+ ENTRENAMIENTO
EXPERIENCIA

GALAXY
TRAINING

ORACLE DATABASE 12c ONWARDS DEVELOPER: PL/SQL

Ing. Cesar Hijar
Instructor

chijar@galaxy.edu.pe

AGENDA

Programación de objetos II

- Manejo de excepciones
- Procedimientos, funciones y paquetes
- Usando SUPPLIED PACKAGES en el desarrollo de aplicaciones
- Consideraciones en la creación de código PL/SQL

MANEJO DE EXCEPCIONES

MANEJO DE EXCEPCIONES

What Is an Exception?

```
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname
  FROM employees
  WHERE first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is :' ||v_lname);
END;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
Error starting at line 3 in command:
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname FROM employees WHERE
  first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is : ' ||v_lname);
END;
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
01422. 00000 -  'exact fetch returns more than requested number of rows'
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

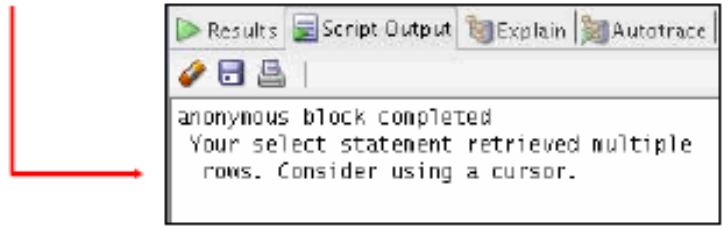
Errores en tiempo de ejecución.

MANEJO DE EXCEPCIONES

… entonces →

Handling the Exception: An Example

