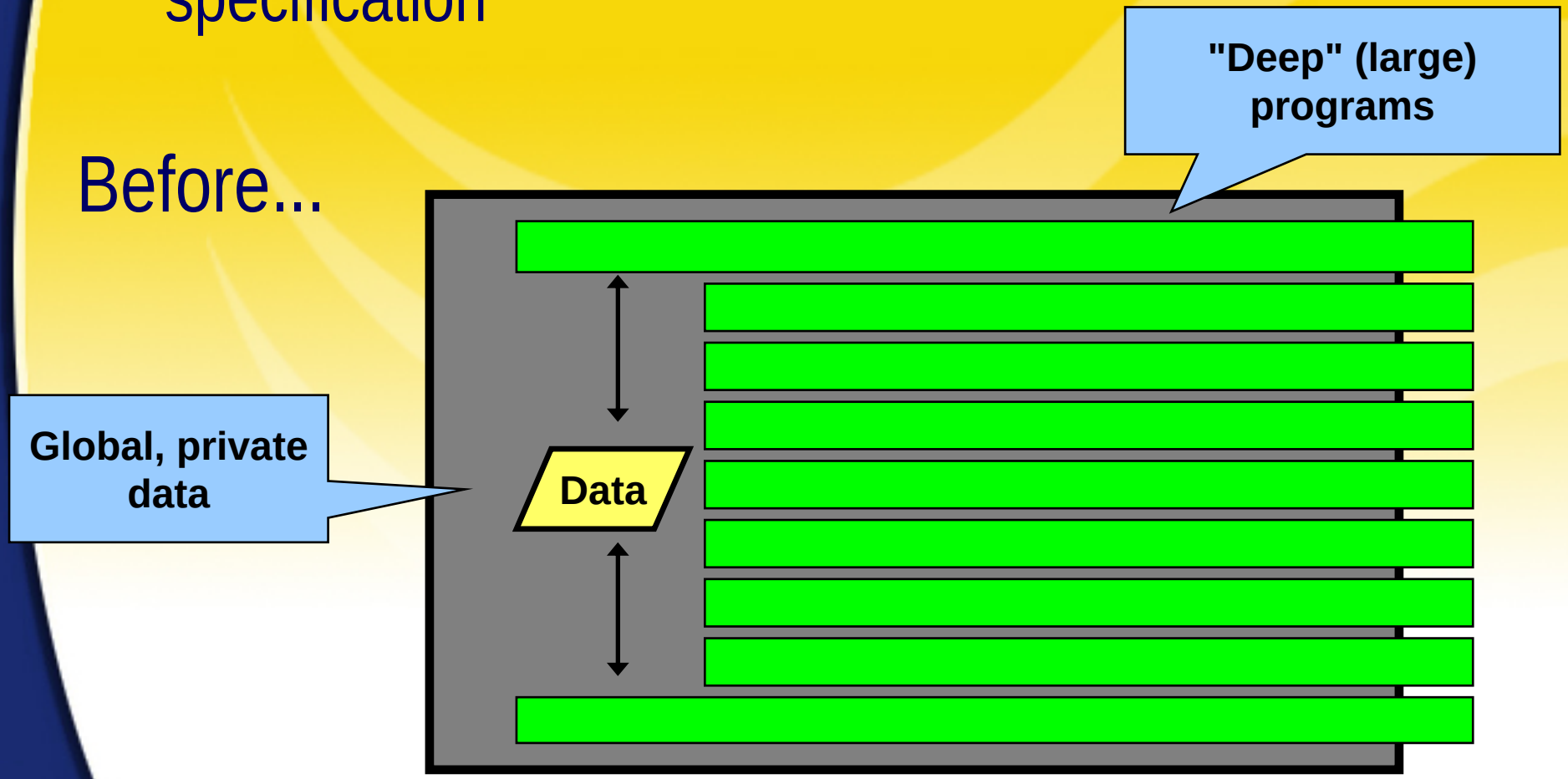
Splitting up Large Packages

- It is certainly possible to create stored code -- especially packages -- which become too large
  - Too large to compile
  - Too large to maintain effectively
  - Too large to enhance in a reasonable span of time
- What can you do when you have created "monsters"?
  - Modularize ruthlessly, avoiding any kind of code repetition by using nested modules or private programs in the body
  - You can also split up large packages into separate program units to make sure that you can compile -- and incrementally compile -- without lengthy delays

Lots of Code, Lots of Headaches

- The problem is with the body (all that code), not the specification

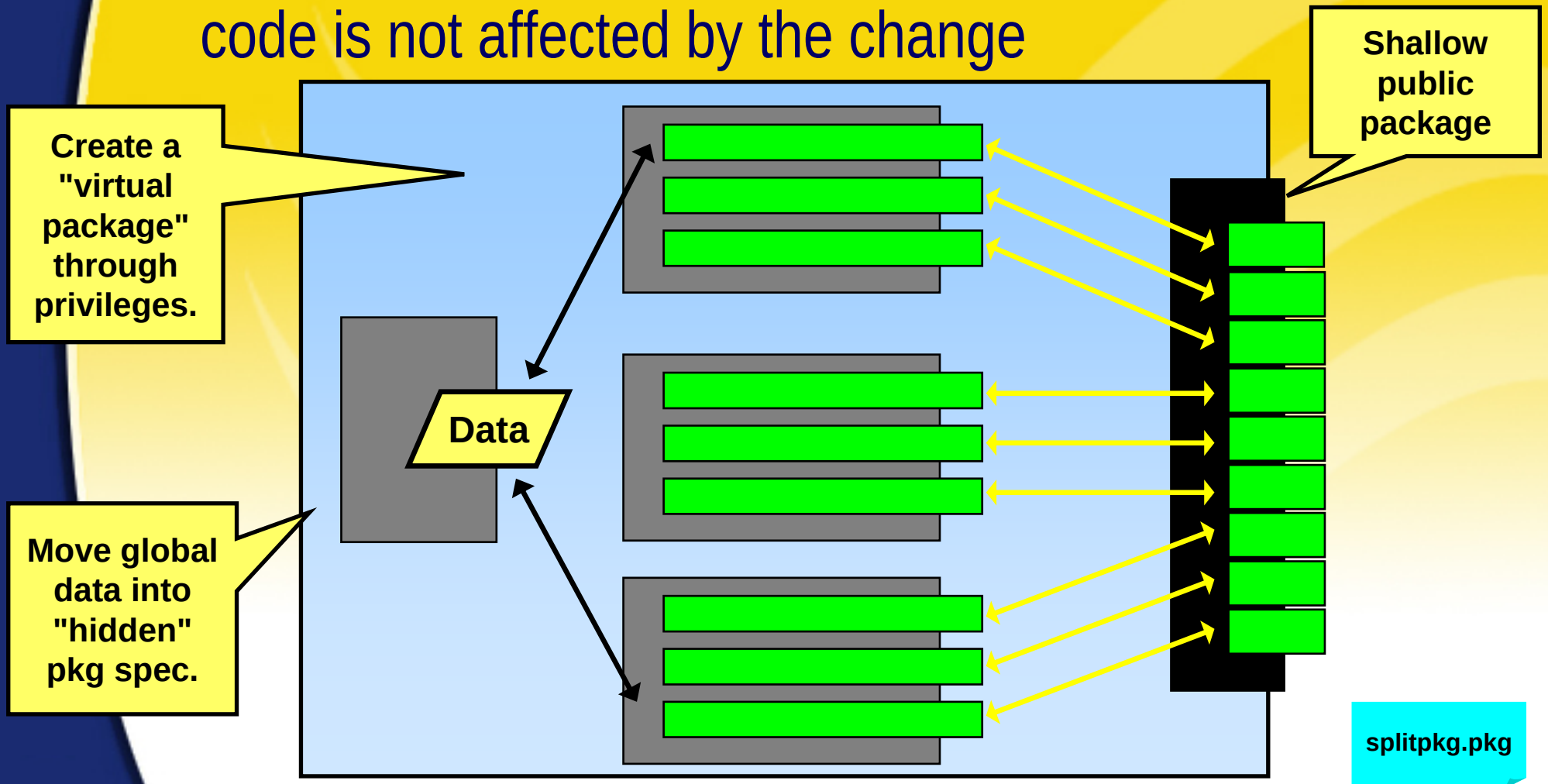
Before...



QUEST SOFTWARE

After: Break Up that Big Body

- Keep the public specification intact so existing code is not affected by the change



QUEST SOFTWARE

Package Construction
Summary

- Hide the Data
  - Build get-and-set routines (or generate them) and keep control of your data
- Make Your Packages Flexible and Accessible
  - Toggles and windows are easy enough to add after the fact
  - Help users do things the right way

PL/SQL Tuning & Best Practices

Modularization & Encapsulating Logic

- Modularizing fundamentals
- Encapsulation scenarios
- The dangers of over-abstraction

Back to Modularizing Fundamentals

- Modularize to reduce complexity, make your tasks manageable, and make your resulting code maintainable
- Tried-and-tested guidelines for 3GLs hold true for PL/SQL
  - Code Complete by Steve McConnell is chock-full of recommendations
- Two to keep in mind:
  - Modularize with top-down design; keep executable sections small and easy to read
  - Avoid multiple RETURN statements in executable section

Keeping Executable Code Tight

- Most of the time when we write programs, we end up with big *blobs* of code spaghetti
- Executable sections that are hundreds, if not thousands of lines long
- If someone -- anyone -- is going to be able to maintain and enhance your code, you need to:
  - Keep your executable sections small and self-documenting
  - Avoid fanatically any code redundancies

An objective worth aiming for:
Thou Shalt Keep Thy Executable Sections Down to a Single Page or Screen

Local Modules Improve Readability

- Move blocks of complex code into the declaration section
- Replace them with descriptive names
- The code is now easier to read and maintain
- You can more easily identify areas for improvement

```
PROCEDURE assign_workload (department_in IN NUMBER)
IS
   CURSOR emp_cur
   IS
      SELECT * FROM emp WHERE deptno = department_in;

   PROCEDURE assign_next_open_case
      (emp_id_in IN NUMBER, case_out OUT NUMBER) IS BEGIN … END;

   FUNCTION next_appointment (case_id_in IN NUMBER) IS BEGIN … END;

   PROCEDURE schedule_case
       (case_in IN NUMBER, date_in IN DATE) IS BEGIN … END;

BEGIN /* main */
   FOR emp_rec IN emp_cur
   LOOP
      IF analysis.caseload (emp_rec.emp_id) <
         analysis.avg_cases (department_in);
      THEN
         assign_next_open_case (emp_rec.emp_id, case#);
         schedule_case
            (case#, next_appointment (case#));
      END IF;
   END LOOP
END assign_workload;
```

locmod.sp
utgen.pkb

Local Modules Avoids Redundancy

- Even if local modules don't have a dramatic impact on code volume, the extra effort to avoid redundancy today is a valuable insurance policy against maintenance costs in the future

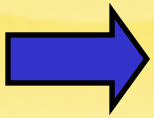
```
PROCEDURE calc_percentages (total_in IN NUMBER)
IS

   FUNCTION formatted_pct (val_in IN NUMBER)
      RETURN VARCHAR2 IS
   BEGIN
      RETURN TO_CHAR ((val_in/total_in) * 100, '$999,999');
   END;

BEGIN
   food_sales_stg := formatted_pct (sales.food_sales);
   service_sales_stg := formatted_pct (sales.service_sales);
   toy_sales_stg := formatted_pct (sales.toy_sales);
END;
```

locmod.sp
hashdemo.sp

Avoid Multiple RETURNS in Functions

- Try to follow the general rule: one way in, one way out
  - Multiple RETURNs in the executable section make it error-prone and difficult to maintain

```
FUNCTION status_desc (
   cd_in IN VARCHAR2)
   RETURN VARCHAR2
IS
BEGIN
   IF cd_in = 'C'
      THEN RETURN 'CLOSED';

   ELSIF cd_in = 'O'
      THEN RETURN 'OPEN';

   ELSIF cd_in = 'I'
      THEN RETURN 'INACTIVE';
   END IF;
END;
```



```
FUNCTION status_desc (
   cd_in IN VARCHAR2)
   RETURN VARCHAR2
IS
   retval VARCHAR2(20);
BEGIN
   IF cd_in = 'C'
      THEN retval := 'CLOSED';
   ELSIF cd_in = 'O'
      THEN retval := 'OPEN';
   ELSIF cd_in = 'I'
      THEN retval := 'INACTIVE';
   END IF;
   RETURN retval;
END;
```

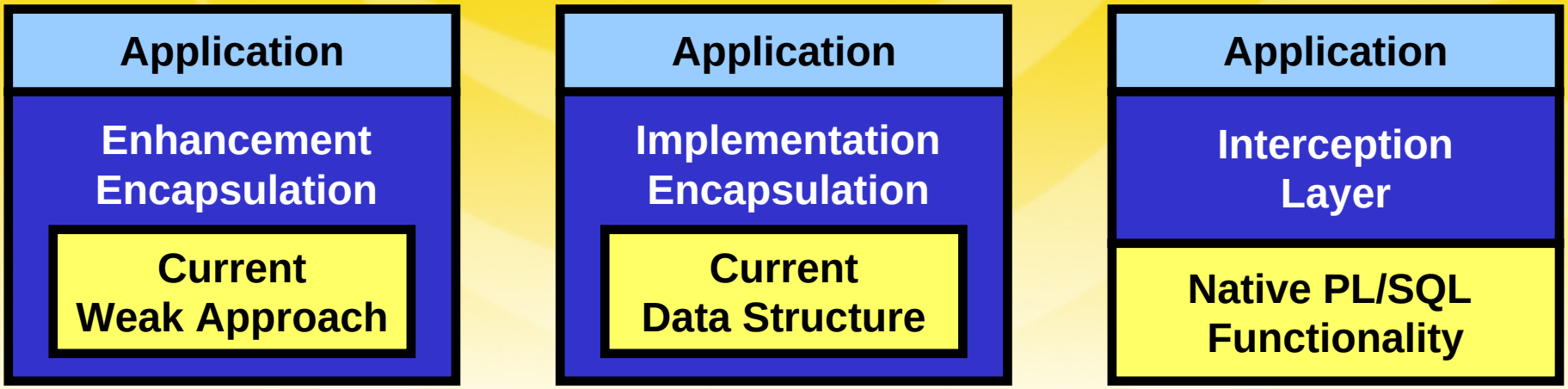
Now it always RETURNs something.

- You *should*, however, include RETURNs in exception handlers unless the error is "catastrophic"

multret.sf
explimpl.pkg

Encapsulating for Protection and Flexibility

- When you encapsulate, you wrap *something* inside a layer of code
- Here are some scenarios crying out for encapsulation:



Developing the Encapsulation Reflex

- We'll explore several examples of encapsulation (a.k.a., abstraction):
  - Implementing transaction flexibility
  - Working around the limitations of DBMS_OUTPUT
  - Storing knowledge so that it won't be forgotten
  - Using assertion routines
- But first...you have to give yourself (or request) the time to be creative, to visualize a higher level of abstraction

Make the Extra Effort!

- The path of "minimal implementation" taken (often) by Oracle:

```
PACKAGE DBMS_UTILITY
IS
   PROCEDURE comma_to_table(
      list   IN  VARCHAR2,
      tablen OUT BINARY_INTEGER,
      tab    OUT uncl_array);
```

All lists manipulated by PL/SQL developers are delimited by commas, right?

- With a user community in the millions, you should aim higher

```
PACKAGE DBMS_UTILITY
IS
   PROCEDURE string_to_table (
      list IN VARCHAR2,
      tab OUT uncl_array,
      delim IN VARCHAR2 := ','
      );

   FUNCTION string_to_table
      (list_in VARCHAR2, delim IN VARCHAR2 := ',')
      RETURN uncl_array;
```

My Dream Implementation...

Allow user definition of the delimiter, overload for more flexible usage.

It's Hard to Achieve Effective Reuse

We need all of the following...

- Quality code that is worth reusing
- Central repository for code
- Means of locating the right piece of code
- Good documentation of code in the repository
- Management commitment to supporting reuse

Transaction Management Flexibility

A simple enough recommendation:
Never call COMMIT; in your PL/SQL code

- When you do that you *hard-code* your transaction boundaries into your code
  - After all, there's no reversing a COMMIT

What? You think that's silly?

Sure, we all need to commit now and then.

The question: what is the best way to do that in your code?
Developers often need more flexibility and features than the COMMIT built-in can provide.

