ORACLE®

ORACLE®

With Oracle Database 12*c*, there is all the more reason to use database PL/SQL

Bryn Llewellyn,
Distinguished Product Manager,
Database Server Technologies Division
Oracle HQ

ORACLE® DATABASE 12*c*

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Before we start…

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

What is SQL?

- This is *not* SQL

```
select Count(*) from DBA_Objects where Object_Type = 'PACKAGE';
```

- It's one statement, from a sequence of statements, in the SQL*Plus scripting language, when you've earlier done this statement

```
SET SQLTERMINATOR ;
```

in that same language

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

What is SQL?

No terminator!

- This is SQL:

```
select Count(*) from DBA_Objects where Object_Type = 'PACKAGE'
```

- It's a declarative scheme to define a single outcome
- Its scope is the single statement
- It has NO notions for assembling SQL statements into a program
- For that you need a procedural language (not a scripting language)

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

What is PL/SQL?

- It's a procedural language designed explicitly for issuing SQL statements and receiving their results
- It's designed for maximum efficiency (e.g. implicit soft parse avoidance)
- It runs inside the database – eliminates client/server round trips, executes in the same process as the SQL
- It has explicit language features for doing SQL – embedded SQL (name-resolved at compile-time), anchored declarations, automatic conversion of a SQL error to a PL/SQL exception…

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This is PL/SQL

```
<<b>>declare
  n integer not null := -1;
  Object_Type constant DBA_Objects.Object_Type%type not null :=
    'PACKAGE';
begin
  select Count(*)
  into b.n
  from DBA_Objects o
  where o.Object_Type = b.Object_Type;
  if n < 1200 then
    Raise_Application_Error(-20000, 'Not enough PL/SQL!');
  end if;
end b;
```