```
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname
  FROM employees
  WHERE first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is :' ||v_lname);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
  DBMS_OUTPUT.PUT_LINE (' Your select statement retrieved
   multiple rows. Consider using a cursor.');
END;
/
```



MANEJO DE EXCEPCIONES

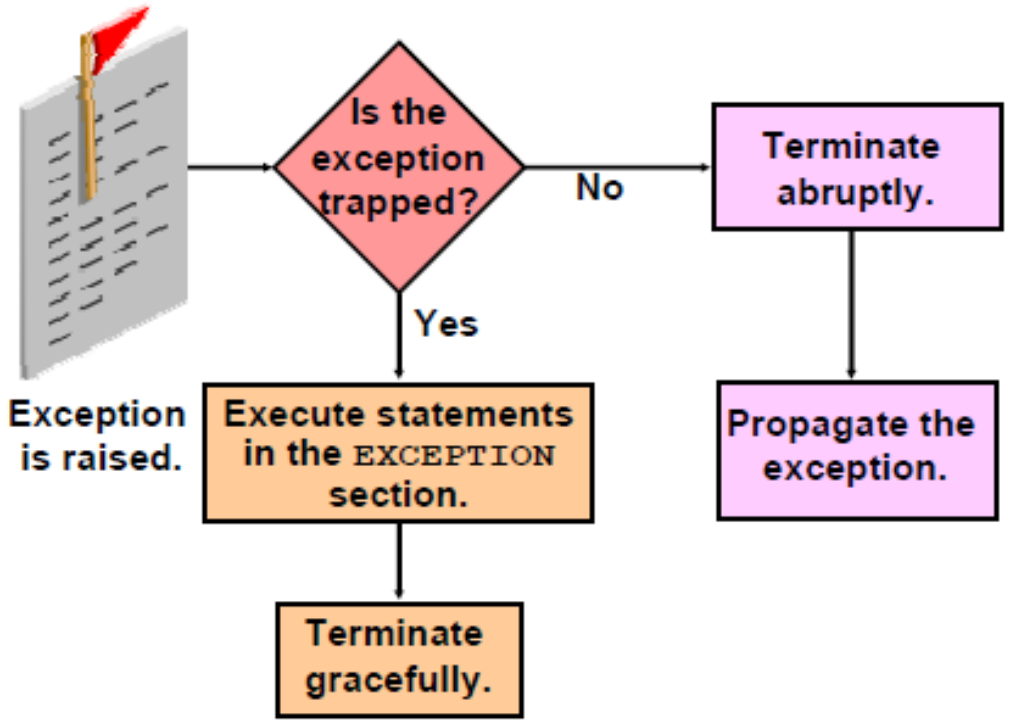
Understanding Exceptions with PL/SQL

- An exception is a PL/SQL error that is raised during program execution.
- An exception can be raised:
  - Implicitly by the Oracle Server
  - Explicitly by the program
- An exception can be handled:
  - By trapping it with a handler
  - By propagating it to the calling environment

Handled:
Haciendo código que se ejecute en el bloque EXCEPTION

MANEJO DE EXCEPCIONES

Handling Exceptions



Exception Types

There are three types of exceptions.

Exception	Description	Directions for Handling
Predefined Oracle Server error	One of approximately 20 errors that occur most often in PL/SQL code	You need not declare these exceptions. They are predefined by the Oracle server and are raised implicitly.
Non-predefined Oracle Server error	Any other standard Oracle Server error	You need to declare these within the declarative section; the Oracle server raises the error implicitly, and you can catch the error in the exception handler.
User-defined error	A condition that the developer determines is abnormal	You need to declare in the declarative section and raise explicitly.

Note: Some application tools with client-side PL/SQL (such as Oracle Developer Forms) have their own exceptions.

Syntax to Trap Exceptions

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

Guidelines for Trapping Exceptions

- The `EXCEPTION` keyword starts the exception-handling section.
- Several exception handlers are allowed.
- Only one handler is processed before leaving the block.
- `WHEN OTHERS` is the last clause.

Trapping Predefined Oracle Server Errors

- Reference the predefined name in the exception-handling routine.
- Sample predefined exceptions:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - INVALID_CURSOR
  - ZERO_DIVIDE
  - DUP_VAL_ON_INDEX

MANEJO DE EXCEPCIONES

Predefined Exceptions

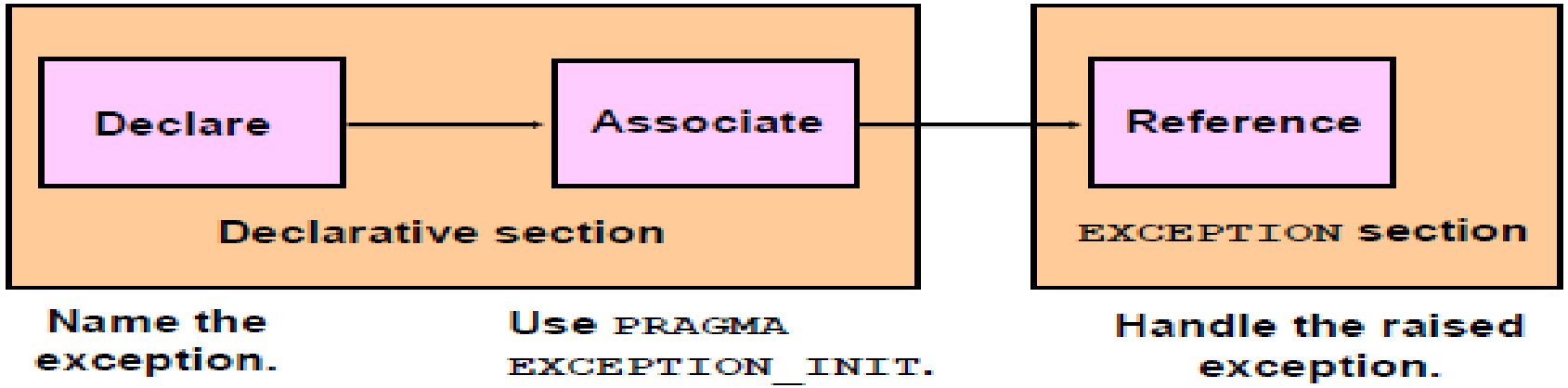
Exception Name	Oracle Server Error Number	Description
ACCESS_INTO_NULL	ORA-06530	Attempted to assign values to the attributes of an uninitialized object
CASE_NOT_FOUND	ORA-06592	None of the choices in the WHEN clauses of a CASE statement are selected, and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	Attempted to apply collection methods other than EXISTS to an uninitialized nested table or VARRAY
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already open cursor
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred.
INVALID_NUMBER	ORA-01722	Conversion of character string to number failed.
LOGIN_DENIED	ORA-01017	Logging on to the Oracle server with an invalid username or password
NO_DATA_FOUND	ORA-01403	Single row SELECT returned no data.
NOT_LOGGED_ON	ORA-01012	The PL/SQL program issues a database call without being connected to the Oracle server.
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	ORA-06504	The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types.

MANEJO DE EXCEPCIONES

Predefined Exceptions (continued)

Exception Name	Oracle Server Error Number	Description
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory, or memory is corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Referenced a nested table or VARRAY element by using an index number larger than the number of elements in the collection
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Referenced a nested table or VARRAY element by using an index number that is outside the legal range (for example, –1)
SYS_INVALID_ROWID	ORA-01410	The conversion of a character string into a universal ROWID fails because the character string does not represent a valid ROWID.
TIMEOUT_ON_RESOURCE	ORA-00051	Time-out occurred while the Oracle server was waiting for a resource.
TOO_MANY_ROWS	ORA-01422	Single-row SELECT returned multiple rows.
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size-constraint error occurred.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero

Trapping Non-Predefined Oracle Server Errors



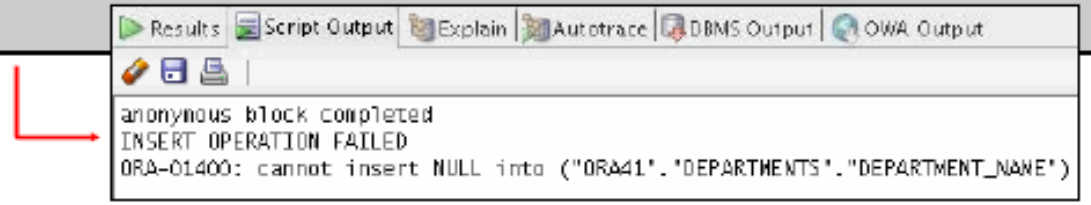
Trapping Non-Predefined Oracle Server Errors

Non-predefined exceptions are similar to predefined exceptions; however, they are not defined as PL/SQL exceptions in the Oracle Server. They are standard Oracle errors. You create exceptions with standard Oracle errors by using the PRAGMA EXCEPTION_INIT function. Such exceptions are called non-predefined exceptions.

Non-Predefined Error Trapping: Example

To trap Oracle Server error 01400 ("cannot insert NULL"):

```
DECLARE
 e_insert_excep EXCEPTION;                          1
 PRAGMA EXCEPTION_INIT(e_insert_excep, -01400);     2
BEGIN
 INSERT INTO departments
 (department_id, department_name) VALUES (280, NULL);
EXCEPTION                                           3
 WHEN e_insert_excep THEN
   DBMS_OUTPUT.PUT_LINE('INSERT OPERATION FAILED');
   DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```



The example illustrates the three steps associated with trapping a non-predefined error:

1. Declare the name of the exception in the declarative section, using the syntax:
   exception EXCEPTION;
   In the syntax, *exception* is the name of the exception.
2. Associate the declared exception with the standard Oracle Server error number by using the PRAGMA EXCEPTION_INIT function. Use the following syntax:
   PRAGMA EXCEPTION_INIT(*exception*, *error_number*);
   In the syntax, *exception* is the previously declared exception and *error_number* is a standard Oracle Server error number.
3. Reference the declared exception within the corresponding exception-handling routine.

MANEJO DE EXCEPCIONES

Functions for Trapping Exceptions

- SQLCODE: Returns the numeric value for the error code
- SQLERRM: Returns the message associated with the error number

Functions for Trapping Exceptions

When an exception occurs, you can identify the associated error code or error message by using two functions. Based on the values of the code or the message, you can decide which subsequent actions to take.