Guys Don't Wanna COMMIT - Reason 1

- During test phase, they want to easily reset their test data back to its original form
  - Sure, you could build a script to drop all the objects and recreate them with the proper data, but that can be time-consuming
- Instead, why not just comment out the COMMIT?
  - When I'm done testing, I will "un-comment" as needed

It's a classic mistake. You finish debugging your application -- and then you *change* it to make it "production ready".

```
PROCEDURE my_monster_application
IS
BEGIN
   Insert_a_bunch_of_rows;

   Change_lots_more_data;

   -- COMMIT;
END;
```

Guys Don't Wanna COMMIT - Reason 2

- They need to update 1M rows and find themselves running out of rollback segments or getting "snapshot too old" errors
  - So you have to commit "every N records"

```
commit_counter := 0
FOR original_rec IN original_cur
LOOP
   translate_data (original_rec);

   IF commit_counter >= 10000
   THEN
      COMMIT;
      commit_counter := 0;
   ELSE
      commit_counter := commit_counter + 1;
   END IF;
END LOOP;
COMMIT;
```

Surely you have better things to do with your time than write code like this!

Guys Don't Wanna COMMIT - Reason 3

- They use a non-default rollback segment, and that darn COMMIT resets the active RB segment to the default
  - The "fix" is easy: you just have to remember to call DBMS_TRANSACTION.USE_ROLLBACK_SEGMENT *every time you execute a COMMIT*
  - And you have to remember the name of that obscure program!

Here's one!

```
BEGIN
   COMMIT;
   DBMS_TRANSACTION.USE_ROLLBACK_SEGMENT ('big_one');
```

Uh oh! There's another one!

```
FOR I IN 1 .. max_years
   do_stuff;
   COMMIT;
   DBMS_TRANSACTION.USE_ROLLBACK_SEGMENT ('bigone');
END LOOP;
```

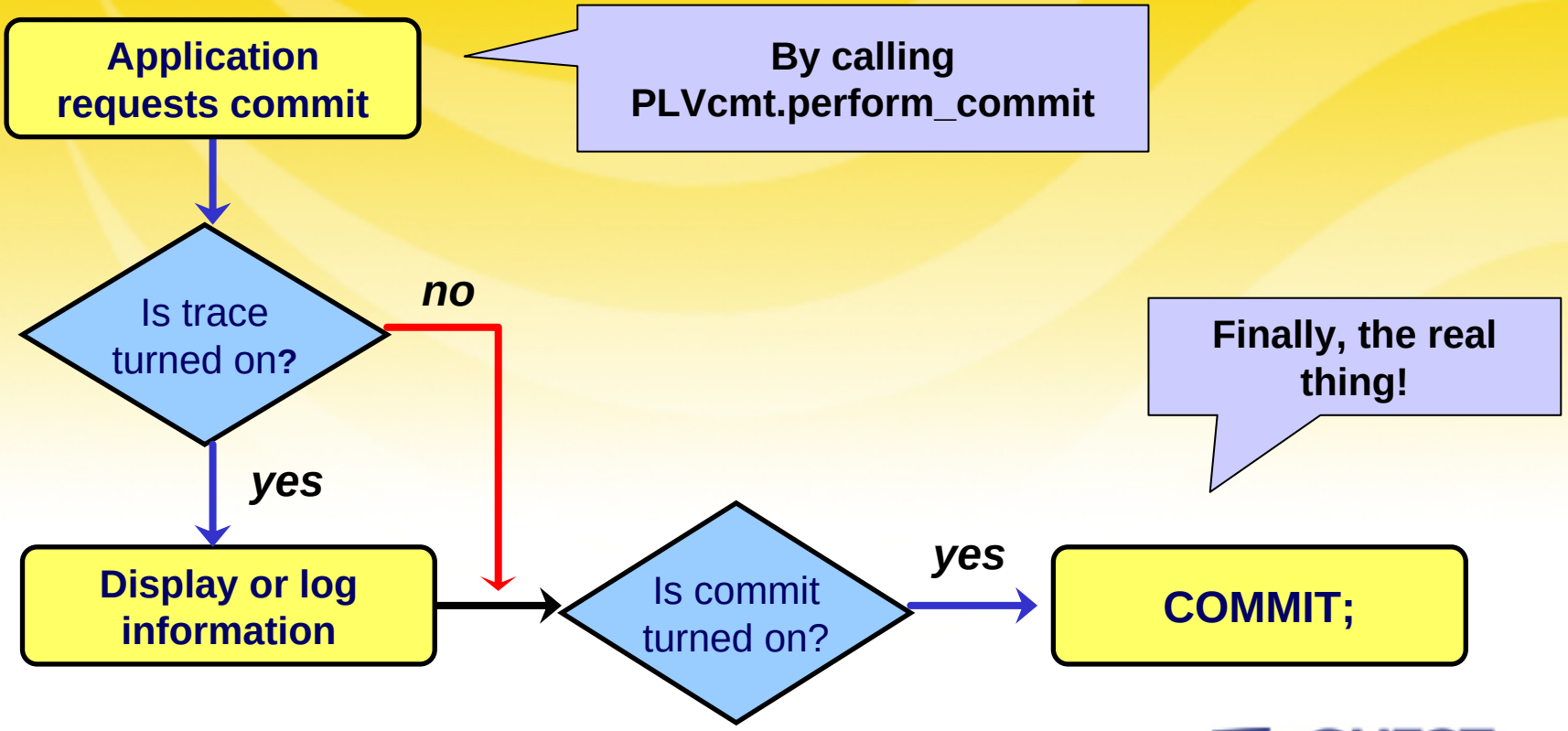
Whoops! Typo!

Who Needs Those Complications?

- You've got enough to worry about, without having to deal with all of that stuff!
- Make your coding life simpler by hiding all those details, all those complications, behind a packaged interface
  - Use toggles and windows in the package to change the commit behavior of your application…without changing your application!

Intercepting and Modifying a Commit

- Use toggles to override and even turn off default processing



QUEST SOFTWARE

Using the COMMIT Alternative

- The following program uses the PL/Vision alternative to COMMIT

```
CREATE OR REPLACE PROCEDURE myapp (counter IN INTEGER)
IS
BEGIN
   /* Define the current rollback segment. */
   PLVcmt.set_rbseg ('bigseg');

   FOR cmtind IN 1 .. counter
   LOOP
      DELETE FROM emp2 WHERE ROWNUM < 2;

      PLVcmt.perform_commit (
        'DELETED ' || SQL%ROWCOUNT ||
        ' on iteration ' || cmtind);
   END LOOP;
END;
```

Set the non-default rollback segment.

While committing, also pass trace information.

cmt.tst
plvcmt.sps
plvcmt.spb

Simplify Incremental Commit Logic

- The following program commits every N record
  - But the N is not fixed in the code
  - Much less code needs to be written

```
PROCEDURE update_seven_million_rows (every_n IN INTEGER)
IS
BEGIN
   PLVcmt.commit_after (every_n);

   PLVcmt.init_counter;

   FOR original_rec IN original_cur
   LOOP
      translate_data (original_rec);
      PLVcmt.increment_and_commit;
   END LOOP;

   PLVcmt.perform_commit;
END;
```

Set the increment and initialize the counter.

Set the increment and initialize the counter.

Save any "leftovers".

The Added Value Adds Up

- Some people right now are surely thinking to themselves:

This instructor must be *nuts*. You actually expect us to build a package that contains 18 programs, instead of just COMMIT?"

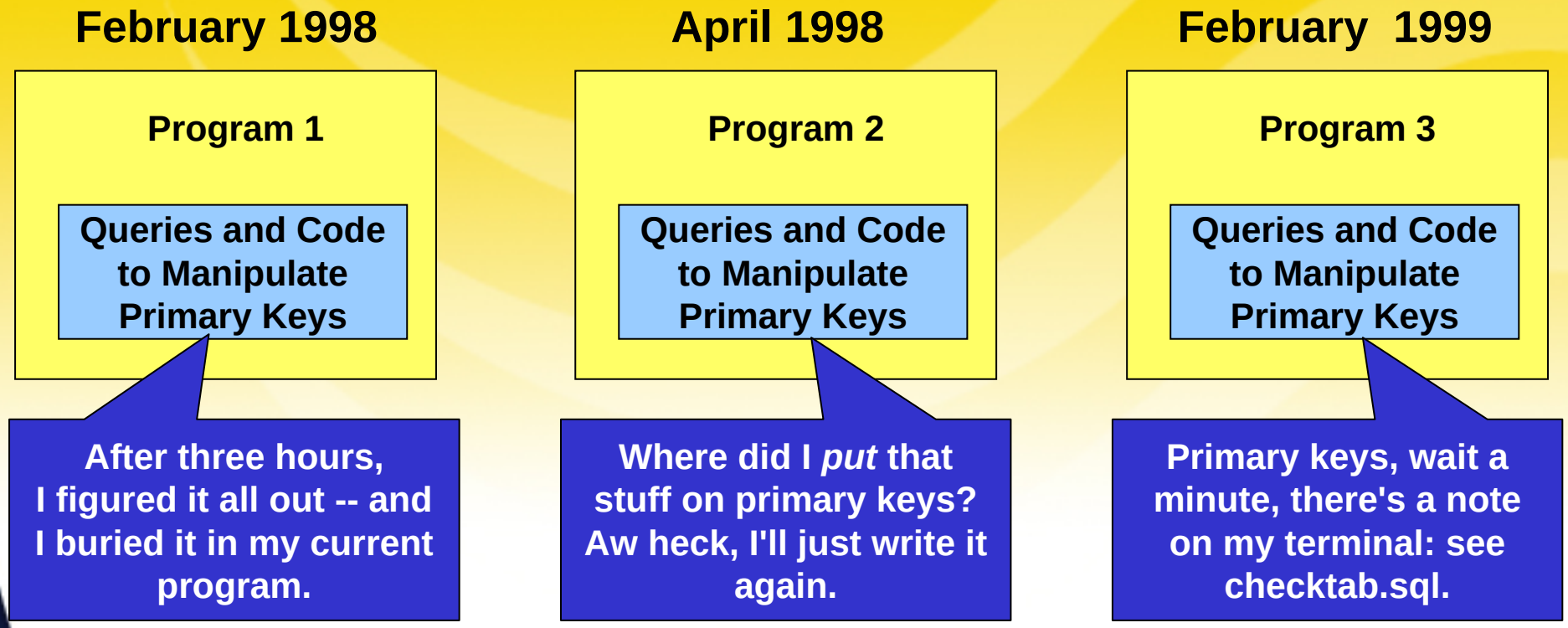
Well, it *would* be nuts to build something that already exists. So use PLVcmt (Lite or otherwise) if this technique looks handy.

Otherwise, the next time you find yourself writing incremental commit logic or wishing you could see when/if you committed, remember PLVcmt!

QUEST SOFTWARE

Storing Acquired Knowledge

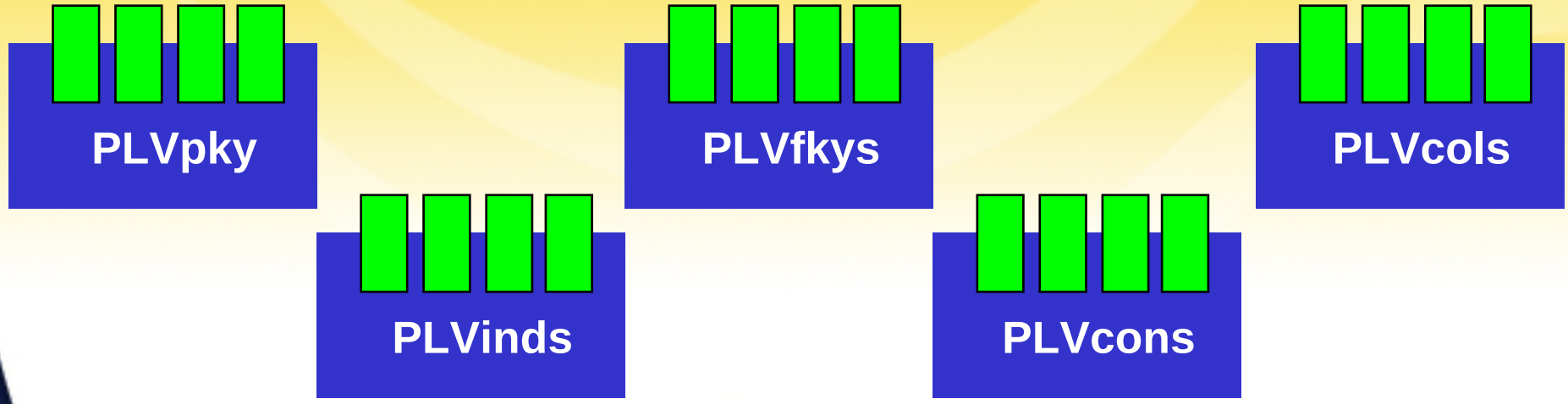
- We learn new techniques after significant investment of research time, and then we *lose* access to that new knowledge
  - The result is lowered productivity and redundant code



- Give yourself a break and encapsulate as you learn

Encapsulating Data Dictionary Info

- How many times have you ventured deep into the reference manuals to figure out how to extract information from the data dictionary on a given topic?
  - It's a jungle in there!
  - A package is a perfect place to store the knowledge and make it easily accessible

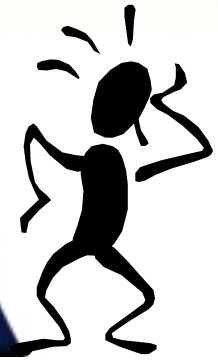


pky.tst
tabhaslong.sf

QUEST SOFTWARE

Let's Talk About DBMS_OUTPUT

- PL/SQL 1 had ZERO tracing/output capability, causing an uproar from the developer community
- So Oracle Corporation gave us DBMS_OUTPUT, and it:
  - Requires the typing of 20 letters just to call the darned thing
  - Knows nothing about the Boolean datatype
  - Ignores my requests to display blank lines and trims leading blanks
  - Raises VALUE_ERROR if the string has more than 255 bytes
  - Refuses to display anything until the block finished executing?

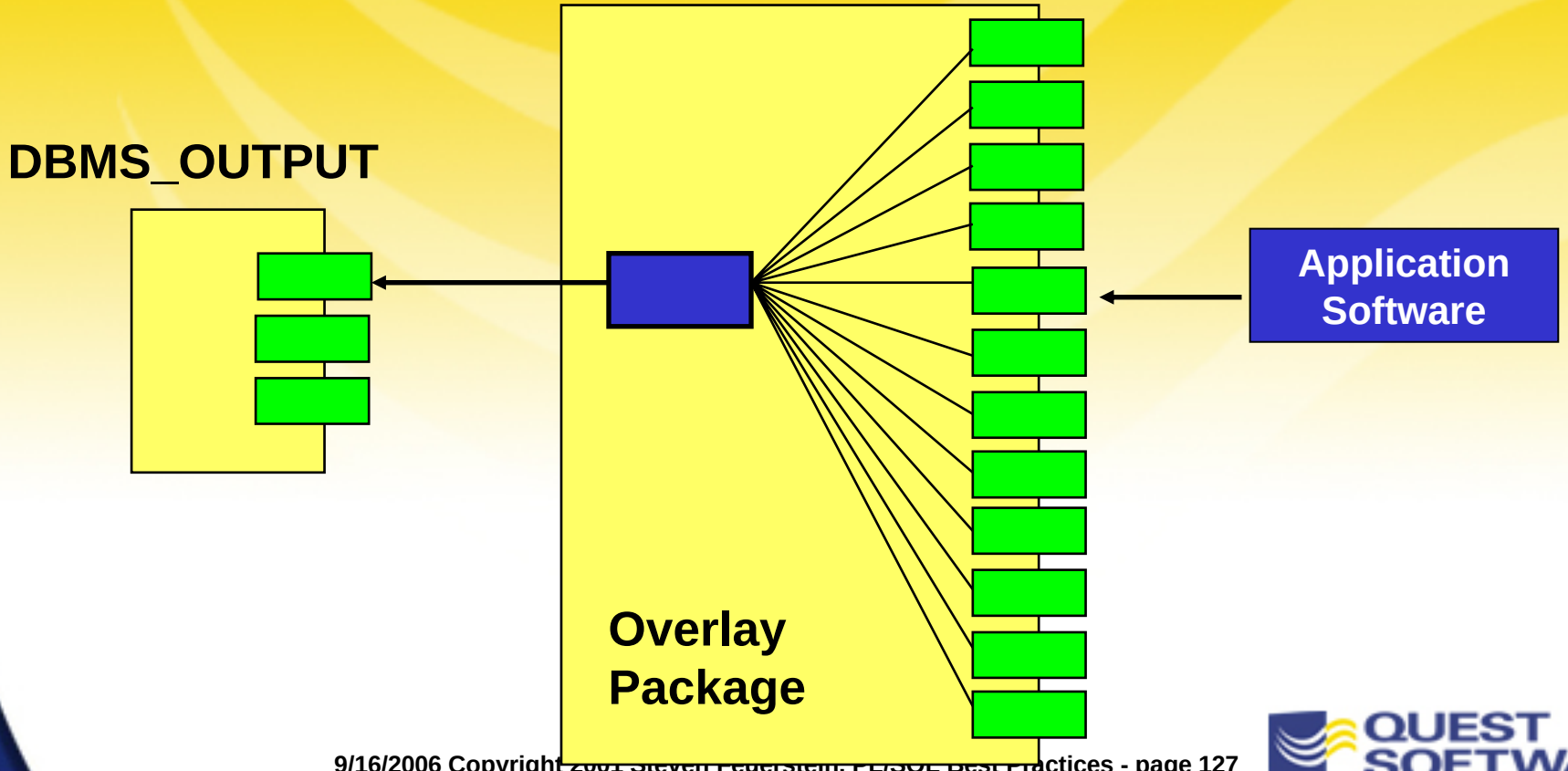


How did they manage to make something this straightforward so painful to use?