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

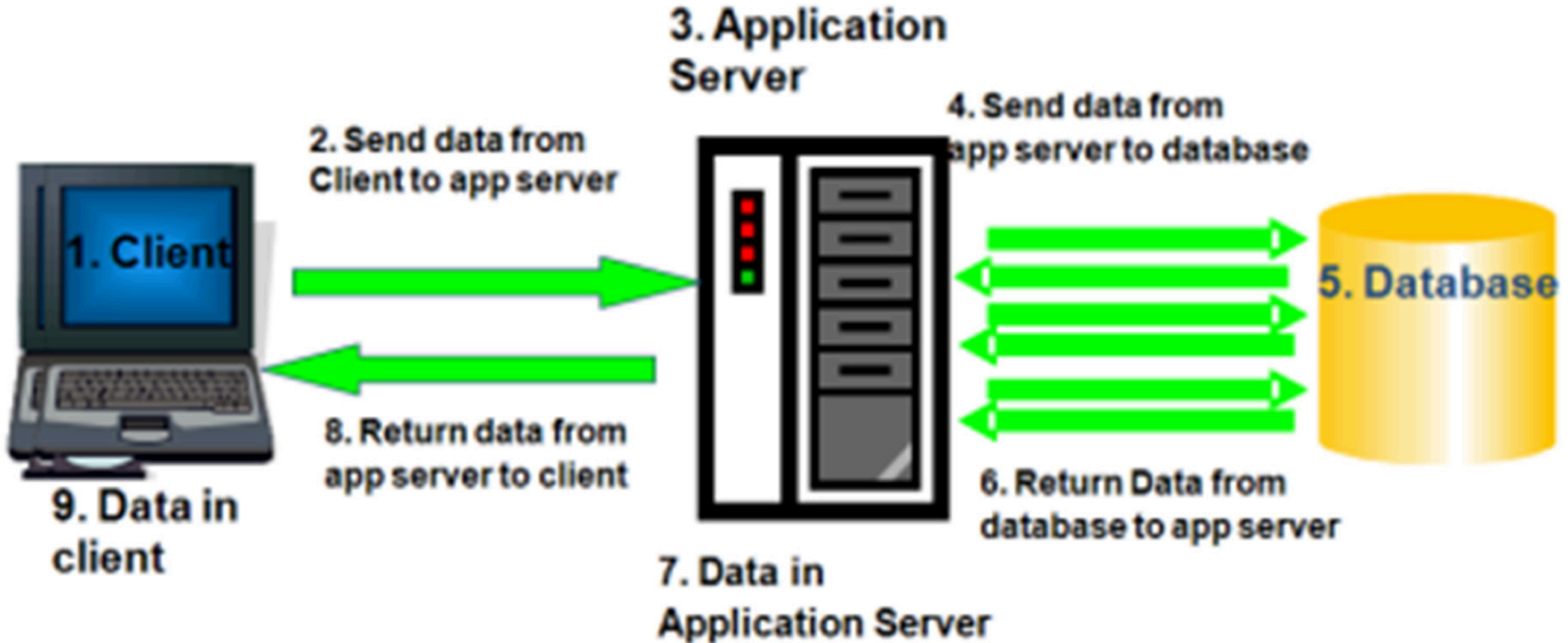
Agenda

- The universally accepted 50-year-old modular design principles – and why do many (most?) Oracle customers refuse to follow?
- Objection #1 demolished by EBR
- Objection #2 demolished by binding to PL/SQL datatypes from the client
- Objection #3 demolished ‘cos each developer can self-provision his own PDB

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The big picture



 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Let's start by thinking about the big picture of applications that use Oracle Database to implement their database of record and, in that context, how to understand the optimal top-down architecture of such an application.

Using PL/SQL, rather than avoiding it altogether, brings a significant performance benefit. Only when the case for using PL/SQL is time to talk about its details. PL/SQL's purpose in the optimal architecture is to issue SQL statements and to deal with the results, and in that sense, SQL statements can be seen as a special kind of subprogram within the closure of subprograms that implement the net effect of entry into PL/SQL. Therefore, the division of labor between the PL/SQL and SQL subsystems in the database, is the most critical determinant of overall PL/SQL performance. The stage-setting, and the interplay between PL/SQL and SQL, are addressed in the book's first two sections.

It would be meaningless to try to improve the performance of an incorrect application! Correctness must first be established; and only then, may performance be considered.

I'll take the opportunity, here, to specialize software engineering's central, generic principle for maximizing the chance of application correctness to the problem domain that this book addresses. By a very happy coincidence, it turns out that we can both have our cake and eat it: the architectural approach that maximizes the chance of application correctness is the same one that brings optimal performance.

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Few would deny that the correct implementation of a large software system depends upon good modular design. A module is a unit of code organization that implements a coherent subset of the system's functionality and that exposes this via an API that directly expresses this, and only this, functionality, and that hides all the implementation details behind this API.

Of course, this principle of modular decomposition is applied recursively: the system is broken into a small number of major modules, each of these is further broken down, and so on. This principle is arguably the most important one among the legion best practice principles for software engineering — and it has been commonly regarded as such for at least the past fifty years.

These days, an application that uses Oracle Database as its persistence mechanism is decomposed at the coarsest level into the database module, the application server module, and the client module.

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The ultimate implementation of the database module is the SQL statements that query from, and make changes to, the content of the application's tables. However, very commonly, an operation upon a single table implements just part of what, in the application's functional specification, is characterized as a business transaction.

The canonical example is the transfer funds business function within the scope of all the accounts managed by an application for a particular bank. This function is parameterized primarily by identifying the source account, the target account, and the cash amount; other parameters, like the date on which the transaction is to be made, and a transaction memo, are sometimes required.

This immediately suggests this API:

```
function Transfer_Funds(Source in..., Target in..., Amount in..., ...)
  return Outcome_t is...
```

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The API is specified as a function to reflect the possibility that the attempt may be disallowed, and the return datatype is nonscalar to reflect the fact that the reason that the attempt is disallowed might be characterized by several values, including, for example, the shortfall amount in the source account.

We can see immediately that there are several different design choices. For example, there might be a separate table for each kind of account, reflecting the fact that different kinds of account have different account details; or there might be a single table, with an account kind column, together with a family of per-account-kind details tables. There will similarly be a representation of account holders, and again these might have various kinds, like personal and corporate, with different details. There will doubtless be a table to hold requests for transfers that are due to be enacted in the future.