SQLCODE returns the Oracle error number for internal exceptions. SQLERRM returns the message associated with the error number.

Function	Description
SQLCODE	Returns the numeric value for the error code (You can assign it to a NUMBER variable.)
SQLERRM	Returns character data containing the message associated with the error number

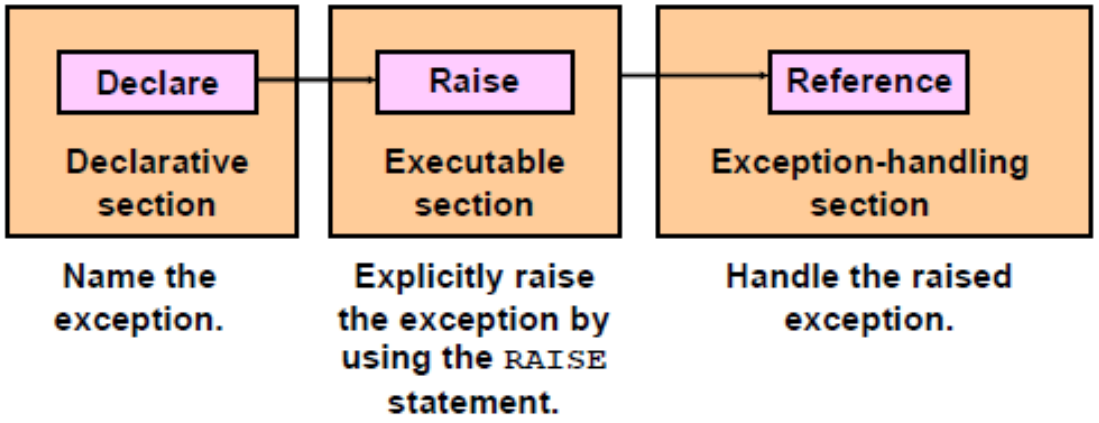
SQLCODE Values: Examples

SQLCODE Value	Description
0	No exception encountered
1	User-defined exception
+100	NO_DATA_FOUND exception
negative number	Another Oracle server error number

Functions for Trapping Exceptions

```
DECLARE
  error_code      NUMBER;
  error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    error_code := SQLCODE ;
    error_message := SQLERRM ;
   INSERT INTO errors (e_user, e_date, error_code,
   error_message) VALUES(USER,SYSDATE,error_code,
   error_message);
END;
/
```

Trapping User-Defined Exceptions



PL/SQL exceptions must be:

- Declared in the declarative section of a PL/SQL block
- Raised explicitly with `RAISE` statements
- Handled in the `EXCEPTION` section

MANEJO DE EXCEPCIONES

Trapping User-Defined Exceptions

```
DECLARE
  v_deptno NUMBER := 500;
  v_name VARCHAR2(20) := 'Testing';
  e_invalid_department EXCEPTION;          <-- 1
BEGIN
  UPDATE departments
  SET department_name = v_name
  WHERE department_id = v_deptno;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_department;            <-- 2
  END IF;
  COMMIT;
EXCEPTION                                  <-- 3
WHEN e_invalid_department THEN
  DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
/
```

Results | Script Output | Explain

```
anonymous block completed
No such department id.
```

You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name of the user-defined exception within the declarative section.
   Syntax:
   ```
   exception  EXCEPTION;
   ```
   In the syntax, *exception* is the name of the exception.
2. Use the RAISE statement to raise the exception explicitly within the executable section.
   Syntax:
   ```
   RAISE exception;
   ```
   In the syntax, *exception* is the previously declared exception.
3. Reference the declared exception within the corresponding exception-handling routine.

MANEJO DE EXCEPCIONES

RAISE_APPLICATION_ERROR Procedure

Syntax:

```
raise_application_error (error_number,
                         message[, {TRUE | FALSE}]);
```

- You can use this procedure to issue user-defined error messages from stored subprograms.
- You can report errors to your application and avoid returning unhandled exceptions.

In the syntax:

error_number	Is a user-specified number for the exception between –20,000 and –20,999
message	Is the user-specified message for the exception; is a character string up to 2,048 bytes long
TRUE | FALSE	Is an optional Boolean parameter (If TRUE, the error is placed on the stack of previous errors. If FALSE, which is the default, the error replaces all previous errors.)

RAISE_APPLICATION_ERROR Procedure

- Is used in two different places:
  - Executable section
  - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle Server errors

RAISE_APPLICATION_ERROR Procedure

Executable section:

```
BEGIN
...
  DELETE FROM employees
     WHERE  manager_id = v_mgr;
  IF SQL%NOTFOUND THEN
     RAISE_APPLICATION_ERROR(-20202,
       'This is not a valid manager');
  END IF;
   ...
```

Exception section:

```
...
EXCEPTION
   WHEN NO_DATA_FOUND THEN
     RAISE_APPLICATION_ERROR (-20201,
        'Manager is not a valid employee.');
END;
/
```

Quiz

You can trap any error by including a corresponding handler within the exception-handling section of the PL/SQL block.

1. True
2. False

PROCEDIMIENTOS, FUNCIONES Y PAQUETES

Procedures and Functions

- Are named PL/SQL blocks
- Are called PL/SQL subprograms
- Have block structures similar to anonymous blocks:
  - Optional declarative section (without the `DECLARE` keyword)
  - Mandatory executable section
  - Optional section to handle exceptions