QUEST SOFTWARE

Saving Yourself From DBMS_OUTPUT

- Just say to yourself: "I will never call *that thing* in my application, instead I will call my own, better alternative"



QUEST SOFTWARE

Constructing a Useful Substitute

```
PROCEDURE p.l (line_in IN VARCHAR2) IS
BEGIN
   IF RTRIM (line_in) IS NULL
   THEN
      DBMS_OUTPUT.PUT_LINE (g_prefix);

   ELSIF LENGTH (line_in) > p.linelen
   THEN
      PLVprs.display_wrap (line_in, linelen);

   ELSE
      DBMS_OUTPUT.PUT_LINE (g_prefix || line_in);
   END IF;

EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.ENABLE (1000000);
      DBMS_OUTPUT.PUT_LINE (g_prefix || line_in);
END;
```

Don't ignore leading blanks or blank lines.

Wrap (instead of blowing up on) long strings.

Anticipate and attempt to correct common problems.

Validation of Assumptions

- Every module has its assumptions
  - A certain set of key data is available, or dates must be within a specific range
  - If you don't validate those assumptions, your programs will break in unpredictable, ugly ways
- Code defensively
  - Assume that developers will *not* use your programs properly

Traditional Approach to Validation

- Use conditional logic to perform the checks

```
PROCEDURE calc_totals (
    dept_in IN INTEGER, date_in IN DATE) IS
BEGIN
   IF dept_in IS NULL
   THEN
      p.l ('Provide a non-NULL department ID.');
      RAISE VALUE_ERROR;
   END IF;

   IF TRUNC (date_in) NOT BETWEEN
      TRUNC (ADD_MONTHS (SYSDATE, -60)) AND TRUNC (SYSDATE)
   THEN
      p.l ('Date is out of range.');
      RAISE VALUE_ERROR;
   END IF;

   /* Ok, now perform the calculation. */
```

If department pointer is NULL, display message & raise error.

If date is out of range, display message & raise error.

The Problems with Business-as-Usual

- First, notice the repetition in this approach
  - Two IF statements (issues of excessive code volume)
  - Two different calls to a display mechanism
  - Two different RAISEs
- Second, I have exposed or hard-coded the way I check for and deal with assumption violations
  - If I ever decide to change the way I handle violations of assumptions, I have to go to each IF statement and make the fix
- You can avoid all of these problems and end up with a much more powerful and flexible way of validating assumptions by using assertion routines

Taking the Assertion Approach

```
PROCEDURE calc_totals
   (dept_in IN INTEGER, date_in IN DATE) IS
BEGIN
   assert (dept_in IS NOT NULL,
      'Provide a non-NULL department ID.');

   assert (TRUNC (date_in) BETWEEN
      TRUNC (ADD_MONTHS (SYSDATE, -60)) AND TRUNC (SYSDATE),
      'Date is out of range.');

   /* Ok, now perform the calculation. */
   . . .
END;
```

Ease of use improves chances of checking all conditions.

- Call a program instead of writing the code repeatedly
  - A generic condition tester, consolidating all logic in one place
  - Work in positive terms, asserting that your assumption holds
  - Segregate validation logic from "approved" executable logic

Building a Generic Assertion Program

Let's start with simplest, most naive implementation —

```
PROCEDURE assert (condition_in IN BOOLEAN)
IS
BEGIN
   IF NOT condition_in
   THEN
      RAISE VALUE_ERROR;
   END IF;
END;
```

Stops the calling program if the condition is FALSE.

- Puts a thin layer of code between the application and the asserting logic
  - This gives you flexibility...
  - And the ability to recover from oversights and near misses

I need to display a message!

I need to raise a different exception!

And what about Nulls?

Enhancing the Assertion Program

```
PROCEDURE assert (
   condition_in IN BOOLEAN,
   msg_in IN VARCHAR2 := NULL)
IS
BEGIN
   IF NOT condition_in OR condition_in IS NULL
   THEN
      IF msg_in IS NOT NULL
      THEN
         DBMS_OUTPUT.PUT_LINE (msg_in);
      END IF;
      RAISE errpkg.assertion_failure;
   END IF;
END;
```

Ah, what a relief! Now I know for certain that all NULLs are trapped.

Display a message if it is provided.

Special exception just for assertions.

- Make any needed corrections in one place only in your entire application
  - Correct mistakes with a minimum of embarrassment!

Offer Specialized Assertions

- You will find yourself writing the same Boolean expressions repeatedly
  - Bury the repetition in a variety of assertion programs
- The result is that you write less code, get more consistent behavior, and are more likely to actually test for these conditions

```
PACKAGE PLV
IS
   PROCEDURE assert
     (bool_in IN BOOLEAN,
      stg_in IN VARCHAR2 := NULL);

   PROCEDURE assert_notnull
     (val_in IN BOOLEAN|DATE|NUMBER|VARCHAR2,
      stg_in IN VARCHAR2 := NULL);

   PROCEDURE assert_inrange
      (val_in IN DATE,
       start_in IN DATE := SYSDATE,
       end_in IN DATE := SYSDATE+1,
       stg_in IN VARCHAR2 := NULL,
       truncate_in IN BOOLEAN := TRUE);

   PROCEDURE assert_inrange
     (val_in IN NUMBER,
      start_in IN NUMBER, end_in IN NUMBER,
      stg_in IN VARCHAR2 := NULL);

   . . .
```

plv.sps

Using Specialized Assertions

- Let's take a final look at the calc_totals procedure, this time using assertion programs designed to check for specific *types* of conditions

```
PROCEDURE calc_totals
   (dept_in IN INTEGER, date_in IN DATE)
IS
BEGIN
   PLV.assert_notnull (dept_in,
      'Provide a non-NULL department ID.');

   PLV.assert_inrange (date_in,
      ADD_MONTHS (SYSDATE, -60), SYSDATE,
      'Date is out of range.');

   /* Ok, now perform the calculation. */
   . . .
END;
```

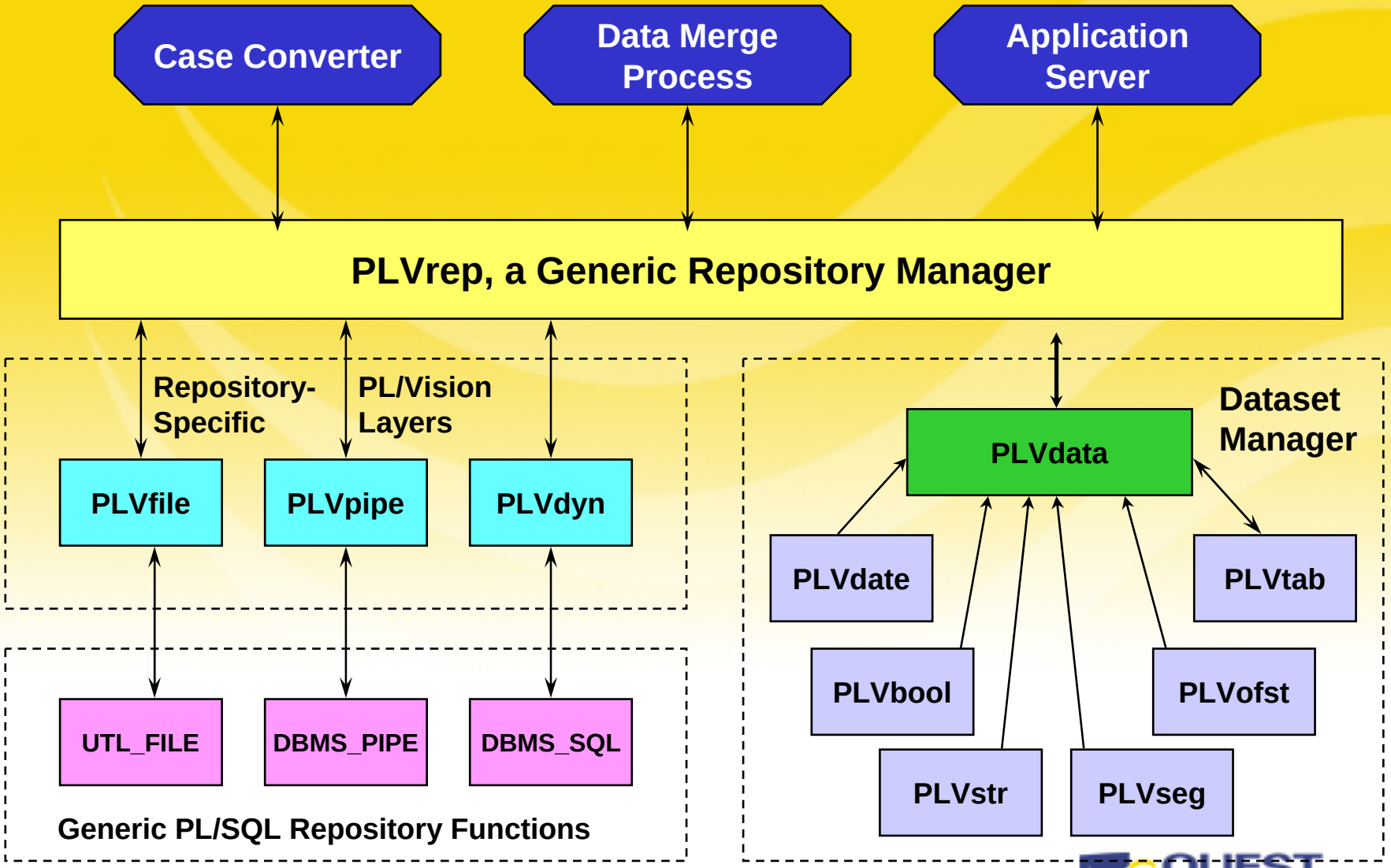
The Dangers of Excessive Encapsulation

- Suppose that I am building an application that needs to make extensive use of UTL_FILE to write the contents of a variety of tables to files
- Two possible implementations:
  - Create a single, generic program that can write the contents of any table to any file…HARD!
  - Write a new "dump" procedure for each table…TEDIOUS!
- I decided to tackle the hard challenge, but do so in a way that would offer a powerful "repository management" utility to any use of PL/Vision: PLVrep

Design Objective of PLVrep

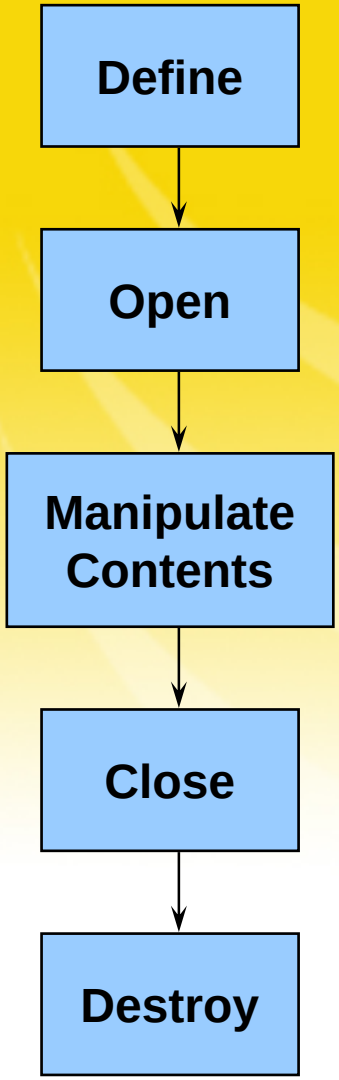
- Single language or API that can be used to manipulate the various kinds of data repositories known to PL/SQL
  - Database table
  - Collection (index-by table)
  - Database pipe
  - Operating system file
  - String
  - Screen
- With PLVrep, you (theoretically) don't need to know about UTL_FILE, SQL, DBMS_SQL, DBMS_PIPE, DBMS_OUTPUT, etc. *Excellent!*

Layers of Abstraction with PLVrep



QUEST SOFTWARE

Flow of Generic Repository Mgt.



Allocates memory to keep track of information about the repository.

Opens a dynamic cursor, opens a file, etc. Makes repository available for read/write operations.

Read from repository.
Write to repository.
Analyze status of repository.

Closes a file, closes a cursor, etc. Makes repository unavailable for read/write operations

Deallocates memory from the repository. Pointer becomes invalid.

QUEST SOFTWARE

PLVrep Tackles "Table to File"

```
CREATE OR REPLACE PROCEDURE tab2file (
   tab IN VARCHAR2, file IN VARCHAR2 := NULL,
    delim IN VARCHAR2 := ',')
IS
   tid PLS_INTEGER := PLVrep.dbtabid (tab);
   fid PLS_INTEGER :=
      PLVrep.fileid (NVL (file, tab || '.dat'),
         fixedlen=>PLV.ifelse (delim IS NULL, TRUE, FALSE));
BEGIN
   PLVrep.tabsegs (tid, tab); -- Make the repositories look like
   PLVrep.tabsegs (fid, tab); -- the table.
   PLVrep.copy (tid, fid, segdelim => delim);
END;
```

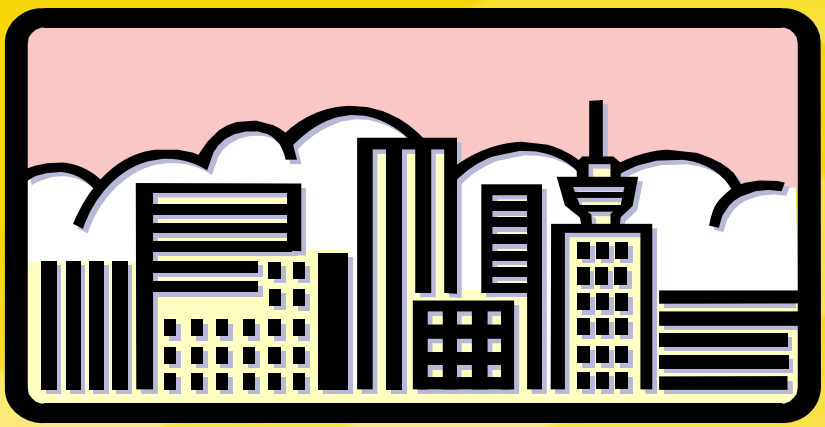
Write *any* table to *any* file.
Five lines of executable code.
Wow!

Problems with PLVrep

- Too complicated
  - 143 procedures and functions
  - A robust new *vocabulary* for PL/SQL developers ?!?!
- Too generic
  - It tries to achieve (and largely accomplishes) *too much*.
  - Feature set overwhelms potential advantage of a “single language”
- Too slow
  - The tradeoff for being so generic, for hiding so much complexity, is that PLVrep has to do *lots* of work on your behalf

What is High Quality Software?

- Is it a "perfectly" designed and implemented system, built upon many layers of abstracted, theoretically reusable code?
  - Is such a thing (a) truly desirable or (b) even possible?



or…

- Should I try to build a beautiful monument to myself or should I create a living, breathing *home* that others can inhabit -- and modify -- as needed?

Let's Talk About Habitability

"Habitability is the characteristic of a piece of source code that enables programmers, coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions, and to change it comfortably and confidently."

… Richard Gabriel, Patterns of Software

- Why would habitability be important?
  - Software is written by people; what makes us human must be taken into account in the software development process
  - Software must constantly be changed. It can never possibly be perfect, because it is supposed to reflect the real world
- A closely-related concept: *piecemeal growth*

Piecemeal Growth of Software

"Piecemeal growth is the process of design and implementation in which software is embellished, modified, reduced, enlarged and improved through a process of repair rather than of replacement."