The point is obvious: a single API design that exactly reflects the functional specification may be implemented in many different ways. The conclusion is equally obvious:

The database module should be exposed by a PL/SQL API.
And the details of the names and structures of the tables,
and the SQL that manipulates them,
should be securely hidden from the application server module.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This paradigm is sometimes known as “thick database”. It sets the context for the discussion of when to use SQL and when to use PL/SQL. The only kind of SQL statement that the application server may issue is a PL/SQL anonymous block that invokes one of the API’s subprograms:

```
begin :r := Transfer_Funds(:s, :t, :a, ...); end;
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As a bonus, PL/SQL is better suited to the task of executing SQL statements and processing their results than any other programming language that can do this. For example:

- It supports embedded SQL as an intrinsic part of the definition of the syntax and the semantics of the language.
- The fact that a PL/SQL identifier can be used in embedded SQL where ordinary SQL would use a placeholder not only frees the programmer from the chore of writing code to bind to placeholders programatically, but also guarantees that the code is not vulnerable to SQL injection.
- The ability to anchor declarations of PL/SQL variables and types to the datatypes of columns in schema-level tables (%type, %rowtype, and the iterator in a cursor for loop) means that PL/SQL programs automatically adjust themselves to changes in the definitions of the tables they manipulate. PL/SQL programs that make only static references to other PL/SQL programs, tables, views, and so on are guaranteed to execute using only the latest definitions of what they depend upon.
- The ability to control which users can perform which business functions by selectively granting the Execute privilege on specified PL/SQL subprograms, rather than controlling which users can see and change data in which individual tables, by granting the Select, Insert, Update, and Delete privileges, is key to protecting the integrity of data.
- The fact that PL/SQL has intrinsic exception handling and that SQL errors (the notorious ORA-nnnnn) are mapped to PL/SQL exceptions, together with the fact the commit and rollback SQL statements are supported as embedded SQL statements in PL/SQL, guarantees the atomicity of business functions that change more than one table. This is another key factor in protecting data integrity.
- PL/SQL's SQL processing is optimally performant, not only because SQL executes in the same server process as the PL/SQL that issues it, but also because of various under-the-covers optimizations like the famous soft-parse avoidance.

Of course, it is no coincidence that PL/SQL uniquely has these properties. They were defined specifically as the requirements, at the time of its invention, that the language should meet.

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

I know of many customers who strictly adhere to the thick database paradigm; and I know of many who do not — to the extent that all calls to the database are implemented as SQL statements that explicitly manipulate the application's tables. This, of course, makes the database unworthy of the term "module"!

Customers in the first group seem generally to be very happy with the performance and maintainability of their applications.

Ironically, customers in the second group routinely complain of performance problems (because the execution of a single business transaction often involves many round trips from the application server module to the database module). And they complain of maintenance problems (because even small patches to the application imply changes both to the implementation of the database module and to the implementation of the application sever module).

I am convinced that an application that uses Oracle Database as its persistence mechanism has no special properties that recommend that its design, uniquely among an uncountable number of diverse software systems, should disregard otherwise universally respected wisdom.

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Modular software principles applied to applications that use Oracle Database

- Fifty-year-old wisdom instructs us to expose the database to client-side code as a PL/SQL API:
  - securely hide the implementation details:
    - *the names and structures of the tables*
    - *and the SQL statements that manipulate them*

  from the client.

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Modular software principles with an Oracle Database conference slant

- *Everyone at every OUG-style conference:* bind to placeholders; avoid literals (performance and injection-proofing)
- *Cary Millsap (and legion others):* one client/server round trip rather than many
- *Mark Farnham:* avoid deadlocks by updating the tables in a FK-PK related set in the proper, and fixed, order
- *Tom Kyte:* avoid implicit conversions when binding to SQL

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Modular software principles with an Oracle Database conference slant – cont

- *Tom Kyte:* never (think that you can) enforce data rules using client-side code
- *Tom (and many others):* use SQL in preference to procedural code, and write the proper SQL…
- *Jeff Jacobs (and many others):* I can't stop a Java developer writing bad SQL

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Why don’t the unhappy customers do what’s good for them?

- I hear four reasons
  - “PL/SQL” doesn’t start with the letter “J”
  - I can’t change PL/SQL code in my production app without causing lots of downtime
  - I can’t call PL/SQL procedures with IBPI’s of records from Java and other clients
  - I can’t give each developer his own sandbox database to work in

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle Database 12*c* brings changes that demolish each of these objections

- EBR is enhanced to make adopting it a no-brainer
- PL/SQL is enhanced to allow subprograms with formals declared using PL/SQL-only datatypes (IBBI of records)
- Oracle Multitenant allows a PDB to be created (new or as a clone) very quickly – or even instantaneously using SLQ statements. Building a self-provisioning mini-app is trivial

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Agenda