Block Types

Subprograms

Anonymous

```
[DECLARE]

BEGIN
  --statements

[EXCEPTION]

END;
```

Procedure

```
PROCEDURE name
IS

BEGIN
  --statements

[EXCEPTION]

END;
```

Function

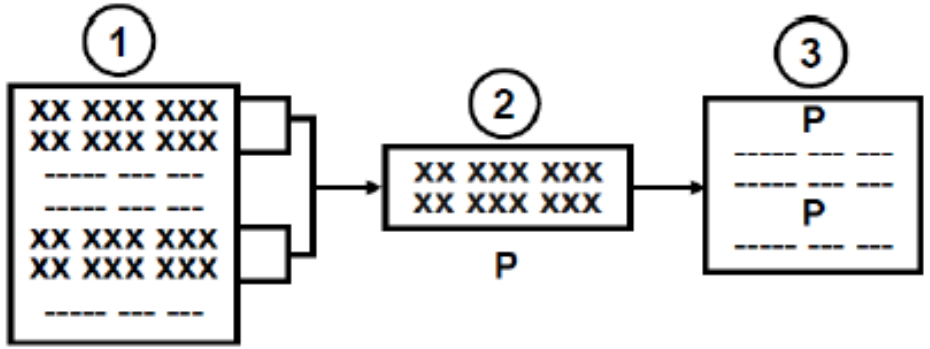
```
FUNCTION name
RETURN datatype
IS
BEGIN
  --statements
  RETURN value;
[EXCEPTION]

END;
```

Differences Between Anonymous Blocks and Subprograms

Anonymous Blocks	Subprograms
Unnamed PL/SQL blocks	Named PL/SQL blocks
Compiled every time	Compiled only once
Not stored in the database	Stored in the database
Cannot be invoked by other applications	Named and, therefore, can be invoked by other applications
Do not return values	If functions, must return values
Cannot take parameters	Can take parameters

Creating a Modularized Subprogram Design




```
update emp
set sal=sal+ 100
where emp_id =100;

update emp
set sal=sal+ 100
where emp_id =101;

update emp
set sal=sal+ 100
where emp_id =102;
```

```
procedure update_emp
(p_emp_id number, p_am number)
is
begin
 update emp
set sal=sal+ p_am
where emp_id=p_emp_id;
commit;
end;
```

```
procedure update_emp_group
is
...
....
update_emp( x , y);
...
....

....
end;
```

Modularize code into subprograms.

1. Locate code sequences repeated more than once.
2. Create subprogram P containing the repeated code
3. Modify original code to invoke the new subprogram.

Modularizing Development with PL/SQL Blocks

- PL/SQL is a block-structured language. The PL/SQL code block helps modularize code by using:
  - Anonymous blocks
  - Procedures and functions
  - Packages
  - Database triggers
- The benefits of using modular program constructs are:
  - Easy maintenance
  - Improved data security and integrity
  - Improved performance
  - Improved code clarity

So what is the benefits of anonymous block ?

- Writing trigger code for oracle forms components


```
Begin
P1;
P2;
End;
```

- Initiate calls for procedures , functions and packages
- Isolating exception handling within a block of code

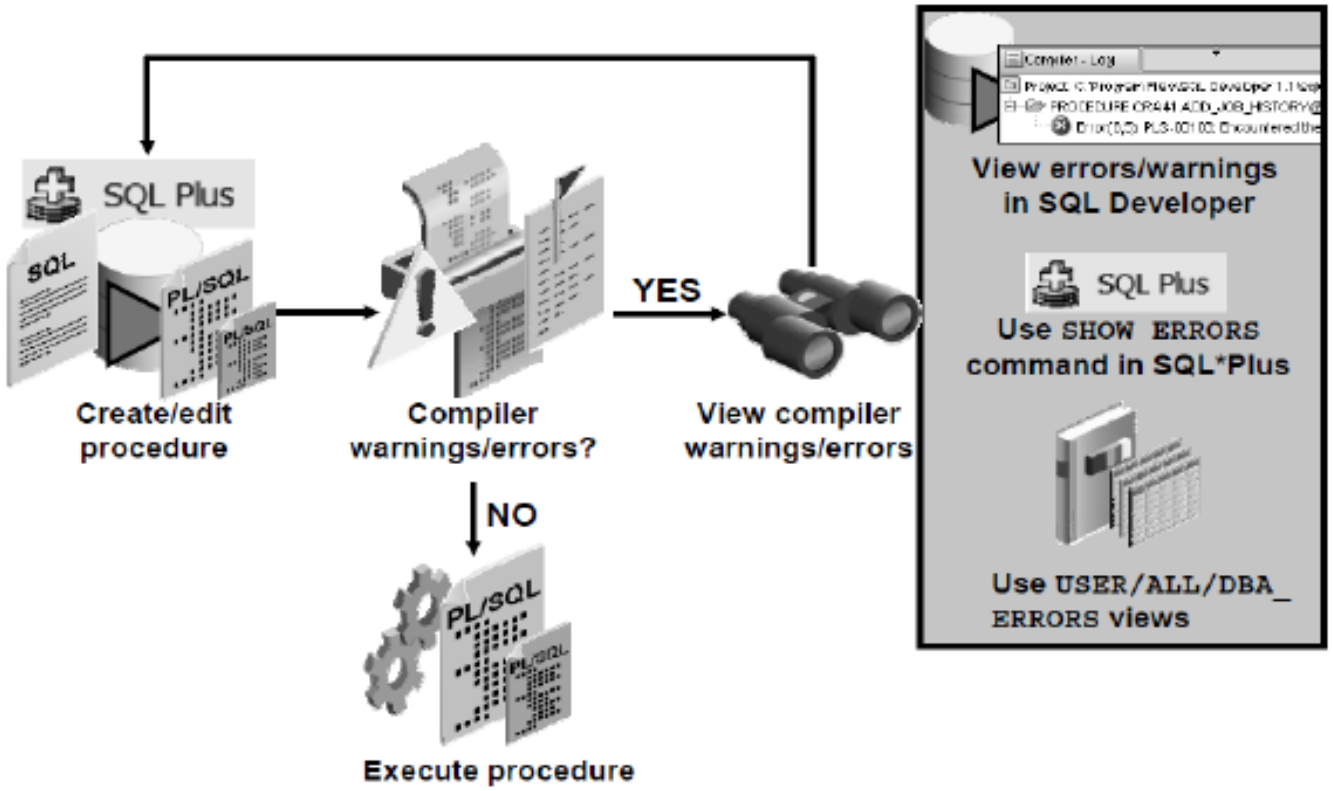
```
Procedure p1
Is
Begin
    begin
     --------
    Exception
    End;

End;
```

What Are PL/SQL Subprograms?