… Richard Gabriel, Patterns of Software

- Gee, it's like someone gave a name to what we are already and always doing
- Let's stop beating ourselves up with a bunch of shoulds: should have encapsulated, should have reused, should have abstracted…
  - Instead, we should accept it as reality and change our environment and our processes to support it

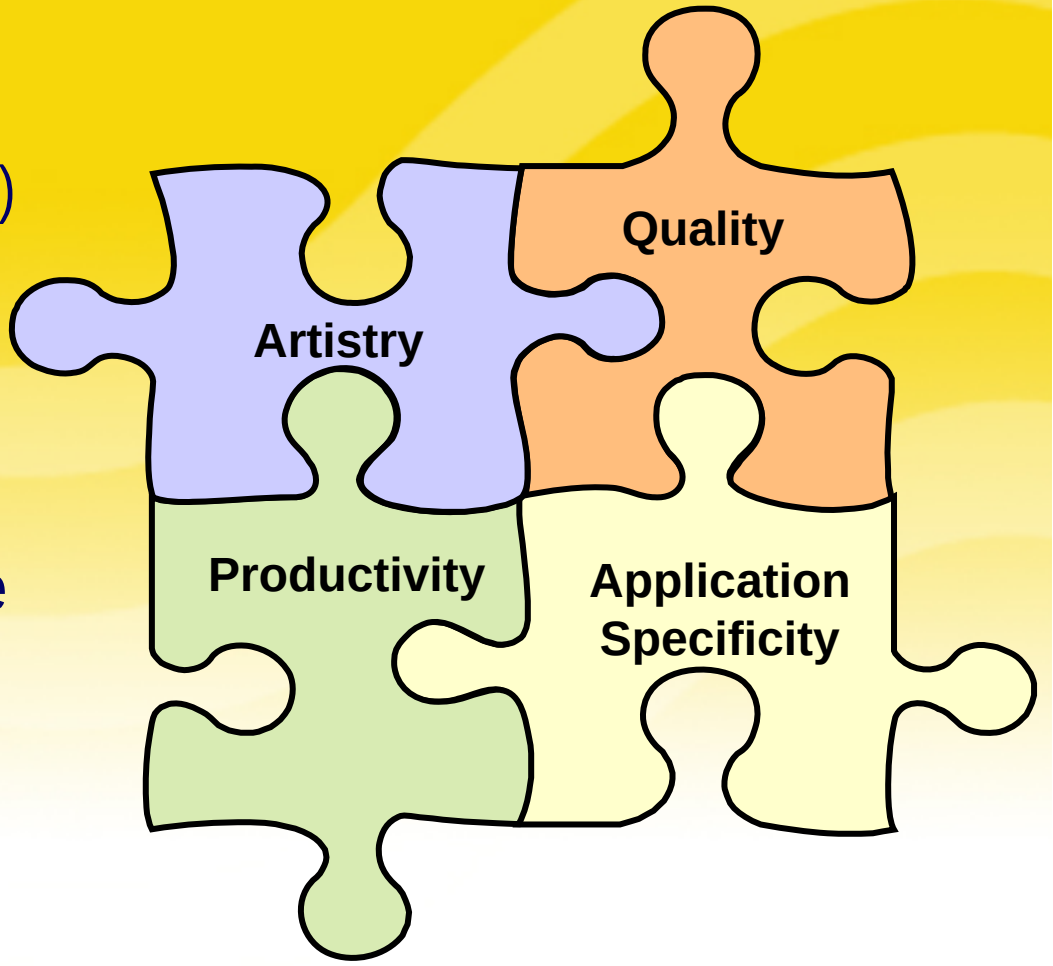
Habitable “Table to File” Program

```
CREATE OR REPLACE PROCEDURE DEPARTMENT2file (
   loc IN VARCHAR2,
   file IN VARCHAR2 := 'DEPARTMENT.dat',
   delim IN VARCHAR2 := '|'
   )
IS
   fid UTL_FILE.FILE_TYPE;
   line VARCHAR2(32767);
BEGIN
   fid := UTL_FILE.FOPEN (loc, file, 'W');

   FOR rec IN (SELECT * FROM DEPARTMENT)
   LOOP
      /* Construct the single line of text.*/
      line :=
         TO_CHAR (rec.DEPARTMENT_ID) || delim ||
         rec.NAME || delim ||
         TO_CHAR (rec.LOC_ID);

      UTL_FILE.PUT_LINE (fid, line);
   END LOOP;

   UTL_FILE.FCLOSE (fid);
END;
```

Easy to understand, but I have to write a new procedure for each table.

Resolving the Dilemma w/Code Generation

- So we have a bind:
  - Write (and learn how to use) highly abstracted, complex code, or…
  - Write (and maintain) lots of code, and make it decent quality, maintainable stuff
- Perhaps there is a "third way": context-aware code generation



Generate "Habitable" Code?

```
[STOREIN][objname]2file.sp
CREATE OR REPLACE PROCEDURE [objname]2file (
   loc IN VARCHAR2,
   file IN VARCHAR2 := '[objname].dat',
   delim IN VARCHAR2 := '|'
   )
IS
   fid UTL_FILE.FILE_TYPE;
[IF]oraversion[IN]8.0,8.1
   line VARCHAR2(32767);
[ELSE]
   line VARCHAR2(1023);
[ENDIF]
BEGIN
   fid :=
     UTL_FILE.FOPEN (loc, file, 'W');

   FOR rec IN (SELECT * FROM [objname])
   LOOP
      line :=
   [FOREACH]col
      [IF][coldatatype][eq]VARCHAR2
...
```

Template captures pattern.

Utility generates code from template for a specific object.

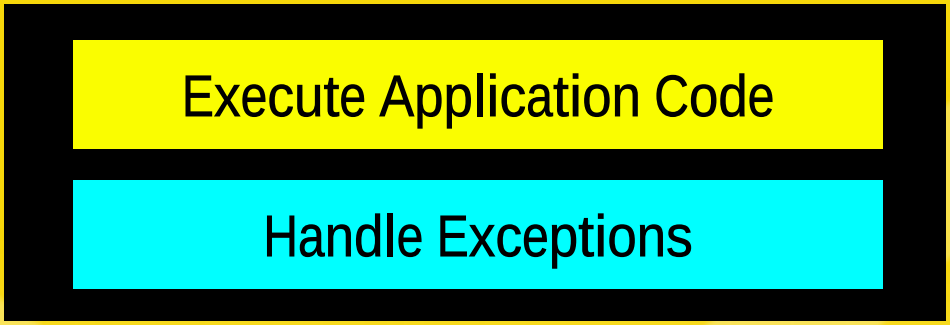
Resulting code is readily understandable and easily maintained.

Encapsulation Recommendations

- Build small encapsulations/abstractions
  - If I need to take a week-long class to learn how to use the abstraction/library, I will never really (re)use that code
- Keep hierarchies shallow as long as possible
  - If you make a mistake along the way, you can fix it with a minimum of impact
- Build incrementally
  - Avoid “master plans”; build in direct response to programmer needs
- Don't accept performance trade-offs for abstraction
  - It is very rarely justified in a production environment
- Find or build tools/utilities to leverage abstractions
  - The toughest challenge of all!

Developing an Exception Handling Architecture

Exception Handling in PL/SQL



- 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

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;
```

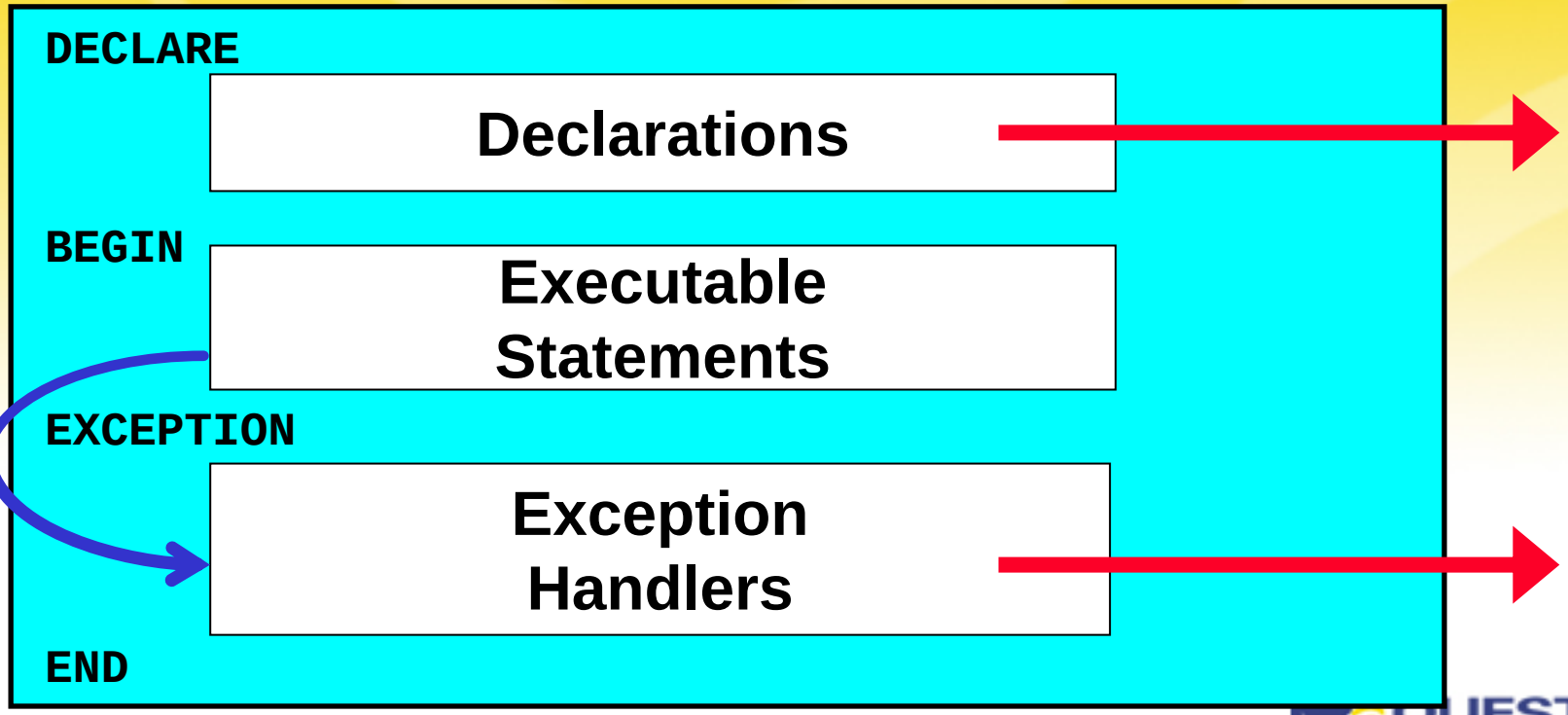
```
PROCEDURE calc_expenses
IS BEGIN
   ...
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      log_error;
END;
```

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;
```

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

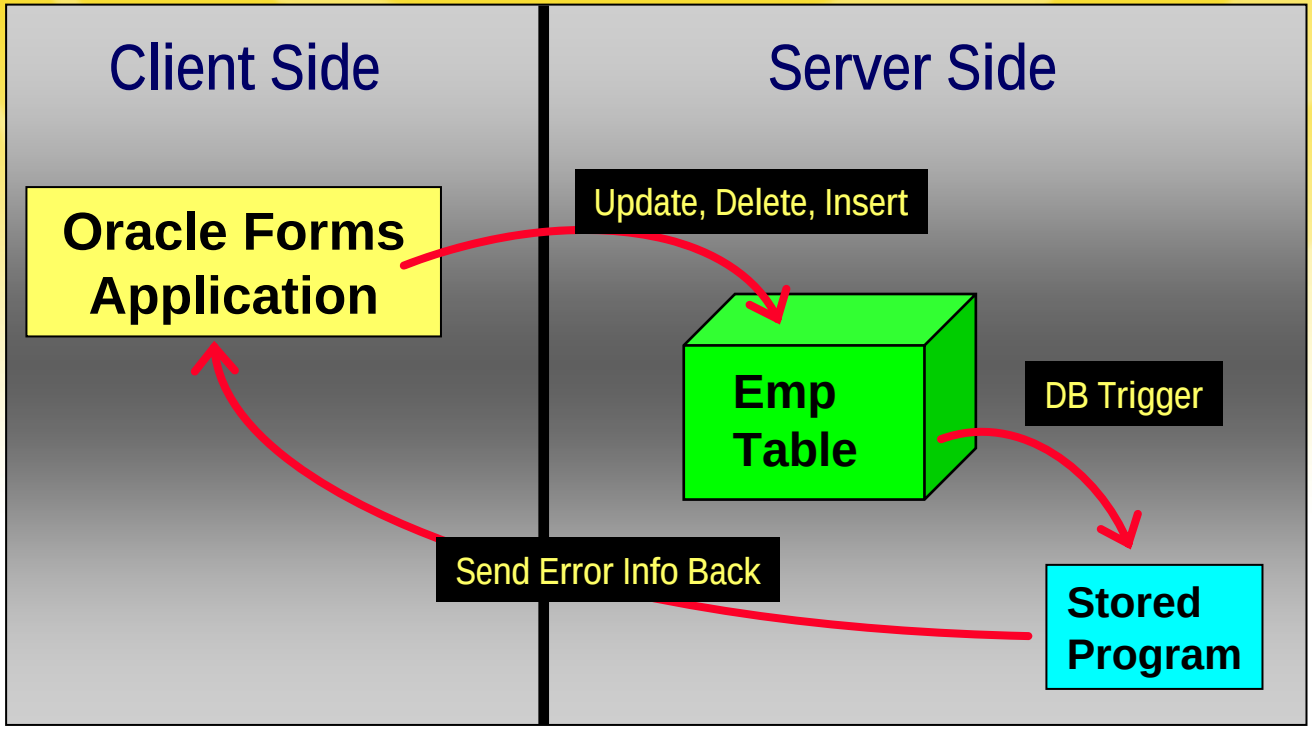
```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      ROLLBACK TO last_log_entry;
      INSERT INTO log VALUES (...);
      SAVEPOINT last_log_entry;
END;
```

But it can get complicated!

lostlog*.sql
rb.pkg

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);
```

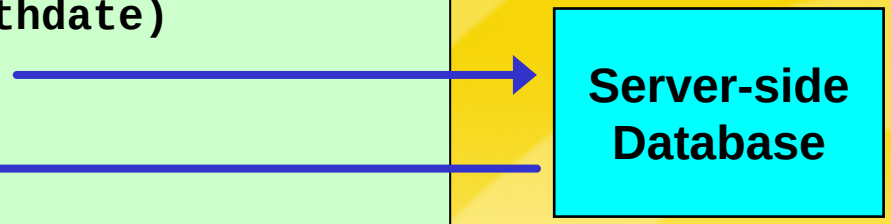
The following code from a database triggers shows a typical usage of RAISE_APPLICATION_ERROR

```
IF :NEW.birthdate > ADD_MONTHS (SYSDATE, -1 * 18 * 12)
THEN
   RAISE_APPLICATION_ERROR
      (-20070, 'Employee must be 18.');
END IF;
```

Handling App. Specific Exceptions

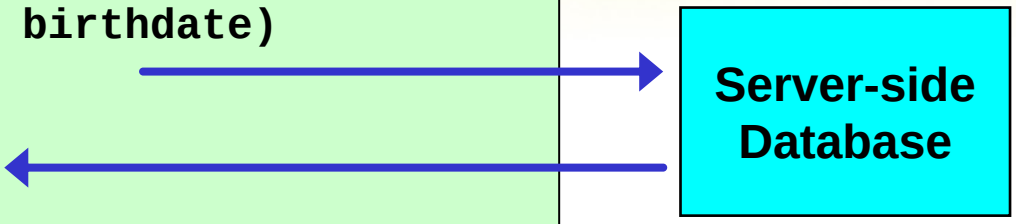
– Handle in OTHERS with check against SQLCODE...

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



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;
```



QUEST SOFTWARE

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.

Database trigger raises exception.

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

```
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
excquiz6.sql
excquiz6a.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

Where Did the Error Occur?

- When an error occurs inside your code, the most critical piece of information is the line number of the program in which the error was raised
- How can you obtain this information?

A. DBMS_UTILITY.FORMAT_ERROR_STACK

B. Unhandled exception

C. DBMS_UTILITY.FORMAT_CALL_STACK

D. Write error information to log file or table

disperr*.tst

What, No GOTO?

- It gets the job done…but does the end justify the means?

```
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 and then execute the following command. What is displayed on the screen?

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

valerr.pkg
valerr2.pkg

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

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. Here are my recommendations:

DISCOURAGE individual developer usage of RAISE_APPLICATION_ERROR, PRAGMA EXCEPTION_INIT, explicit (hard-coded) -20,NNN error numbers, hard-coded error messages, exposed exception handling logic.

ENCOURAGE use of standardized components, including programs to raise application-specific exception, handle (log, re-raise, etc.) errors, and rely on pre-defined error numbers and messages.

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.pkg

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.pkg

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.pkg

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!

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.