- Why do many (most?) Oracle customers refuse to follow universally accepted 50-year-old modular design principles?
- **Objection #1 demolished by EBR**
- Objection #2 demolished by binding to PL/SQL datatypes from the client
- Objection #3 demolished ‘cos each developer can self-provision his own PDB

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Quiz

- Who does scheduled backups too?
- Who has Data Guard?
- Who has used Oracle Active Data Guard Rolling Upgrade?
- Who has used Rolling RAC Upgrade?
- Who has used edition-based redefinition (EBR)?

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The EBR page on OTN

- `http://www.oracle.com/technetwork/database/features/availability/ebr-455513.html`

ORACLE DATABASE 12c **Edition-Based Redefinition**

Edition-based redefinition (EBR), enables online application upgrade with uninterrupted availability of the application. When the installation of an upgrade is complete, the pre-upgrade application and the post-upgrade application are able to be used at the same time. Therefore an existing session can continue to use the pre-upgrade application until its user decides to end it; and all new sessions can use the post-upgrade application. When there are no longer any sessions using the pre-upgrade application, it can be retired. When used in this manner, EBR enables hot rollover from the pre-upgrade version to the post-upgrade version, with zero downtime.

EBR is an included feature of Oracle Database 12c that enables online application upgrades in the following manner:

- Code changes are installed in the privacy of a new *edition*.
- Data changes are made safely by writing only to new columns or new tables not seen by the old edition. An *editioning view* exposes a different projection of a table into each edition to allow each to see just its own columns..
- A *crossedition trigger* propagates data changes made by the old edition into the new edition's columns, or (in hot-rollover) vice-versa..

Additional technical information on EBR includes:

- Technical White Paper
- Recorded Overview Presentation (download)
- Documentation
- Tutorial
- Self-contained EBR exercise

- Technical White Paper
- Recorded Overview Presentation (download)
- Documentation
- Tutorial
- Self-contained EBR exercise

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Lightening review of the “classic” conference presentation on EBR

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Editioned and noneditioned objects

- An object whose type is noneditionable is never editioned
- An object whose type is editionable is editioned only when you request it for that object (requires that the owner is *editions-enabled*)
- *Theorem:* a noneditioned object cannot *ordinarily* depend on an editioned object
  - For example, a table cannot depend on an editioned UDT
  - If you want to use a type as the datatype for a column, that UDT must not be editioned

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Improvements to EBR in Oracle Database 12*c*

- The granularity of the editioned state of an object
  - In 11.2, the granularity is the whole schema
  - From 12.1, the granularity is the indvidual object
- A materialized view or an index on a virtual column is allowed to depend on an editioned PL/SQL function or an editioned view

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Public synonyms for customer editioned objects

- Notice that a public synonym is no more than an ordinary synonym that happens to have the owner *public*
- In 11.2, you couldn't editions-enable *public*, so a public synonym couldn't denote (and therefore depend upon) an editioned object. This caused a noticeable adoption barrier for EBR
- In 12.1, *public* is always editions-enabled – but the Oracle-maintained public synonyms are not editioned. (If you did make any of them editioned, it would cause havoc.)
- You *can* safely make public synonyms that denote your own editions objects editioned

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Tables with UDT columns

- An ordinary (as opposed to virtual) column cannot specify the *evaluation edition* or *edition range* metadata
- Therefore, a UDT that defines the datatype for a table column must remain noneditioned.
- In an EBR exercise, if the aim is to redefine the UDT, then the “classic” replacement column paradigm is used
    - A spec doesn’t depend on its body. So the body of an ADT, where the code is, can be editioned. (The appropriate one is found at run time.)

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Materialized views and indexes on virtual columns

- Objects of these kinds have metadata that is explicitly set by the *create* and *alter* statements
  - the *evaluation edition* explicitly specifies the name of the edition in which the resolution of editioned names, within the closure of the object's dependency parents (at compile time), and those objects that are identified during SQL execution (at run time)
  - The *edition range* explicitly specifies the set of adjacent edtitions in which the optimizer will consider the object when computing the execution plan

ORACLE

 | Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Agenda

- Why do many (most?) Oracle customers refuse to follow universally accepted 50-year-old modular design principles?
- Objection #1 demolished by EBR
- **Objection #2 demolished by binding to PL/SQL datatypes from the client**
- Objection #3 demolished ‘cos each developer can self-provision his own PDB

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Binding values of PL/SQL-only datatypes into SQL statements

- Before 12.1, you could bind only values of SQL datatypes
- In 12.1, you can bind PL/SQL index-by-pls_integer tables (of records) and booleans
  - **from** client-side programs – OCI or both flavors of JDBC – and from PL/SQL
  - **to** anonymous blocks, statements using functions, or statements using the *table* operator

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Binding a PL/SQL index-by table to SQL

- Before 12.1, you could select from a collection, but
  - The type had to be defined at schema-level
  - Therefore it had to be a nested table or a varray
  - A non-scalar payload had to be an ADT
- New in 12.1
  - The type can be defined in a package spec – can be *index by pls_integer* table
  - The payload can be a record – but the fields must still be SQL datatypes

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The collection