- A PL/SQL subprogram is a named PL/SQL block that can be called with a set of parameters.
- You can declare and define a subprogram within either a PL/SQL block or another subprogram.
- A subprogram consists of a specification and a body.
- A subprogram can be a procedure or a function.
- Typically, you use a procedure to perform an action and a function to compute and return a value.

Creating Procedures: Overview



Procedure: Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
IS|AS
procedure_body;
```

procedure_name	Is the name of the procedure to be created
argument	Is the name given to the procedure parameter. Every argument is associated with a mode and data type. You can have any number of arguments separated by commas.

mode	Mode of argument:
	`IN` (default)
	`OUT`
	`IN OUT`

datatype	Is the data type of the associated parameter. The data type of parameters cannot have explicit size; instead, use `%TYPE`.
Procedure_body	Is the PL/SQL block that makes up the code

Creating a Procedure

```
...
CREATE TABLE dept AS SELECT * FROM departments;
CREATE PROCEDURE add_dept IS
 v_dept_id dept.department_id%TYPE;
 v_dept_name dept.department_name%TYPE;
BEGIN
 v_dept_id:=280;
 v_dept_name:='ST-Curriculum';
 INSERT INTO dept(department_id,department_name)
 VALUES(v_dept_id,v_dept_name);
 DBMS_OUTPUT.PUT_LINE(' Inserted '|| SQL%ROWCOUNT
||' row ');
END;
```

PROCEDIMIENTOS, FUNCIONES Y PAQUETES

Create procedure statement

```
CREATE OR REPLACE PROCEDURE UPDATE_SAL
(P_EMP_ID IN NUMBER, P_AMOUNT IN NUMBER)
IS
BEGIN

  UPDATE employees
  set salary=salary+P_AMOUNT
  where employee_id=P_EMP_ID;
  Commit;
exception

WHEN OTHERS THEN

DBMS_OUTPUT.PUT_LINE (SQLCODE);
DBMS_OUTPUT.PUT_LINE (SQLERRM);

END;
```

Optional to overwide existing procedure

Optional parameters list

Parameter mode

Here you declare variables

Or

AS

- Parameters data types without size
- You should have create procedure privilege
- Substitution and host variables not allowed in procedures

1- you can call the procedure alone by this command

```
execute UPDATE_SAL (100,50);
```

2- you can call the procedure inside any PL/SQL block

```
Begin
...
UPDATE_SAL (100,50);
.....
End;
```

CONSULTANDO A LAS VISTAS DEL DICCIONARIO DE DATOS

```
SELECT object_name,object_type FROM
user_objects
```

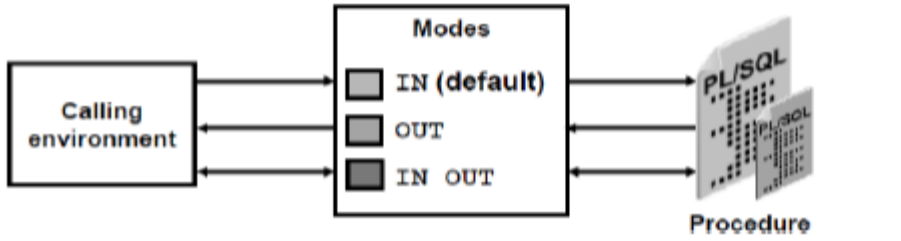
```
SELECT * FROM user_source WHERE
name='ADD_DEPT';
```

parameter-passing mode:

- An `IN` parameter mode (the default) provides values for a subprogram to process
- An `OUT` parameter mode returns a value to the caller
- An `IN OUT` parameter mode supplies an input value, which may be returned (output) as a modified value

- Parameter modes are specified in the formal parameter declaration, after the parameter name and before its data type.
- The `IN` mode is the default if no mode is specified.

```
CREATE PROCEDURE proc_name(param_name [mode] datatype)
...
```



Comparing the Parameter Modes

IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Returned to calling environment	Passed into subprogram; returned to calling environment
Formal parameter acts as a constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, expression, constant, or initialized variable	Must be a variable	Must be a variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value

Available Notations for Passing Actual Parameters

When calling a subprogram, you can write the actual parameters using the following notations:

- Positional:
  - Lists the actual parameters in the same order as the formal parameters
- Named:
  - Lists the actual parameters in arbitrary order and uses the association operator (=>) to associate a named formal parameter with its actual parameter
- Mixed:
  - Lists some of the actual parameters as positional and some as named

Using the DEFAULT Option for the Parameters

- Defines default values for parameters.
- Provides flexibility by combining the positional and named parameter-passing syntax.

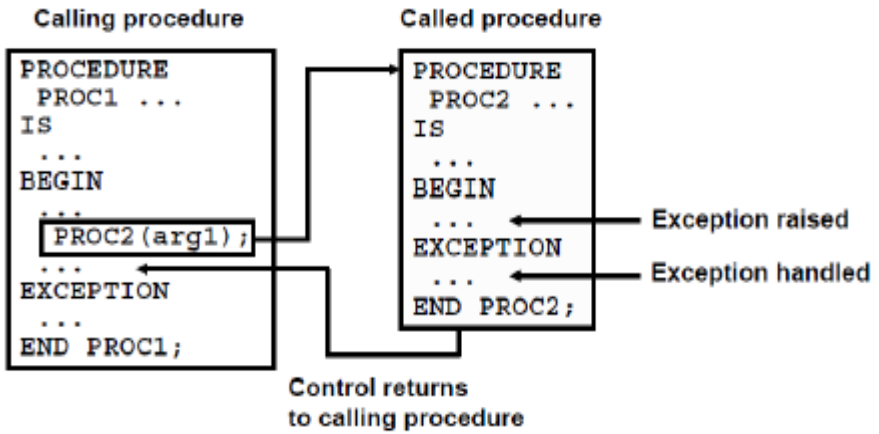
```
CREATE OR REPLACE PROCEDURE add_dept(
 p_name departments.department_name%TYPE:='Unknown',
 p_loc  departments.location_id%TYPE DEFAULT 1700)
IS
BEGIN
  INSERT INTO departments (department_id,
     department_name, location_id)
  VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
```