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

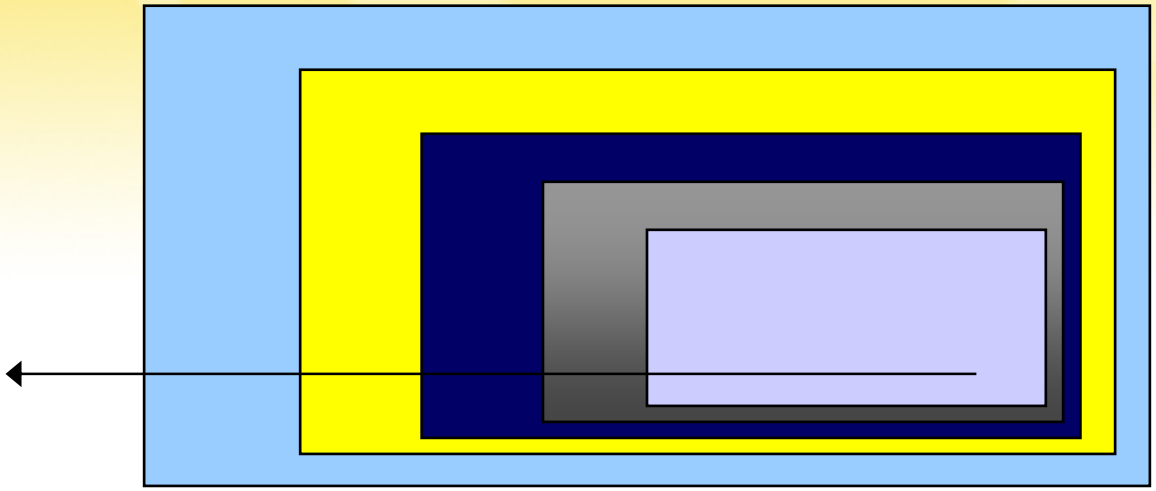
```
FUNCTION errtext (err_in IN INTEGER := SQLCODE) RETURN VARCHAR2 IS
   CURSOR txt_cur IS
      SELECT text FROM message_text
       WHERE texttype = 'EXCEPTION' AND code = err_in;
   txt_rec txt_cur%ROWTYPE;
BEGIN
   OPEN txt_cur;
   FETCH txt_cur INTO txt_rec;
   IF txt_cur%NOTFOUND THEN
      txt_rec.text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

You can selectively override default, cryptic Oracle messages.

- Or, as shown in the errpkg.pkg 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 Summary

- 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

PL/SQL Tuning & Best Practices

Control Contention in Multi-User Systems

Sharing Resources

- Anyone writing single-user applications can skip this topic.
  - Any takers?
- Contention for shared resources can skewer performance of the most perfectly written pl/sql programs.
- Many contention issues have their origin in application design and implementation.

Types of Contention

- Lock contention: waiting to access shared data
- Latch contention: waiting to access Oracle internal resources
  - Can be difficult to diagnose properly, but there are classic cases to avoid
- I/O contention: waiting to read/write to disk
  - Usually a DBA datafile / tablespace distribution issue
- When user processes WAIT, algorithm efficiency doesn't matter at all!

SQL Locking

```
LOCK TABLE emp in EXCLUSIVE
MODE;

SELECT * FROM emp FOR UPDATE;

UPDATE emp SET sal = sal*1.1
 WHERE dept = 10;

SELECT * FROM dept;
```

Only one user can modify any row in table

Row-share lock for all rows selected

Row-exclusive lock for all rows updated

No data locks taken

- Locks released at transaction commit (rollback)
- Lock only what you need and release as soon as possible to avoid contention

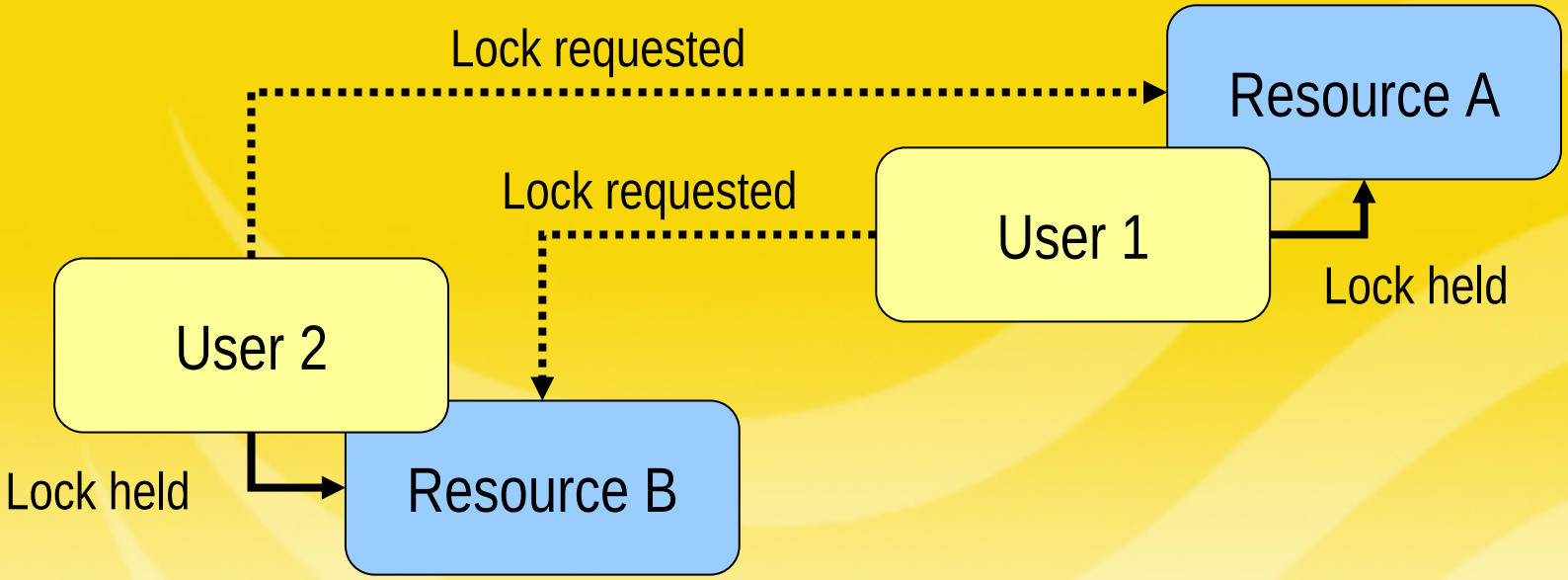
Controlled lock waiting

```
resource_busy  EXCEPTION;
PRAGMA EXCEPTION_INIT(resource_busy,-54);
tries INTEGER := 1;
LOOP
EXIT WHEN locked OR tries = 3;
   BEGIN
      SELECT * FROM emp WHERE dept = 10
      FOR UPDATE NOWAIT;
      locked := TRUE;
   EXCEPTION
      WHEN resource_busy
      THEN tries := tries + 1; DBMS_LOCK.SLEEP(.5);
   END;
END LOOP;
```

Oracle also exposes lock mgmt services to PL/SQL

- NOWAIT raises ORA-54 when row(s) cannot be locked
- Instead of waiting forever, catch exception and try again ½ second later (up to 3 times)

Deadlock (name says it)



- Each session needs A and B to complete transaction
  - And each holds what the other needs
- One session's transaction will be rolled back, which?
- Avoid deadlock by locking resources in consistent order
  - Lock A first, then B and deadlock will not occur
  - Another good reason to encapsulate transactions

Latching

- Latches are like very short duration locks
- Latches protect Oracle internal (SGA) data structures
  - Your application cannot ask for a latch explicitly
- All SQL activity involves numerous latches
  - Latching is not a problem in itself
  - Contention arises when too many processes ask for latches too fast that are being held too long
- Latch contention manifests as excess CPU consumption and session waiting

SQL processing latches

library cache: lookup cached SQL object
cache buffers chains: lookup data blocks in buffer cache
cache buffers lru chain: add new blocks to buffer cache LRU lists

You can't prevent latching but you can avoid many latching problems.

library cache: lookup cached SQL
shared pool: allocate space for new SQL

```
SELECT * FROM EMP;

UPDATE EMP SET name = 'JB'
WHERE id = '123';

COMMIT;
```

redo allocation: get space in redo buffer
redo copy: copy changes to buffer

Controllable latch contention

- Library cache and shared pool latches
  - Too much hard parsing => not enough SQL sharing
  - Use bind variables, not literal concatenation
  - Avoid object invalidation/recompilation
- Redo allocation and redo copy latches
  - May be caused by very high commit frequency
  - Another good reason to use a commit encapsulation package

ORA-1555: Snapshot too old

- Common cause:
  - Batch update driven by cursor on large result set
  - Other processes updating cursor table(s)
  - Concurrency problem, not rollback segment sizing
- False cure: commit more frequently
  - Problem is NOT rollback for updates, but rollback for read-consistent image of cursor (unaffected by commit)
- True cure: close and re-open the driving cursor
  - Process the large result set in a sequence of read-consistent images

ORA-1555 Candidate

```
CURSOR big_result_cur IS
SELECT * FROM big_table;

FOR result_rec IN big_result_cur
LOOP
   /* do some DML based on result_rec */
END LOOP;
```

- Oracle tries to keep read-consistent image of big_result_cur (as/of loop start time) until loop finishes
- If big_table rows modified by other processes this can become difficult (or impossible = ORA-1555)

Avoiding ORA-1555

```
last_id_processed INTEGER := 0;
Batch_size INTEGER := 1000;

CURSOR result_set_cur(Lid integer) IS
   SELECT * FROM big_table BT
    WHERE BT.id > Lid
    ORDER BY BT.id ASC;

result_set_rec result_set_cur%ROWTYPE;

OPEN result_set_cur(last_id_processed);
LOOP
   FETCH result_set_cur INTO result_set_rec;
   EXIT WHEN result_set_cur%NOTFOUND;

   /* process result_set_rec */
   last_id_processed := result_set_rec.id;

   IF MOD(result_set_cur%ROWCOUNT,batch_size=0)
   THEN CLOSE result_set_cur; COMMIT;
        OPEN result_set_cur(last_id_updated);
   END IF;
END LOOP; COMMIT;
```

Parameterized cursor pushes through successive result sets

Assumes results can be ordered by id and all ids > 0

Release read-consistent image of result_set_cur and open new one, this avoids the ORA-1555

Dangers of Dynamic SQL

- Common tradeoff: flexibility vs. performance
- Runtime compilation is always slower than pre-compilation
  - Extra parsing/optimization (CPU) and memory allocation
- May only be problematic when volume is large
  - So don't be dogmatic about it!
- Dynamic DDL can really be a killer
  - Library cache object invalidation/recompilation

Synonyms add overhead

- Public and private synonyms both add additional overhead vs. *schema.objname* references
  - Both incremental CPU and shared pool memory overhead
- When many database objects are referenced by many users on high-throughput systems...
  - It may pay to use fully qualified object references
  - BUT this is a form of hard-coding that can have maintenance implications

PL/SQL Tuning & Best Practices

Optimize Algorithms

- Avoid Unnecessary Code Execution
- Answer the Question Being Asked
- Do Lots of Stuff At the Same Time
- Avoid the Heavy Lifting

Avoid Unnecessary Code Execution

- Avoid repetitive code execution
- Optimal collection scanning code
- Minimize database trigger firing

Avoid Repetitive Code Execution

A classic problem area:

```
PROCEDURE process_data (nm_in IN VARCHAR2) IS
BEGIN
   FOR rec IN pkgd.cur
   LOOP
      process_rec (UPPER (nm_in), rec.total_production);
   END LOOP;
END;
```

Easy solution:

```
PROCEDURE process_data (nm_in IN VARCHAR2) IS
   v_nm some_table.some_column%TYPE := UPPER (nm_in);
BEGIN
   FOR rec IN pkgd.cur
   LOOP
      process_rec (v_nm, rec.total_production);
   END LOOP;
END;
```

effdsql.tst
loadlots*.*

Sometimes the problem is less obvious...

Optimal Collection Scanning Code

- Oracle offers three types of collections (index-by tables, nested tables and variable arrays) and a set of methods to modify/access those collections
- A common requirement is to scan the contents of these collections. What is the best way to do that?
- Considerations as you think about writing such code:
  - Index-by tables are not necessarily sequentially filled
  - References to undefined rows raise the NO_DATA_FOUND exception

Using the FOR Loop

- Methods used: COUNT, FIRST and LAST
  - The COUNT method is used to make sure there is something in the table
  - If not, FIRST and LAST return NULL and a numeric FOR loop from NULL to NULL raises VALUE_ERROR

```
BEGIN
   IF birthdays.COUNT > 0
   THEN
      FOR rowind IN birthdays.FIRST .. birthdays.LAST
      LOOP
         DBMS_OUTPUT.PUT_LINE (birthdays(rowind).best_present);
      END LOOP;
   END IF;
END;
```

But I am still making a dangerous assumption about the contents of this PL/SQL table!

Protection in the FOR Loop

- If I want to make sure that my PL/SQL table references never raise NO_DATA_FOUND, I should use the EXISTS operator

```
BEGIN
   IF birthdays.COUNT > 0
   THEN
      FOR rowind IN birthdays.FIRST .. birthdays.LAST
      LOOP
         IF birthdays.EXISTS (rowind)
         THEN
            DBMS_OUTPUT.PUT_LINE
              (birthdays(rowind).best_present);
         END IF;
      END LOOP;
   END IF;
END;
```

Still, this is far less than ideal as far as algorithms go. Why?

Optimal Table Scanning Code

- Using the FIRST and NEXT methods, I don't have to worry about table sparseness and I don't have to use EXISTS to check my next row for existence
  - NEXT returns NULL when you are at the highest row
  - To scan in reverse, start with LAST and use PRIOR

```
   rowind PLS_INTEGER := birthdays.FIRST;
BEGIN
   LOOP
      EXIT WHEN rowind IS NULL;
      DBMS_OUTPUT.PUT_LINE
         (birthdays(rowind).best_present);
      rowind := birthdays.NEXT (rowind);
   END LOOP;
END;
```

plsqlloops.sp

- Note: you can also take advantage of various sorting algorithms, depending on how the data is stored in a densely-filled collection

Quiz!

- How can I optimize this code?

```
PROCEDURE exec_line_proc (line IN INTEGER)
IS
BEGIN
   IF line = 1 THEN exec_line1; END IF;
   IF line = 2 THEN exec_line2; END IF;
   IF line = 3 THEN exec_line3; END IF;
   ...
   IF line = 2045 THEN exec_line2045; END IF;
END;
```

slowalg_q2.sql
slowalg_a2.sql

Minimize Firing of Database Triggers

```
CREATE OR REPLACE TRIGGER check_raise
   AFTER UPDATE OF salary, commission
   ON employee FOR EACH ROW
WHEN (NVL (OLD.salary, -1) != NVL (NEW.salary, -1) OR
      NVL (OLD.commission, -1) != NVL (NEW.commission, -1))
BEGIN
   ...
```

- Use the WHEN and UPDATE OF clauses
  - Trigger will not fire unless the NEW salary is different from the OLD -- and both are NOT NULL
  - Trigger will not fire unless salary or commission are referenced in the UPDATE statement
- Disable trigger execution when you are certain of the data.
  - Commonly applicable to batch processes
  - The settrig.sp file contains a generalized mechanism to do this

settrig.sp

Answer the Question Being Asked

- Are you a good listener? Listening to what other people say is an excellent skill to have and develop -- and it applies to programming as well
- All too often, we don't listen or read carefully enough to the requirement -- and we answer the wrong question

Do We Have Any Rows?

```
CREATE OR REPLACE PROCEDURE drop_dept
   (deptno_in IN NUMBER, reassign_deptno_in IN NUMBER)
IS
   temp_emp_count  NUMBER;
BEGIN
   -- Do we have any employees in this department to transfer?
   SELECT COUNT(*)
     INTO temp_emp_count
     FROM emp WHERE deptno = deptno_in;

   -- Reassign any employees
   IF temp_emp_count >0
   THEN
      UPDATE emp
         SET deptno = reassign_deptno_in
       WHERE deptno = deptno_in;
   END IF;

   DELETE FROM dept WHERE deptno = deptno_in;
   COMMIT;
END drop_dept;
```

- *How much* is wrong with this code?

The Minimalist Approach

- At least one row?

```
BEGIN
   OPEN cur;
   FETCH cur INTO rec;
   IF cur%FOUND
   THEN
      ...
```

Use an explicit cursor, fetch once and then check the status.

- More than one row?