```
package Pkg authid Definer is
  type r is record(n integer, v varchar2(10));
  type t is table of r index by pls_integer;
  x t;
end Pkg;
```


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.
ORACLE

Example 1
binding an IBPI to a PL/SQL function in SQL

```
function f(x in Pkg.t) return varchar2 authid Definer is
  r varchar2(80);
begin
  for j in 1..x.Count() loop
    r := r||...;
  end loop;
  return r;
end f;
```

```
procedure Bind_IBPI_To_Fn_In_SQL authid Definer is
  v varchar2(80);
begin
  select f(Pkg.x) into v from Dual;
  ...
  execute immediate 'select f(:b) from Dual' into v
    using Pkg.x;
end Bind_IBPI_To_Fn_In_SQL;
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Example 2
using with the *table* operator

```
procedure Select_From_IBPI authid Definer is
  y Pkg.t;
begin
  for j in (select n, v from table(Pkg.x)) loop
    ...
  end loop;

  execute immediate 'select n, v from table(:b)'
  bulk collect into y
  using Pkg.x;
  for j in 1..y.Count() loop
    ...
  end loop;
end Select_From_IBPI;
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Example 3
binding an IBPI to an anonymous block

```
procedure p1(x in Pkg.t) authid Definer is
begin
  for j in 1..x.Count() loop
    ...;
  end loop;
end p1;
```

```
procedure Bind_IBPI_To_Anon_Block authid Definer is
begin
  execute immediate 'begin p1(:b); end;' using Pkg.x;
end Bind_IBPI_To_Anon_Block;
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Example 4
binding a boolean to an anonymous block

```
procedure p2(b in boolean) authid Definer is
begin
  DBMS_Output.Put_Line(case b
                         when true  then 'True'
                         when false then 'False'
                         else            'Null'
                       end);
end p2;
```

```
procedure Bind_Boolean_To_Anon_Block authid Definer is
  Nil constant boolean := null;
begin
  execute immediate 'begin p2(:b); end;' using true;
  execute immediate 'begin p2(:b); end;' using false;
  execute immediate 'begin p2(:b); end;' using Nil;
end Bind_Boolean_To_Anon_Block;
```

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Improved support for binding PL/SQL types in JDBC

- Before 12.1
  - Generate a schema level object type to mirror the structure of the non-SQL package type
  - Populate and bind the object into a custom PL/SQL wrapper around the desired PL/SQL subprogram
  - Convert the object to the package type in the wrapper and call the PL/SQL subprogram with the package type

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Improved support for binding PL/SQL types in JDBC