```
EXECUTE add_dept
EXECUTE add_dept ('ADVERTISING', p_loc => 1200)
EXECUTE add_dept (p_loc => 1200)
```

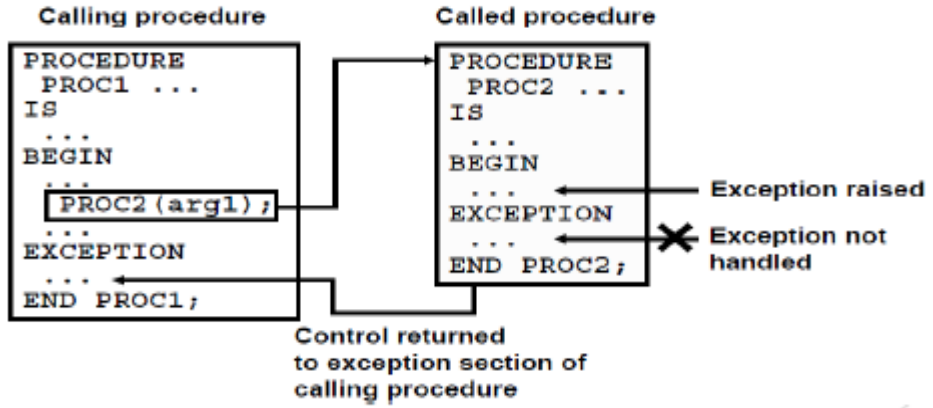
Note: Default value used only for IN parameters

PROCEDIMIENTOS, FUNCIONES Y PAQUETES

Handled Exceptions



Exceptions Not Handled



Overview of Stored Functions

A function:

- Is a named PL/SQL block that returns a value
- Can be stored in the database as a schema object for repeated execution
- Is called as part of an expression or is used to provide a parameter value

Examples :

- we can crate function to return the salary for an employee
- Function to retrieve the full name for the employee
- Function to calculate the GPA for the Student
- Function to compute the tax for a salary

Creating Functions

The PL/SQL block must have at least one `RETURN` statement.

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1, . . .)]
RETURN datatype IS|AS
 [local_variable_declarations;
  . . .]
BEGIN
  -- actions;
  RETURN expression;
END [function_name];
```

PL/SQL Block

should be same Data type

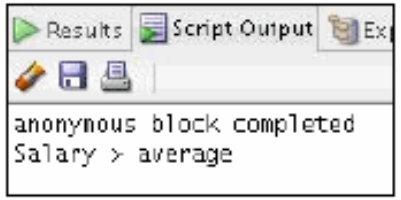
- Host variables not Allowed, also substitute variables &
- It should be at least one return expression in executable section
- Return datatype should be without size.
- Out / IN OUT can be used , but this not good Practice

Creating a Function

```
CREATE FUNCTION check_sal RETURN Boolean IS
v_dept_id employees.department_id%TYPE;
 v_empno    employees.employee_id%TYPE;
 v_sal      employees.salary%TYPE;
 v_avg_sal employees.salary%TYPE;
BEGIN
 v_empno:=205;
 SELECT salary,department_id INTO v_sal,v_dept_id FROM
employees
 WHERE employee_id= v_empno;
 SELECT avg(salary) INTO v_avg_sal FROM employees WHERE
department_id=v_dept_id;
 IF v_sal > v_avg_sal THEN
  RETURN TRUE;
 ELSE
  RETURN FALSE;
 END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
   RETURN NULL;
END;
```

Invoking a Function

```
BEGIN
 IF (check_sal IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal) THEN
 DBMS_OUTPUT.PUT_LINE('Salary > average');
 ELSE
 DBMS_OUTPUT.PUT_LINE('Salary < average');
 END IF;
END;
/
```

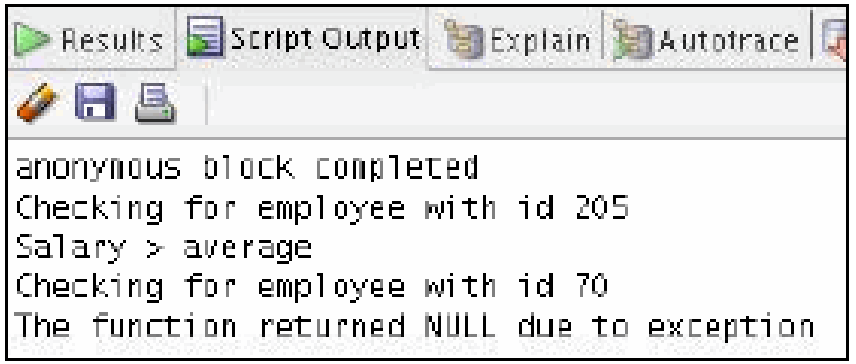


Passing a Parameter to the Function

```
DROP FUNCTION check_sal;
CREATE FUNCTION check_sal(p_empno employees.employee_id%TYPE)
RETURN Boolean IS
 v_dept_id employees.department_id%TYPE;
 v_sal     employees.salary%TYPE;
 v_avg_sal employees.salary%TYPE;
BEGIN
 SELECT salary,department_id INTO v_sal,v_dept_id FROM employees
   WHERE employee_id=p_empno;
 SELECT avg(salary) INTO v_avg_sal FROM employees
   WHERE department_id=v_dept_id;
 IF v_sal > v_avg_sal THEN
  RETURN TRUE;
 ELSE
  RETURN FALSE;
 END IF;
EXCEPTION
  ...
```

Invoking the Function with a Parameter

```
BEGIN
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 205');
 IF (check_sal(205) IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal(205)) THEN
 DBMS_OUTPUT.PUT_LINE('Salary > average');
 ELSE
 DBMS_OUTPUT.PUT_LINE('Salary < average');
 END IF;
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 70');
 IF (check_sal(70) IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal(70)) THEN
 ...
 END IF;
END;
/
```