```
BEGIN
   OPEN cur;
   FETCH cur INTO rec;
   IF cur%FOUND
   THEN
      FETCH cur INTO rec;
      IF cur%FOUND
      THEN
         ...
```

Use an explicit cursor, fetch once and then fetch *again*. "Two times" is the charm.

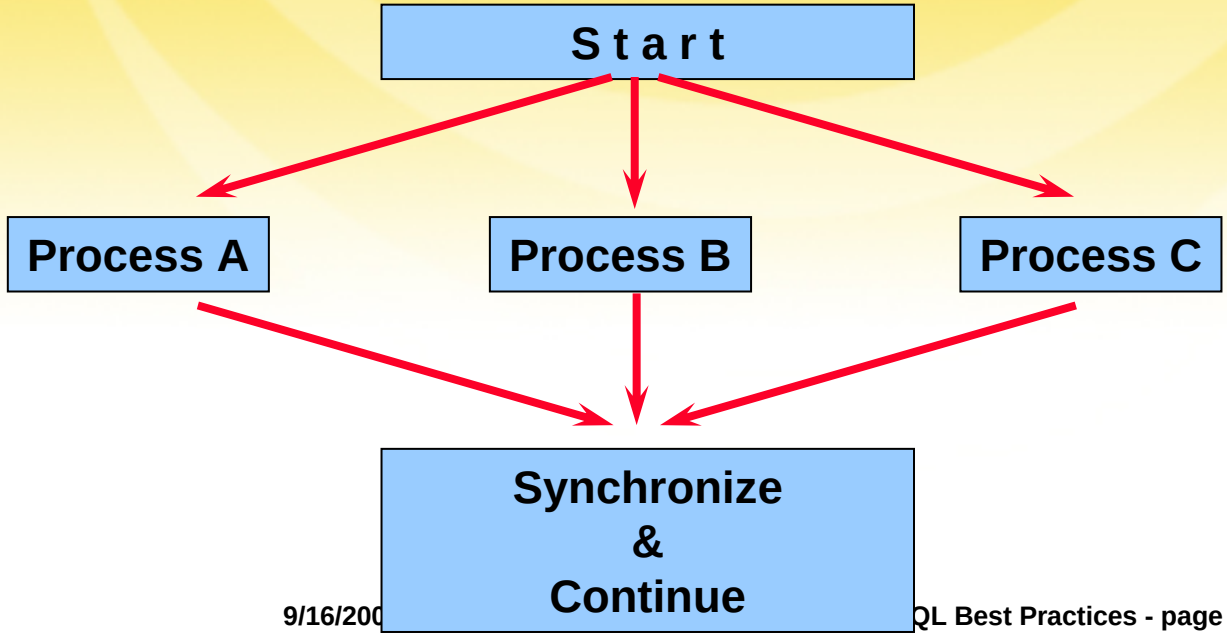
atleastone.sql

Do Lots of Stuff At the Same Time

- Big computers, powerful software...we should do all we can to take full advantage of those resources
- PL/SQL does not offer support for multi-threading, but you do have some options for simultaneous execution of multiple programs in your application:
  - Run programs in parallel with DBMS_PIPE
  - Schedule asynchronous execution with DBMS_JOB

Parallelize *Your* Code with Pipes

- Oracle uses DBMS_PIPE to improve RDBMS performance
  - Parallel updates of indexes, partitioned queries, etc.
- You can do the same for your application if:
  - You have multiple CPUs available
  - You have processes which can run in parallel; they mustn't be inter-dependent



parallel.sql

QUEST SOFTWARE

Schedule Asynchronous Execution

- You can also use DBMS_JOB to execute programs in another Oracle session
  - The job can be run immediately, or at a specific time (and at a specific interval. You will probably need to COMMIT to get the job going
  - DBMS_JOB does not offer any inter-job relationships, so you have to built in a pause for confirmation and so on

```
DECLARE
   job# PLS_INTEGER;
   PROCEDURE doit (job IN VARCHAR2) IS BEGIN
      DBMS_JOB.SUBMIT (
         job#, 'calc_comp (''' || job || ''')', SYSDATE, NULL);
      COMMIT;
   END;
BEGIN
   doit ('CLERK');
   doit ('VP');
   doit ('PROGRAMMER');
END;
```

Avoid the Heavy Lifting

- For reasons of both efficiency and productivity, you should always check to see if a solution is already available before you build it yourself
  - If Oracle wrote it, it is probably implemented in C, increasing the likelihood that it will run faster than what you produce
- Familiarize yourself with the built-in functionality, from STANDARD-based functions to the built-in packages
- Consider the following requirements:
  - Return TRUE if a string is a valid number, false otherwise
  - Search for the Nth occurrence of a substring within a string, and return the rest of the string from that point

isnum*.*
fromnth.*

Quiz!

- Assuming that **process_employee_history** has been optimized (a big assumption!), how else can we improve the performance of this code?

```
DECLARE
   CURSOR emp_cur
   IS
      SELECT last_name, TO_CHAR (SYSDATE, 'MM/DD/YYYY') today
        FROM employee;
BEGIN
   FOR rec IN emp_cur
   LOOP
      IF LENGTH (rec.last_name) > 20
      THEN
         rec.last_name := SUBSTR (rec.last_name, 1, 20);
      END IF;
      process_employee_history (rec.last_name, rec.today);
   END LOOP;
END;
```

slowalg_q1.sql
slowalg_a1.sql

Quiz!

- Construct a schedule of lease payments for a store and save them in a PL/ SQL table
  - For each of 20 years, the lease amount is calculated as the sum of lease amounts for the remaining years
  - The total lease for year 10, in other words, would consist of the lease amounts (fixed and variable) for years 10 through 20
- Functions returning the lease amounts are provided (dummies): pv_of_fixed, pv_of_variable
- How best can we implement this requirement?

presvalue.sql

Quiz!

```
SQL> BEGIN
  2     p.l (ltrim ('abcabcbadef', 'abc'));
  3     p.l (lstrip ('abcabcbadef', 'abc', 2));
  4  END;
  5  /
def
badef
```

- LTRIM and RTRIM trim specified characters, but do not recognize *patterns*
- Challenge: build a function that combines trim capabilities with REPLACE, which replaces patterns instead of individual characters

lstrip*.sf
lstrip.tst

PL/SQL Tuning and Best Practices

Use Data Structures Efficiently

Use Data Structures Efficiently

- When VARCHAR2 is not really VARCHAR2...
- Rely on PLS_INTEGER
- Avoid Implicit Conversions
- Minimizing Transfer of Large Structures
- Don't Use NOT NULL Constraints
- Defer CPU and Memory Utilization Until Needed
- Cache Data with Packages
- Leverage Oracle Hashing

When VARCHAR2 is not really VARCHAR2

- Before Oracle8, variables declared as VARCHAR2 were treated computationally as variable length, but memory was allocated as fixed-length. Beware!
- On Oracle 7.3, you can use the following script to soak up all the real memory available in your system:

```
DECLARE
   TYPE big_type IS TABLE OF VARCHAR2(32767)
      INDEX BY BINARY_INTEGER;
   big big_type;
BEGIN
   DBMS_OUTPUT.enable;
   FOR i IN 1 .. 32767
   LOOP
      big (i) := NULL;
   END LOOP;
END;
```

Note: memory is released when you end the session

Rely on PLS_INTEGER

- PLS_INTEGER operations use machine arithmetic, making it the most efficient datatype for integer manipulation
  - It has the same range as BINARY_INTEGER (-2147483647 .. 2147483647)
  - Performance gains will vary greatly depending on the integer computations
- Prior to Oracle8, watch out for differences in behavior between BINARY_INTEGER and PLS_INTEGER
  - Use of PLS_INTEGER might raise an overflow exception, while use of BINARY_INTEGER does not
  - In Oracle8 and above, use of either raises consistent ORA-01426 errors (NUMERIC_OVERFLOW)

plsint*.sql
overflow.tst

Avoid Implicit Conversions

- If you are not careful, you can easily rely on the PL/SQL engine to perform implicit (and perhaps unnecessary) datatype conversions
  - The result is minor performance degradation

Instead of this...

```
DECLARE
   num NUMBER;
BEGIN
   num := num + 1;
```

Do this...

```
DECLARE
   num NUMBER;
BEGIN
   num := num + 1.0;
```

```
DECLARE
   str VARCHAR2(100);
   int PLS_INTEGER;
BEGIN
   str := str || int;
```

```
DECLARE
   str VARCHAR2(100);
   int PLS_INTEGER;
BEGIN
   str := str || TO_CHAR (int);
```

implconv.tst

Minimize Transfer of Large Structures

- Default parameter passing in PL/SQL is by value, not reference
  - This means that IN OUT parameters are copied *into* local data structures, upon which changes are made. The local contents are then copied back to the parameter upon successful completion
- Large IN OUT parameters (records and collections, primarily) can cause performance degradation
- Two solutions to this problem:
  - Move data structure out of parameter list and into a package Specification (making it a "global")
  - Use the NOCOPY feature of Oracle8i
- You can also use the Oracle8i UTL_COLL to tune access to nested tables

Move Structure to Package Spec.

- You can avoid passing a large structure by defining it in the specification of a package, making it globally accessible
  - You can then remove it from the parameter list of your program and reference it directly within the program
  - SAFEST way to do this is to put the collection or record itself in the package body and provide programs to access that structure
  - Note: performance gains will vary greatly depending on what kind of operations you perform on the structure
- When you do this, you lose some flexibility
  - Only once instance of the global can be manipulated within your session
  - This is quite different from passing any number of different structures as parameters

pkgvar.pkg
pkgvar.tst

Use the NOCOPY Feature of Oracle8i

```
PROCEDURE nocopydemo (
   some_collection IN OUT NOCOPY some_collection_tabtype,
   some_object OUT NOCOPY some_object_type);
```

- Most useful for collections, records and objects, but remember:
  - When you specify NOCOPY, you are only making a request of or suggestion to the compiler
- Your request to use NOCOPY will be ignored when:
  - Program is called via RPC (remote procedure call)
  - You pass in just an element of a coll3ection
  - Collection elements have a constraint (e.g., NOT NULL)
  - Parameters are records with anchored declarations
  - Implicit datatype conversions are required
  - And more...see the documentation

nocopy*.tst

Nasty Tradeoff with By Reference Passing

- With conventional call by value:
  - Parameter value changes are “rolled back” on an exception in called routine
  - May be preferred for data integrity -- behavior is in accord with Oracle's read consistent principles
- With call by reference (NOCOPY and using globals):
  - Changes made to to a by-ref parameter take effect immediately and are *not* rolled back when an exception occurs in your program. The result may be a data structure with partial changes
- Watch out for parameter aliasing problems (an issue for by-value parameter passing as well)
  - Issue arises when you pass a global structure as a parameter *and* directly reference/change that global inside the program

parmalias.sql

Tune Access to Nested Tables

- UTL_COLL is a new Oracle8i package that allows you to take advantage of stored nested table "locators"
- You can avoid materializing a large collection into memory by instead accessing it via locator
  - Handy when you only want to access a portion of the collection's attributes

utlcoll*.sql

```
CREATE OR REPLACE FUNCTION getpets_like
   (petlist IN Pettab_t, like_str IN VARCHAR2)
   RETURN pettab_t
IS
   list_to_return Pettab_t := Pettab_t();
   onepet Pet_t;
   counter PLS_INTEGER := 1;
BEGIN
   IF UTL_COLL.IS_LOCATOR(petlist)
   THEN
      FOR theRec IN
         (SELECT VALUE(l) apet
           FROM TABLE(CAST(petlist AS Pettab_t)) l
          WHERE l.name LIKE like_str)
      LOOP
         list_to_return.EXTEND;
         list_to_return(counter) := theRec.apet;
         counter := counter + 1;
      END LOOP;
   ELSE
      FOR i IN 1..petlist.COUNT
      LOOP
         IF petlist(i).name LIKE like_str
         THEN
            list_to_return.EXTEND;
            list_to_return(i) := petlist(i);
         END IF;
      END LOOP;
   END IF;
   RETURN list_to_return;
END;
```

Avoid NOT NULL Constraint

- Oracle recommends that you avoid applying the NOT NULL constraint to a declaration, and instead check for a NULL value yourself
- Instead of this:

```
DECLARE
   my_value INTEGER NOT NULL := 0;
BEGIN
   IF my_value > 0 THEN ...
END;
```

- Do this:

```
DECLARE
   my_value INTEGER := 0;
BEGIN
   IF my_value IS NULL THEN /* ERROR! */
   ELSIF my_value > 0 THEN ...
END;
```

notnull.tst

- Note: my tests indicate that the savings are nominal

Defer Memory and CPU Allocation

- Use nested, anonymous blocks to defer allocation of memory and even CPU utilization

Before

```
PROCEDURE always_do_everything
    (...)
IS
   big_string VARCHAR2(32767) :=
      ten_minute_lookup (...);
   big_list
      list_types.big_strings_tt;
BEGIN
   IF <condition>
   THEN
      use_big_string (big_string);
      Process_big_list (big_list);
   ELSE
      /* Nothing big
         going on here */
      ...
   END IF;
END;
```

defer.sql

After

```
PROCEDURE only_as_needed (...) IS
BEGIN
   IF <condition>
   THEN
      DECLARE
         big_string VARCHAR2(32767) :=
           ten_minute_lookup (...);
         big_list
           list_types.big_strings_tt;
      BEGIN
         use_big_string (big_string);
         Process_big_list (big_list);
      END;
   ELSE
      /* Nothing big
         going on here */
      ...
   END IF;
END;
```

Cache Data with Packages

- General principle: if you need to access the same data and it doesn't change, cache it in the most accessible location to maximize lookup performance
- Packages offer an ideal caching mechanism
  - Any data structure defined at the package level (whether in specification or body) serves as a persistent, global structure
  - Remember: separate copy for each connection to Oracle
- Let's explore further:
  - Define, protect and access package data
  - Use the initialization section

Define, Access and Protect Pkg Data

- Great example: the USER function
  - The value returned by USER never changes in a session
  - Each call to USER is in reality a SELECT FROM dual
  - So why do it more than once?

```
CREATE OR REPLACE PACKAGE thisuser
IS
   FUNCTION name RETURN VARCHAR2;
END;

CREATE OR REPLACE PACKAGE BODY thisuser
IS
   /* Persistent "global" variable */
   g_user VARCHAR2(30) := USER;

   FUNCTION name RETURN VARCHAR2 IS
   BEGIN
      RETURN g_user;
   END;
END;
```

Hide package data!

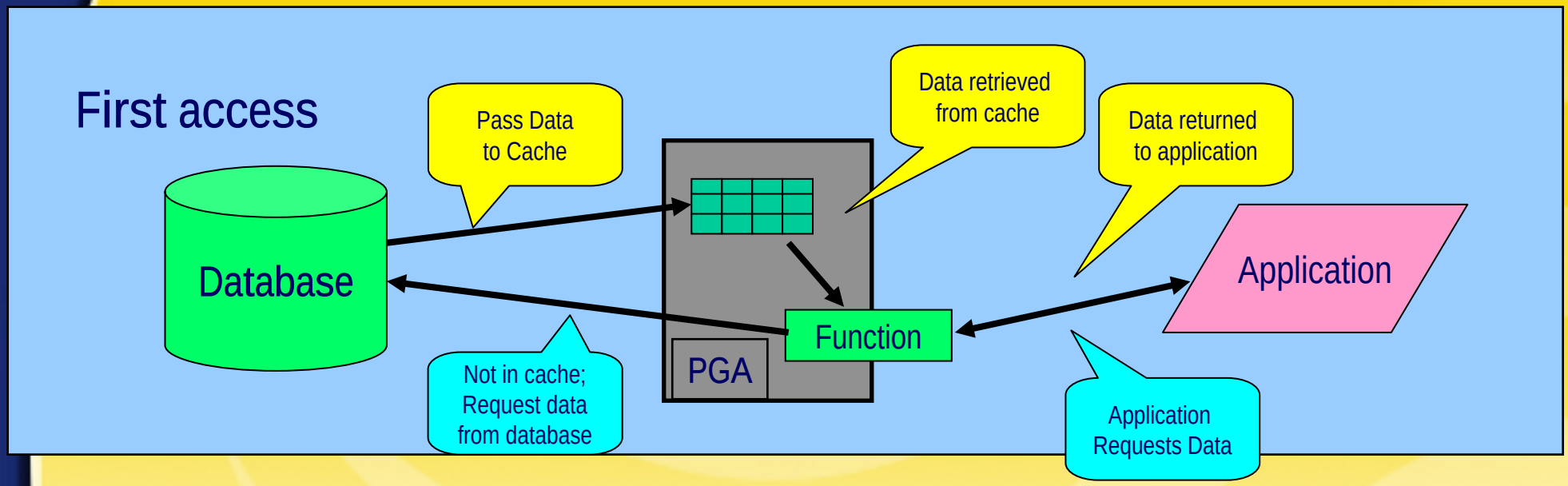
If exposed, you cannot guarantee integrity of data. Build "get and set" programs around it.

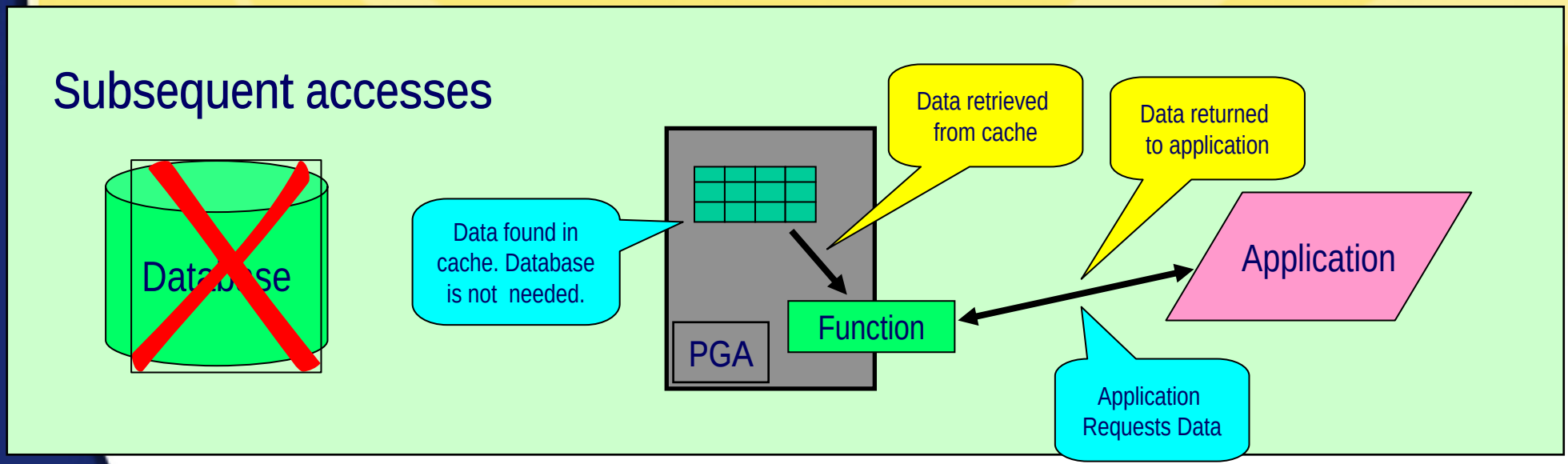
thisuser.pkg
thisuser.tst

Optimize Lookups with a Local Cache

- Common application requirement: look up name or description for foreign key in a table
  - Common, but expensive operation in a client-server environment
- Wouldn't it be nice to be able to have the application *remember* that it just queried that value?
  - In many cases, the value hasn't changed (how many "code" tables do you have?)
- While Oracle does optimize performance of repetitive access, you still pay the price for going through the SQL layer
  - Perhaps we can take advantage of a local cache (per user) to speed up performance

Data Caching with PL/SQL Tables





QUEST SOFTWARE

Code for a Caching Function