- New in 12.1
  - PL/SQL package types supported as binds in JDBC
  - Can now execute PL/SQL subprograms with non-SQL types
  - Supported types include *records*, *index-by tables*, *nested tables* and *varrays*
  - *Table%rowtype*, *view%rowtype* and package defined *cursor%rowtype* also supported. They're technically record types

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Example1: Bind a single record from Java into a PL/SQL procedure, modify it, and bind it back out to Java

```
package Emp_Info is
  type employee is record(First_Name   Employees.First_Name%type,
                          Last_Name    Employees.Last_Name%type,
                          Employee_Id  Employees.Employee_Id%type,
                          Is_CEO       boolean);

  procedure Get_Emp_Name(Emp_p in out Employee);
end;
```


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.
ORACLE

Example1:

- Use the *EmpinfoEmployee* class, generated by JPub, to implement the *Employee* formal parameter

```
{   …
      EmpinfoEmployee Employee = new EmpinfoEmployee();
      Employee.setEmployeeId(new java.math.BigDecimal(100));  // Use Employee ID 100

      // Call Get_Emp_Name() with the Employee object
      OracleCallableStatement cstmt =
         (OracleCallableStatement)conn.prepareCall("call EmpInfo.Get_Emp_Name(?)");
      cstmt.setObject(1, Employee, OracleTypes.STRUCT);

      // Use "PACKAGE.TYPE NAME" as the type name
      cstmt.registerOutParameter(1, OracleTypes.STRUCT, "EMPINFO.EMPLOYEE");
      cstmt.execute();

      // Get and print the contents of the Employee object
      EmpinfoEmployee oraData =
           (EmpinfoEmployee)cstmt.getORAData(1, EmpinfoEmployee.getORADataFactory());
      System.out.println("Employee: " + oraData.getFirstName() + " " + oraData.getLastName());
      System.out.println("Is the CEO? " + oraData.getIsceo());
}
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Example 2: populate a collection of *table%rowtype* using a bulk collect statement, and pass the collection as an *out* parameter back to the caller

```
package EmpRow is
  type Table_of_Emp is table of Employees%Rowtype;
  procedure GetEmps(Out_Rows out Table_of_Emp);
end;
```

```
package Body EmpRow is
  procedure GetEmps(Out_Rows out Table_of_Emp) is
  begin
    select *
    bulk collect into Out_Rows
    from Employees;
  end;
end;
```

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Example 2:

```
{ …
   // Call GetEmps() to get the ARRAY of table row data objects
   CallableStatement cstmt = conn.prepareCall("call EmpRow.GetEmps(?)");

    // Use "PACKAGE.COLLECTION NAME" as the type name
   cstmt.registerOutParameter(1, OracleTypes.ARRAY, "EMPROW.TABLE_OF_EMP");
   cstmt.execute();

   // Print the Employee Table rows
   Array a = cstmt.getArray(1);
   String s = Debug.printArray ((ARRAY)a, "",
                              ((ARRAY)a).getSQLTypeName () +"( ", conn);
   System.out.println(s);
}
```


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.
ORACLE

Agenda

- Why do many (most?) Oracle customers refuse to follow universally accepted 50-year-old modular design principles?
- Objection #1 demolished by EBR
- Objection #2 demolished by binding to PL/SQL datatypes from the client
- **Objection #3 demolished ‘cos each developer can self-provision his own PDB**

ORACLE

 | Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The macroscopic provisioning operations

- Create PDB
- Clone PDB
- Unplug PDB
- Plug in PDB
- Drop PDB
- Also, set the mode
  - read-write, read-only, mounted | restricted?

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The SQL statements

```
create pluggable database... admin user... identified by...

create pluggable database... from...

alter pluggable database... unplug into...