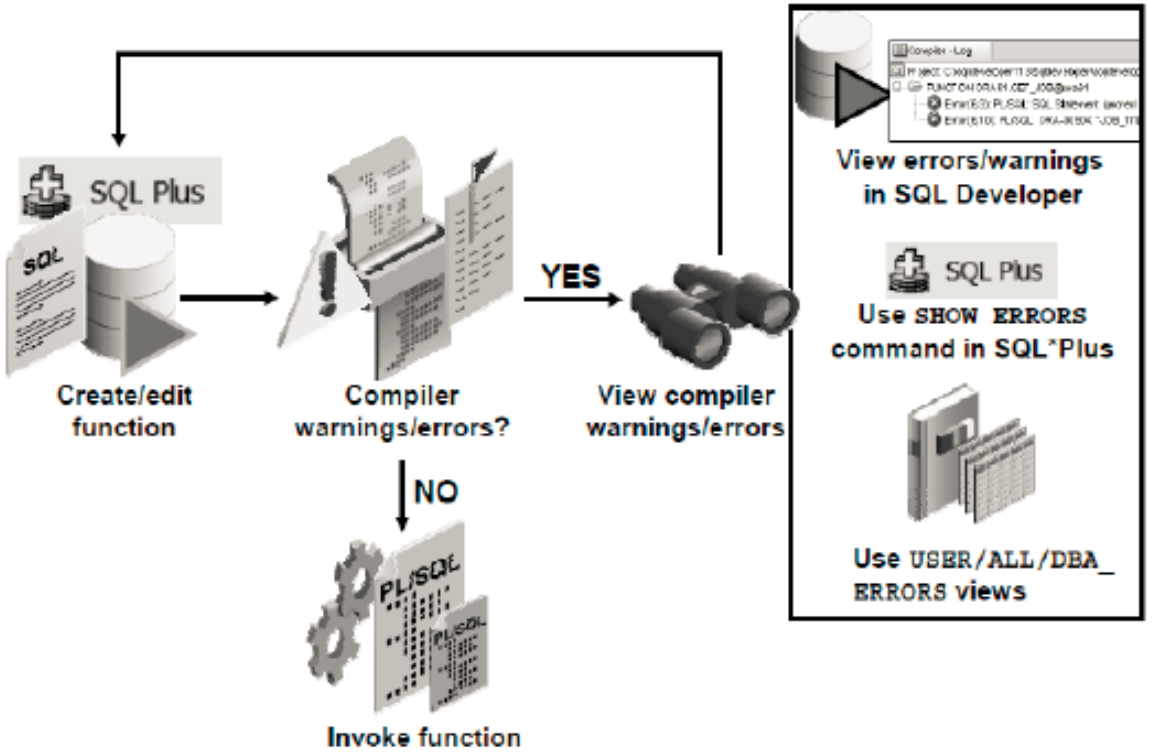
The Difference Between Procedures and Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain RETURN clause in the header	Must contain a RETURN clause in the header
Can pass values (if any) using output parameters	Must return a single value
Can contain a RETURN statement without a value	Must contain at least one RETURN statement

To Preform an Action	To return A value
Can not be used in select	Can be used in Select But it should not include OUT/ IN OUT Parameters

A Procedure that have one Parameter(OUT) would be better rewritten as A function

Creating and Running Functions: Overview



FUNCIONES Y PAQUETES

Como usar las funciones en sentencias SQL
=➔==➔

Calling User-Defined Functions in SQL Statements

User-defined functions act like built-in single-row functions and can be used in:

- **The `SELECT` list or clause of a query**
- **Conditional expressions of the `WHERE` and `HAVING` clauses**
- **The `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses of a query**
- **The `VALUES` clause of the `INSERT` statement**
- **The `SET` clause of the `UPDATE` statement**

FUNCIONES Y PAQUETES

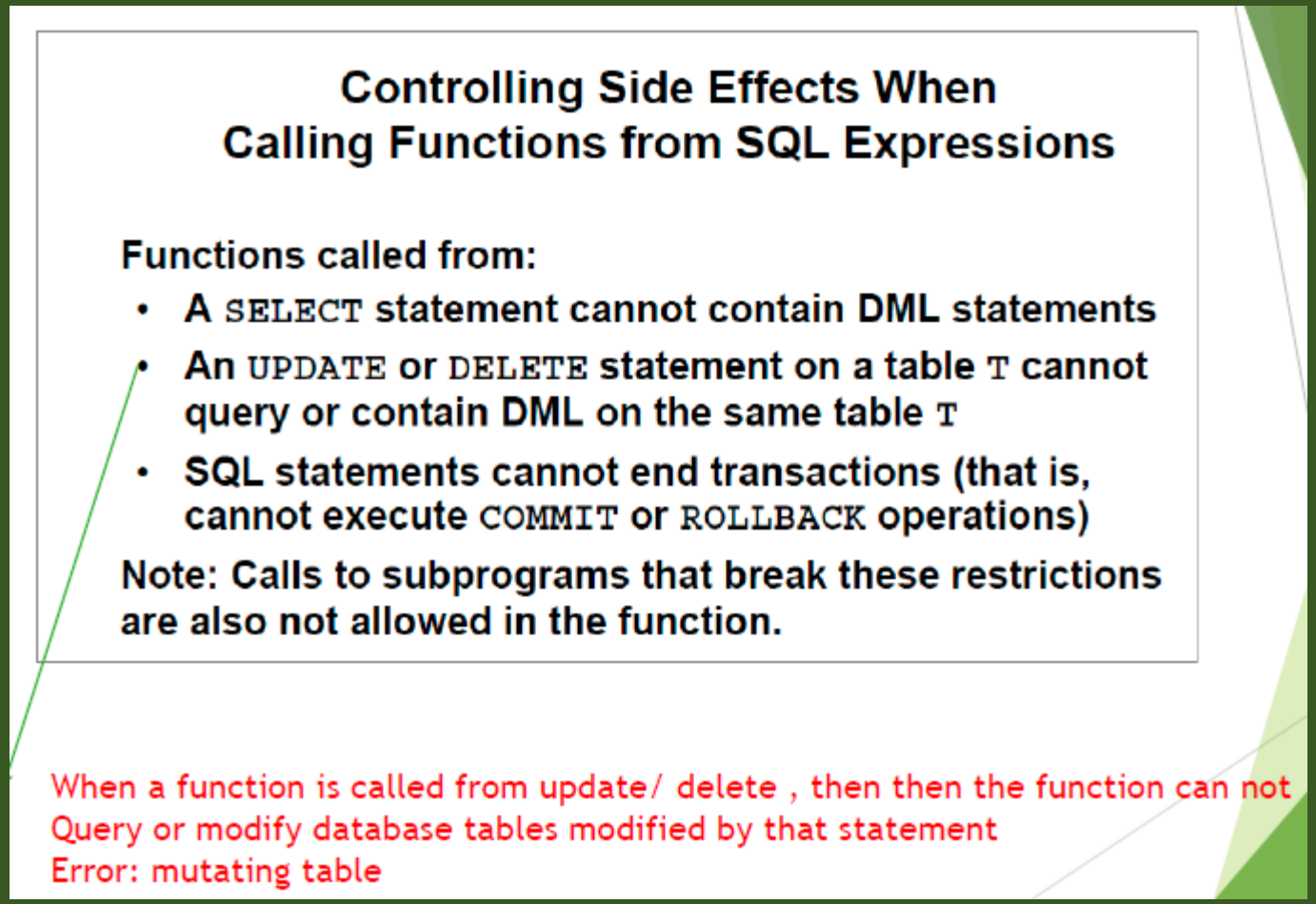
Restrictions When Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
  - Be stored in the database
  - Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types (Record, table , Boolean)
  - Return valid SQL data types, not PL/SQL-specific types
- When calling functions in SQL statements:
  - Parameters must be specified with positional notation
  - You must own the function or have the `EXECUTE` privilege