```
PACKAGE te_company                                    Specification
IS
   FUNCTION name$val (id_in IN company.company_id%TYPE)
      RETURN company.name%TYPE;
END te_company;
```

- You must place the retrieval function inside a package
  - The package body provides persistence for PL/SQL table
  - Note that the specification gives no indication of the technique used to return the company name

```
PACKAGE BODY te_company                               Body...
IS
   TYPE names_tabtype IS
      TABLE OF company.name%TYPE INDEX BY BINARY_INTEGER;
   names names_tabtype;
```

...continued on next slide

Body of Self-Optimizing Function

```
   FUNCTION name$val (id_in IN company.company_id%TYPE)
      RETURN company.name%TYPE
   IS
      CURSOR comp_cur IS
         SELECT name FROM company where company_id = id_in;
      retval company.name%TYPE;
   BEGIN
      IF names.EXISTS (id_in)
      THEN
         retval := names (id_in);
      ELSE
         OPEN comp_cur; FETCH comp_cur INTO retval;
         CLOSE comp_cur;

         IF retval IS NOT NULL
         THEN
            names (id_in) := retval;
         END IF;
      END IF;
      RETURN retval;
   END name$val;
END te_company;
```

If the row is cached, retrieve from cache.

If not in cache, use standard database retrieval.

If a match was found store it in the cache for next time.

emplu.pkg
emplu.tst

Another Tool, To Be Applied Selectively

- Local caching isn't always appropriate
  - The data should be static. If the table from which you cached data changes, it is very difficult to update the cache
  - Either for small tables or for large tables with "hot spots"
- You can produce variations on this theme:
  - Pre-load the entire table into memory. Might be handy for small tables
  - Add a flag to your function to allow either database retrieval or cached retrieval, modifiable at run-time
  - Create a "system-wide" cache that is accessible to all sessions. This can be done using database pipes and is explored in the Oracle Built-in Packages seminar

Leverage Oracle Hashing

- Hashing algorithms transform strings to numbers
  - Standard usage: generate unique values for distinct strings

```
FUNCTION DBMS_UTILITY.GET_HASH_VALUE
   (name IN VARCHAR2,
    base IN NUMBER,
    hash_size IN NUMBER)
RETURN NUMBER;
```

- Provide the string, the base or starting point, and the hash size (total number of possible return values).
- Tips for hashing:
  - You must use the same base and hash size to obtain consistent hash values
  - Maximum hash size: upper limit of BINARY_INTEGER: 2**31-1
  - No guarantee that two different strings will *not* hash to the same number. Check for and resolve conflicts

A Hash Encapsulator

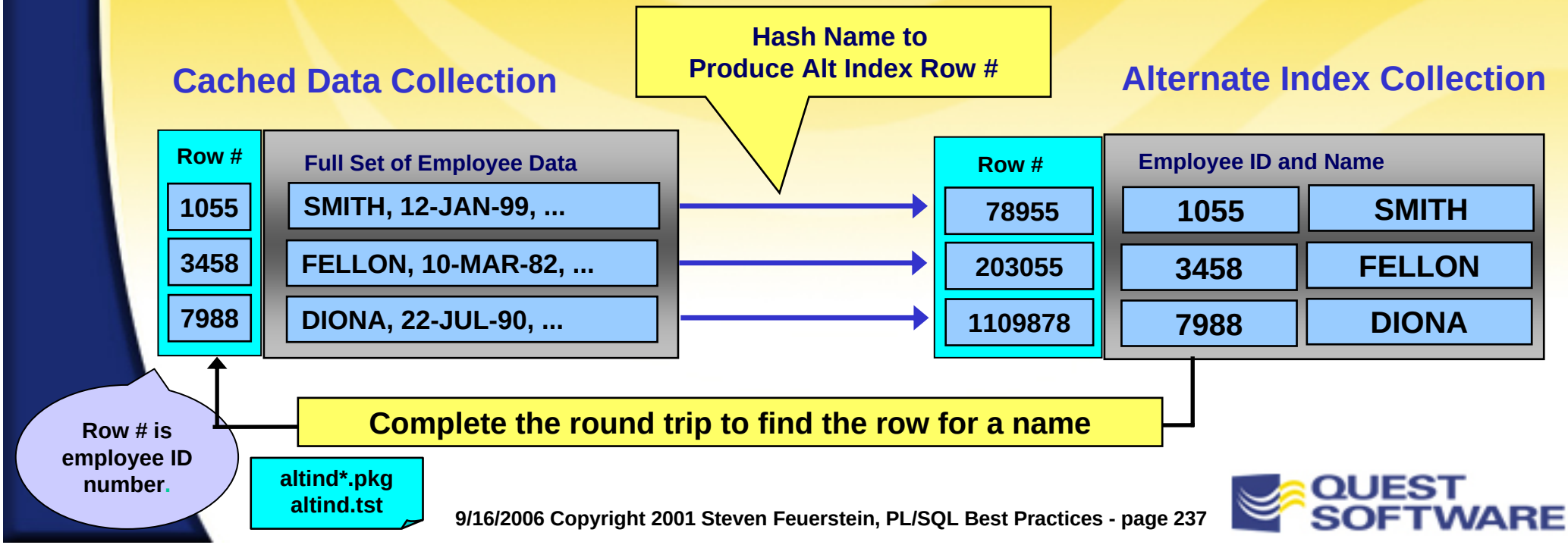
- Hide base and hash table values to ensure consistency

```
CREATE OR REPLACE PACKAGE hash
IS
    FUNCTION val (str IN VARCHAR2) RETURN NUMBER;
END hash;
/
CREATE OR REPLACE PACKAGE BODY hash
IS
    maxRange CONSTANT BINARY_INTEGER := POWER (2, 31) - 1;
    strt CONSTANT BINARY_INTEGER := 2;

    FUNCTION val (str IN VARCHAR2) RETURN NUMBER IS
        retval NUMBER := 0;
    BEGIN
        RETURN
           DBMS_UTILITY.GET_HASH_VALUE (str, strt, maxRange);
    END val;
END hash;
/
```

hash.pkg
hashcomp.tst

Hashing for an Alternative Index

- Index-by tables support only one index: the row number
  - So to locate the row in which a particular string is located, you have to do a "full table scan" -- or do you?
- Use the hash function to build an alternative index to the contents of the PL/SQL table



Quiz!

slowds*.sql

- How can we tune up this program?
- Remember: You will not usually find big gains in data structure tuning, but the cumulative effect can be impressive.

```
CREATE OR REPLACE PROCEDURE fix_me (
   nmfilter IN VARCHAR2, comm_list IN OUT comm_pkg.reward_tt
   )
IS
   v_nmfilter VARCHAR2(2000) NOT NULL := nmfilter;
   v_info VARCHAR2(2000);
   v_nth INTEGER;
   v_sal NUMBER;
   indx INTEGER
BEGIN
   FOR indx IN comm_list.FIRST .. comm_list.LAST
   LOOP
      v_nth := v_nth + 1;
      v_info :=
         'Doubled ' || v_nth || 'th salary on ' ||
         SYSDATE || '  to ' || comm_list(indx).sal * 2;

      IF UPPER (comm_list(indx).nm) LIKE
            UPPER (v_nmfilter)
      THEN
         UPDATE employee
            SET salary := comm_list(indx).sal * 2.0,
                info :=  v_info,
                commission := comm_list.comm
          WHERE last_name = UPPER (comm_list(indx).nm);
         comm_list(indx).comm := 0;
      END IF;
   END LOOP;
END;
```

PL/SQL Tuning and Best Practices

Manage Code in the Database and SGA

Manage Code in the Database/SGA

- PL/SQL code is stored in the Oracle data dictionary and executed from the SGA's shared pool area
  - It is imperative that you understand the PL/SQL architecture and then tune access to your code
- PL/SQL Architecture in Shared Memory
- Analyze Code Usage & Dependencies
- Manage Memory Allocation and Usage

Analyze Code Usage & Dependencies

- How big is my code?
- What does my code need in order to run?
- What is cached in the SGA?
- The V$DB_OBJECT_CACHE view
- The V$SQLAREA view

How Big is My Code?

- Oracle offers the *_OBJECT_SIZE views to give you information about the size of your code. There are three different entries of interest:
  - Source size: the source code, used to recompile the program
  - Parsed size: Code required to compile other programs that reference this one. If you "keep" a program unit, this code is also loaded into the SGA
  - Code size: partially-compiled code that is loaded into the SGA
- Use the view to identify large programs and compare sizes of different implementations
- Large program units are generally good candidates for pinning into the SGA (covered later)

pssize.sql

What Does My Code Need to Run?

- If program A calls program B, then both A and B need to be present in the SGA for A to execute
- Oracle maintains dependency information in a variety of views (*_DEPENDENCIES and PUBLIC_DEPENDENCY)
- Analyze complete memory requirements!
  - Combine dependency information with USER_OBJECT_SIZE to get a more complete picture
  - Or take snapshots (before and after) of what is in the SGA (more on this later)
- Watch out! Accessing these DD views can be very time-consuming (to learn and to execute).

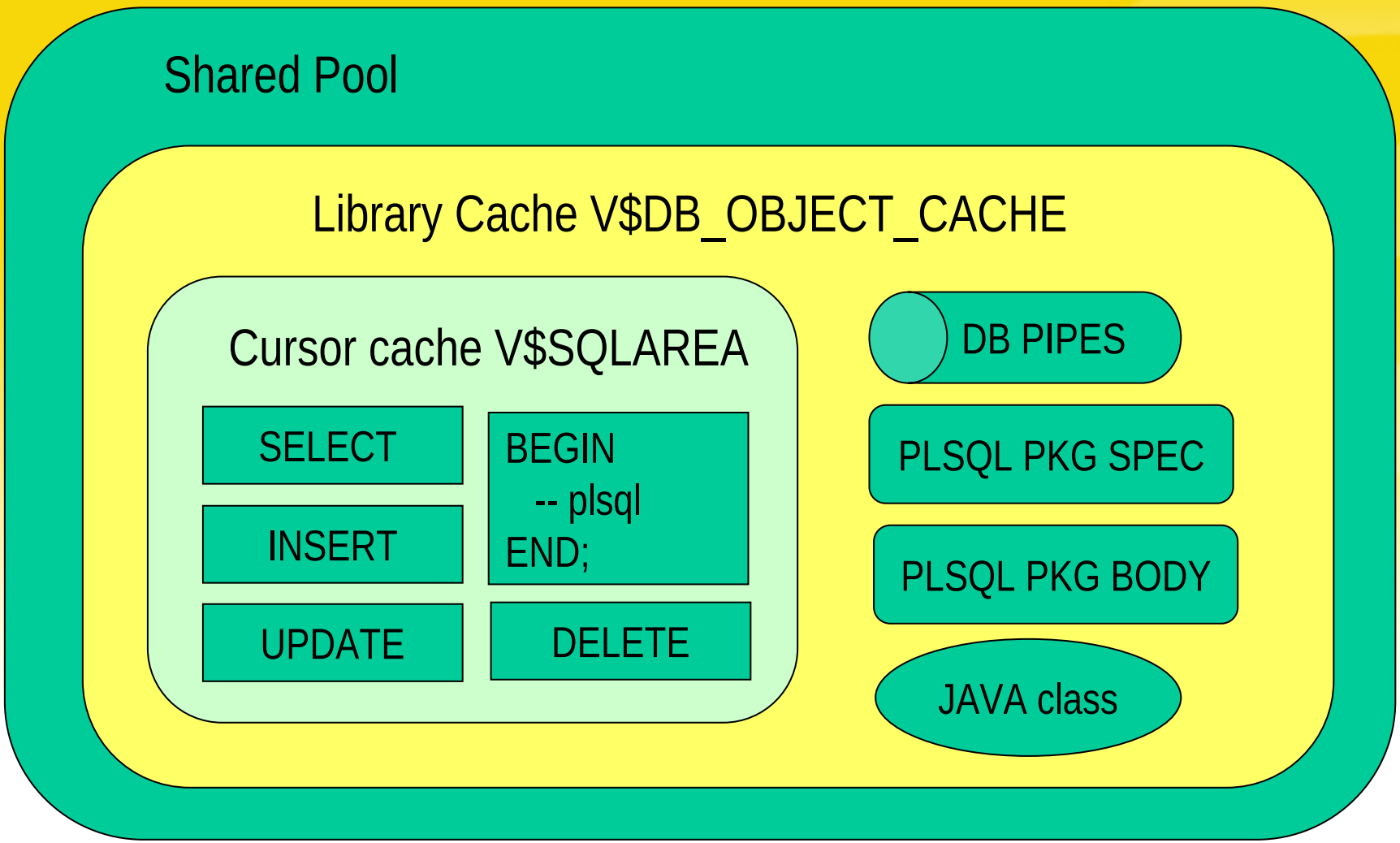
analyzedep.ins
analyzedep.sql
analyzedep.tst
utldtree.sql

What is Cached in the SGA?

- The best way to understand the requirements and activity of the PL/SQL code in the SGA is to look at the SGA
- Oracle offers a variety of data structures to get this information:
  - V$ROWCACHE: check for data dictionary cache hits/misses
  - V$LIBRARYCACHE: check for object access hits/misses
  - V$SQLAREA: statistics on shared SQL area, one row per SQL string (cursor or PL/SQL block)
  - V$DB_OBJECT_CACHE: displays info on database objects that are cached in the library cache
- And much more. We will just scratch the surface and pull out those items most useful for PL/SQL tuning

grantv$.sql
insga.pkg

Library Cache views



The V$DB_OBJECT_CACHE View

- The V$DB_OBJECT_CACHE view gives you this information:
  - LOADS: number of times element has been loaded to the SGA
  - EXECUTIONS: number of times element has been executed. This is *not* aggregated for the program name. Only for the code block.
  - KEPT: YES if the element has been pinned
  - TYPE: NOT LOADED if the element has been referenced but not loaded
- Excessive numbers of loads and/or high execution counts indicate the need for adjusting the shared pool size and/or pinning that program unit

The V$SQLAREA View

- This view gives you lots of information about the contents of the shared pool and parsed SQL statements
- Is a subset of V$DB_OBJECT_CACHE and does not have separate aggregate entries for individual program units
- You can analyze statements for memory and disk usage, repetitive SQL, and so on

```
SQL> exec insga.show_similar
*** Possible Statement Redundancy:
     begin fix_me (1); end;
     begin fix_me(1); end;
*** Possible Statement Redundancy:
     select * from EMP
     select * from emp
```

insga.pkg
similar.sql

Tune Memory Allocation and Usage

- Set the SGA shared pool size
- Manage session memory with DBMS_SESSION
- Use the SERIALLY_REUSABLE Pragma
- Minimize program inter-dependencies

Size the Shared Pool

- The shared pool must be large enough to cache your code without excessive (or any) swapping
  - Oracle uses the least-recently-used algorithm to make room for code that needs to be executed
  - Tune size of shared pool through analysis of contents of shared pool when application is up and running
- INIT.ORA parameters of interest:
  - SHARED_POOL_SIZE: size in bytes of shared pool area
  - SHARED_POOL_RESERVED_SIZE: size in bytes of portion of shared pool area reserved for requests larger than SHARED_POOL_MIN_ALLOCATION when the shared pool does not have enough contiguous blocks for request.