-- "as clone" is incompatible with "move"
create pluggable database... [as clone] using... [copy/move]

-- "keep" makes sense only after "unplug"
drop pluggable database... [including/keep] datafiles

alter pluggable database... close [immediate] [force]

alter pluggable database... open [read only/read write] [restricted]
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Raw PL/SQL encapsulation: clone a PDB

```
declare
  Source_Open_Mode Sys.v_$PDBs.Open_Mode%type not null := '?';
begin
  select a.Open_Mode
  into Source_Open_Mode
  from Sys.v_$PDBs a
  where a.Name = 'PDB3';

  if Source_Open_Mode = 'READ WRITE' then
    execute immediate 'alter pluggable database PDB3 open read only force';
  end if;

  execute immediate 'create pluggable database PDB4 from PDB3';

  if Source_Open_Mode = 'READ WRITE' then
    execute immediate 'alter pluggable database PDB3 open read write force';
  end if;

  execute immediate 'alter pluggable database PDB4 open read write';
end;
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The PL/SQL API – core

```
procedure Sys.Create_PDB(
  PDB_Name in varchar2)

procedure Sys.Clone_PDB(
  Source_PDB_Name in varchar2, Clone_PDB_Name in varchar2)

procedure Sys.Unplug_PDB(
   PDB_Name in varchar2, Manifest_Dir in varchar := null)

procedure Sys.Plug_In_PDB(
   PDB_Name in varchar2, Manifest_Dir in varchar2 := null)

procedure Sys.Drop_PDB_Incl_Files(
   PDB_Name in varchar2)
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The API – extras

```
function Check_Plug_Compatibility(
  PDB_Name in varchar2, Manifest_Dir in varchar2 := null)
  return varchar2


view PDBs(Name, Open_Mode, Res, Status, Create_Scn)
```

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Putting the PL/SQL API through its paces

```
procedure Demo_Plsql_Provisioning_API authid Definer is
begin

  Create_PDB@c##Provisioner_At_cdb2('pdb1');

  Clone_PDB@c##Provisioner_At_cdb2('pdb1', 'pdb2');

  Unplug_PDB@c##Provisioner_At_cdb2('pdb2');

  Drop_PDB_Incl_Files@c##Provisioner_At_cdb2('pdb1');

  Plug_In_PDB@c##Provisioner_At_cdb1('pdb2');

  Drop_PDB_Incl_Files@c##Provisioner_At_cdb1('pdb2');

end Demo_Plsql_Provisioning_API;
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Agenda

- Summary

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle Database 12*c* brings these enhancements

- *Edition-based redefinition*
  - The granularity of the editioned state of the name of a PL/SQL unit, a view, or a synonym is now the single occurring name. Materialized views and virtual columns have new metadata to specify the edition to be used to resolve the name of an editioned dependency parent. They also have new metadata to specify the range of editions within which the optimizer may consider the object.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle Database 12*c* brings these enhancements

- *PL/SQL:*
  - Values of non-SQL datatypes can be bound to the formal parameters of database PL/SQL subprograms invoked from the client. In particular, row sets can now be passed between the client and the database using the natural datatype: an index-by-PL/SQL-table of records.

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle Database 12*c* brings these enhancements

- *The multitenant architecture:*
  - The *clone PDB* operation, taking advantage of the snapshot facility in the underlying filesystem, and the *drop PDB* operation are exposed as SQL statements. This makes it very straightforward to write a PL/SQL application to allow developers to rapidly, and thinly, self-provision a private database environment in which to change and test their checked out code.

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Conclusion

- There is now no excuse for violating the best practice principle that is already universally followed in the majority of successful software projects. Now you can confidently expose the Oracle Database using a PL/SQL API — hiding all the details of tables and SQL statements behind this API — knowing that this approach brings only benefits.

ORACLE

 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The PL/SQL page on OTN

- `http://www.oracle.com/technetwork/database/features/plsql/index.html`

PL/SQL

PL/SQL is an imperative 3GL that was designed specifically for the seamless processing of SQL commands. It provides specific syntax for this purpose and supports exactly the same datatypes as SQL. Server-side PL/SQL is stored and compiled in Oracle Database and runs within the Oracle executable. It automatically inherits the robustness, security, and portability of Oracle Database.

Technical Information

- With Oracle Database 12*c*, There is All the More Reason to Use Database PL/SQL
- Doing SQL from PL/SQL: Best and Worst Practices
- How to write SQL injection-proof PL/SQL
- PL/SQL Enhancements in Oracle Database 11*g*

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Hardware and Software

ORACLE®

Engineered to Work Together

ORACLE®