```
SQL> exec insga.show_hitratio
ROW CACHE Hit ratio =  93.81%
   Ratio is below 95%. Consider raising the SHARED_POOL_SIZE init.ora
parameter.
LIBRARY CACHE Hit ratio =  99.61%
```

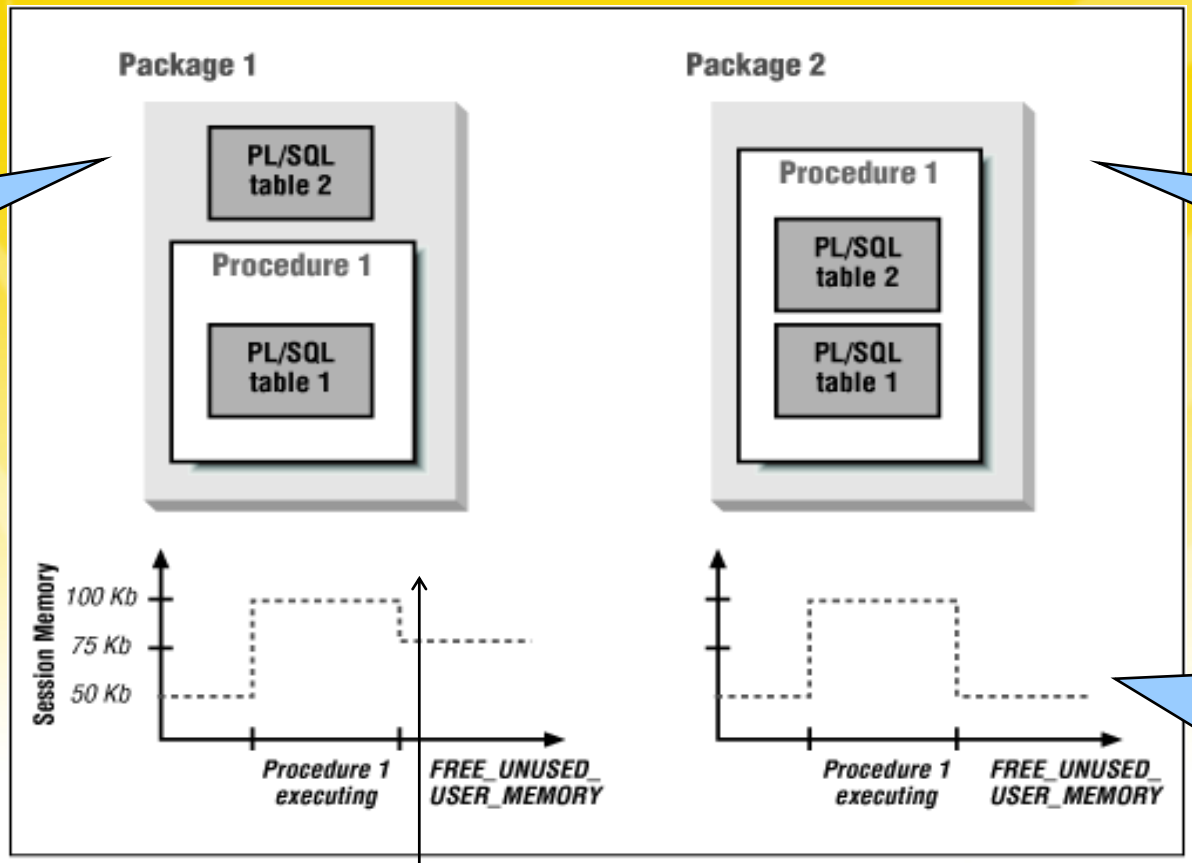
Manage Session Memory

- DBMS_SESSION.RESET_PACKAGE
  - Reinitializes all package states, your own and even the built-in packages like DBMS_OUTPUT
  - Never use in stored PL/SQL (embedded within an application); package states will not be re-instantiated until outer-most PL/SQL block within which RESET_PACKAGE was called completes
  - RESET_PACKAGE does *not* release memory associated with cleared package variables (see next program)
- DBMS_SESSION.FREE_UNUSED_USER_MEMORY
  - Returns memory back to OS or shared pool
  - Call after compilation of large objects, when large PL/SQL tables are no longer needed, or after large sort operations
- Recommendation: clear out memory before you start a profiling session to "make all things equal"

reset.sql
resetmem.sql
mysess.pkg

Freeing Unused User Memory (FUUM)

One local table, one "global" table



Two local tables

After procedure execution and FUUM, no memory allocated.

After procedure execution and FUUM, global memory still allocated.

QUEST SOFTWARE

Pin Code in the SGA

- "Pin" your code (including triggers and types) into shared memory, and exempt it from the LRU algorithm
  - When you pin using, your code remains in the SGA until the instance is shutdown
- Pinning is usually performed right after database startup. Prior to Oracle 7.3, two separate steps were required
  - 1. Register that program, package or cursor to be pinned by calling the DBMS_SHARED_POOL.KEEP procedure
  - 2. Reference the code element so that it will be loaded into memory.
  - This second step is now performed with the call to KEEP
- The DBMS_SHARED_POOL is not automatically installed/available in all versions. See pool.ins.

pool.ins
plvpin.sp
keeper.sql
showkeep.sql

Use the SERIALLY_REUSABLE Pragma

- By default, package-level data structures persist for the entire session and are stored in the user memory area
  - Problematic memory consumption with large numbers of users
- Oracle8 offers a new pragma to instruct the runtime engine to release memory after each package instruction
  - And shift allocation of memory to the SGA, not PGA/UGA
- Especially handy:
  - For closing package-level cursors automatically
  - Releasing memory used for collections and large VARCHAR2s

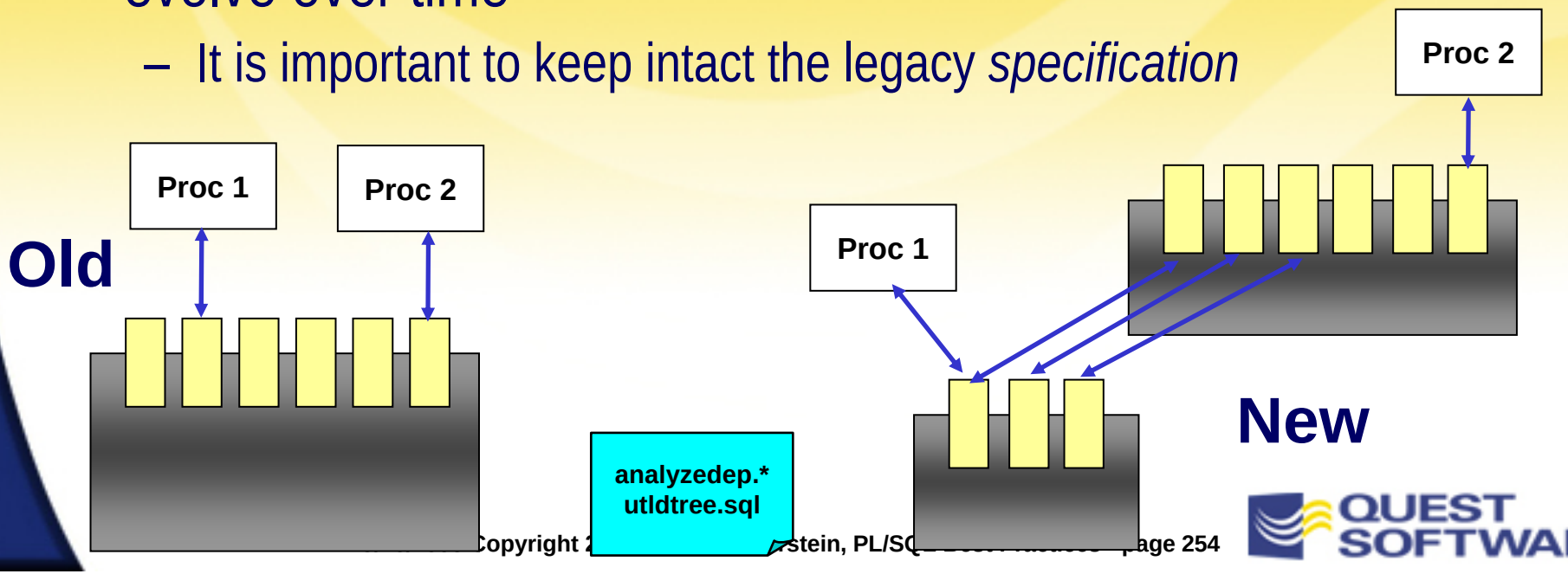
serial.sql

```
CREATE OR REPLACE PACKAGE one_use_pkg
IS
   PRAGMA SERIALLY_REUSABLE;
   CURSOR must_close_cur IS SELECT ...;
   big_collection lotsa_strings_tabtype;
END one_use_pkg;
```

Designate in both specification and body.

Minimize Inter-Program Dependencies

- As you write more and more PL/SQL code and seek to re-use that code, you need to be careful about:
  - The dependencies you establish between programs and packages
  - What programs you put into which packages
- You may sometimes need to re-architect packages as they evolve over time
  - It is important to keep intact the legacy *specification*



QUEST SOFTWARE

PL/SQL Tuning & Best Practices

Creating Readable & Maintainable Code

Writing Code You Can Be Proud Of

- And code you -- as well as others -- can maintain with an absolute minimum of headaches
  - Use the full range of constructs in the language
  - Achieve a consistent, readable coding style
  - Choose names carefully for program elements
  - Document to add value
  - Keep the execution section tight, and its meaning and purpose transparent

Tips for organizing source code in files

Use Everything the Language Offers

- PL/SQL is based on Ada, and is a highly structured and robust language. Use it, all of it! Some examples...

If a variable's value doesn't change, declare a constant. This both protects the value and also documents for others that it should not be changed.

Use labels for anonymous blocks and loops. They're not just or GOTOs.

```
c_max_date CONSTANT DATE DEFAULT SYSDATE;
```

```
<<every_month>>
FOR month_ind IN 1 .. 12
LOOP
   every_month.month_ind ...
END LOOP every_month;
```

Use the native BOOLEAN datatype

The Boolean: A True Pleasure

- The database doesn't support Boolean columns, but that's no excuse to avoid PL/SQL's datatype
  - A Boolean can have one of three values: TRUE, FALSE and NULL

You don't have to use Y or N, T or F, etc. to *represent* these actual values.

```
IS
   valid_company CHAR(1);
BEGIN
   IF valid_company = 'Y'
```

Nasty!

```
IS
   valid_company BOOLEAN;
BEGIN
   IF valid_company = TRUE
```

Better...

```
IS
   valid_company BOOLEAN;
BEGIN
   IF valid_company
```

Best.

Achieving a Consistent Coding Style

- Programs are more readable and easier to maintain when a consistent and effective coding style is applied
  - Each developer should have a consistent style. Best of all? An entire team shares a common style
- There are two ways to achieve this objective:
  - Make sure everyone writes code the same way
  - Use a utility to format the code after or while it is written
- Chapter 3 of Oracle PL/SQL Programming offers 34 pages on the Feuerstein idea of style
- PL/Formatter from RevealNet is the first commercial "pretty printer" for PL/SQL code
  - Check for others as well!

Hard-to-Read SQL Statement

```
SELECT company_id, sales,
address1, company_type_ds description, min_balance
FROM
company A, sales_amount B, company_type C, company_config D,
region E, regional_status F
WHERE
A.company_id = B.company_id and
A.company_type_cd = C.company_type_cd
and C.company_type_cd = D.company_type_cd and A.region_cd =
E.region_cd and E.status = F.status;
```

- No, none of us ever write anything like this, right?
- The ABCs of table aliases are particularly informative: we have consistency, but we don't have *added value*
  - Rules for rules' sake are not necessarily an improvement

A Non-Procedural Format for SQL

Logical "middle" of SQL statement

Application-specific elements left-justified

Clause Keywords of Statement right-justified

Meaningful table aliases

No vertical alignment of elements

```
SELECT COM.company_id,
       sales,
       address1,
       company_type_ds description,
       min_balance

  FROM company COM,
       sales_amount SAL,
       company_type TYP,
       company_config CFG,
       region REG,
       regional_status RST

 WHERE COM.company_id      = SAL.company_id
   AND COM.company_type_cd = TYP.company_type_cd
   AND TYP.company_type_cd = CFG.company_type_cd
   AND COM.region_cd       = REG.region_cd
   AND REG.status          = RST.status;
```

The Name is a *Lot* of the Game

- Both the structure and content of the names you give to program elements have a direct impact on the quality of your code
- Potential problems include:
  - The name does not accurately describe what the code element does or represents
  - The same code element is represented by a bewildering array of *different* names
  - The structure of the name misrepresents the code; usually this occurs when you give "procedure names" to functions
  - Names of database objects, such as tables and columns, are used as names of PL/SQL variables

Recommendations for Naming Elements

- Use standard prefixes or suffixes for code elements

Most important aspect of conventions:

Consistent application!

Cursor	company_cur
Record	company_rec
Local variable	l_ename
Global variable	g_ename
Constant	c_lastdate
PL/SQL Table Type	dates_tabtype
PL/SQL Table	dates_tab

- The *syntax* of program names is critical
  - Verb-noun for procedures
  - Preposition for Boolean functions
  - Noun for other functions

```
PROCEDURE calculate_totals

FUNCTION is_valid_type (...)
   RETURN BOOLEAN

FUNCTION total_sales RETURN NUMBER
```

Comment to Add Value

- Provide documentation in the following areas:
  - Standard program header: who, what, when, why
  - Modification histories
  - Explanations of a workaround, patches, etc.
  - A "translation" of a complex or dense section of code
    - As we shall see, though, a better solution than a comment is to make the code less complex and dense
- Avoid fancy comment "boxes" and right margins
  - Don't waste time maintaining the format of your comments
- Keep comments down to an absolute minimum
  - Remember: they are a form of code repetition
  - Best of all is to avoid comments and allow your code to "speak for itself"

The Path to Righteous Code

```
/* If the first field of the properties record is N... */
IF properties_flag.field1 = 'N' THEN
```

A singularly non-informative comment

```
/* If the customer is not eligible for a discount... */
IF properties_flag.field1 = 'N' THEN
```

At least the comment adds some value now!

```
IF customer_flag.discount = constants.ineligible
THEN
```

Better names for variables, named constants got rid of the comment

```
IF NOT customer_rules.eligible_for_discount (customer_id)
THEN
```

Encapsulation with a function offers readability and easy enhancement

constants.sps
custrules.pkg

OK, Use *Some* Words and Paper

- Create checklists to serve as reminders *before* construction, and guidelines for code review *after* completion
  - The following items are offered simply to give you an idea of the topics you might include

- General coding guidelines
  - All naming conventions are followed
  - No hard-coded values
  - Repetitive code is modularized
  - Functions always return values
  - One way in, one way out: loops and programs
  - Unused variables and code sections are removed

Check out Code Complete by Steve McConnell for many more coding tips and checklists.

Organizing Source Code in Files

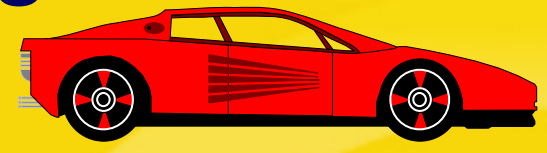
- Use standard file extensions for different kinds of source code: pks for package specs, pkb for package bodies, etc.
- Always separate the "CREATE OR REPLACE" for package specifications and bodies into different files
  - You only recompile your package specifications when needed (minimizing the cascade of INVALID program units)
- Create/compile all of your specifications before you compile your bodies
  - This defines all the headers or stubs for your application code to compile
  - You can then create packages with inter-dependencies and no compilation problems

Creating Readable/Maintainable Code

Summary

- Take Full Advantage of the Language
  - And don't assume you know it all. Take time out regularly for a little studying
- Write Code You Can Be Proud Of
  - From coding style to names to structure, your software is a product of your mind. Why make a mess of it?

You Can Write Blazing Fast PL/SQL!



- Tuning PL/SQL code is an iterative and incremental process
  - You are unlikely to uncover a "silver bullet" that is *not* related to some SQL statement. You can, however, have a substantial impact on the performance of your and others' code
- Write code with efficiency in mind, but save intensive tuning until entire components are complete and you can perform benchmarking
- *MOST IMPORTANT!* Avoid repetition and dispersion of SQL statements throughout your application
- PL/SQL code is executed from shared memory. You *must* tune the shared pool to avoid excessive swapping of code

Closing Comments

- Write code with efficiency in mind, but save intensive tuning until entire components are complete and you can perform benchmarking
- Concerning best practices, give yourself a fighting chance:
  - Use code generators whenever possible; make it hard to *not* employ best practices
  - Write scripts to check source for compliance
  - Share lessons learned and code built

Visit the PL/SQL Pipeline (www.revealnet.com/plsql-pipeline) to share what you learn about tuning and best practices, and to get your questions answered.