This Before 11g only

- Can not be used in Check constraint (create table/ alter table)
- Can not be used as default value for a column

FUNCIONES Y PAQUETES



What Are PL/SQL Packages?

- A package is a schema object that groups logically related PL/SQL types, variables, and subprograms.
- Packages usually have two parts:
  - A specification (spec)
  - A body
- The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package.
- The body defines the queries for the cursors and the code for the subprograms.
- Enable the Oracle server to read multiple objects into memory at once.

Advantages of Using Packages

- Modularity: Encapsulating related constructs
- Easier maintenance: Keeping logically related functionality together
- Easier application design: Coding and compiling the specification and body separately
- Hiding information:
  - Only the declarations in the package specification are visible and accessible to applications
  - Private constructs in the package body are hidden and inaccessible
  - All coding is hidden in the package body

Creating the Package Specification: Using the CREATE PACKAGE Statement

```
CREATE [OR REPLACE] PACKAGE package_name IS|AS
    public type and variable declarations
    subprogram specifications
END [package_name];
```

- The OR REPLACE option drops and re-creates the package specification.
- Variables declared in the package specification are initialized to NULL by default.
- All the constructs declared in a package specification are visible to users who are granted privileges on the package.

Creating the Package Body

```
CREATE [OR REPLACE] PACKAGE BODY package_name IS|AS
    private type and variable declarations
    subprogram bodies
[BEGIN initialization statements]
END [package_name];
```

- The OR REPLACE option drops and re-creates the package body.
- Identifiers defined in the package body are *private* and not visible outside the package body.
- All *private* constructs must be declared before they are referenced.
- Public constructs are visible to the package body.

Guidelines for Writing Packages

- Develop packages for general use.
- Define the package specification before the body.
- The package specification should contain only those constructs that you want to be public.
- Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions.
- The fine-grain dependency management reduces the need to recompile referencing subprograms when a package specification changes.
- The package specification should contain as few constructs as possible.

Guidelines for Writing Packages

The package specification should contain as few constructs as possible.

Package Specification

Procedure insert emp
Procedure update emp_sal
Procedure delete emp

Try to keep package specification Simple as you can

Package body

Proceure check_before_insert
Proceure check_before_delete
Procedure insert emp
Procedure update emp_sal
Procedure delete emp

Try to add additional codes you need in package body as private subprograms

The Visibility of a Package's Components

```
create or replace package p_test
is
c_var1 constant number:=10;
c_var2 varchar2(100):='welcome';

procedure print;
end;
```

c_var1/c_var2 can be referenced any place in package body

c_var3 can be referenced any place in package body

c_var4 can be referenced only in print procedure

```
create or replace package body p_test
is
c_var3 varchar2(100):='hi there';
 procedure print
 is
 c_var4 varchar2(100):='hi';
 begin
 dbms_output.put_line('this variable came from package spec. '||c_var1);
 dbms_output.put_line('this variable came from package spec. '||c_var2);
 dbms_output.put_line('this variable came from package body. '||c_var3);
  dbms_output.put_line('this variable came from print Proc. '||c_var4);
 end;

end;

execute p_test.print;
```

Dictionary tables and Dropping the Package

```
select * from user_objects
where oBject_name='P_TEST'

SELECT * FROM USER_SOURCE
WHERE NAME='P_TEST'
AND TYPE='PACKAGE';

SELECT * FROM USER_SOURCE
WHERE NAME='P_TEST'
AND TYPE='PACKAGE BODY';

--to drop package specification and body

drop package p_test;

--to drop only package body

drop package body p_test;
```

Overloading Subprograms in PL/SQL

- Enables you to create two or more subprograms with the same name
- Requires that the subprogram's formal parameters differ in number, order, or data type family
- Enables you to build flexible ways for invoking subprograms with different data
- Provides a way to extend functionality without loss of existing code; that is, adding new parameters to existing subprograms
- Provides a way to overload local subprograms, package subprograms, and type methods, but not stand-alone subprograms

Overloading Subprograms (continued)

Restrictions

You cannot overload:

- Two subprograms if their formal parameters differ only in data type and the different data types are in the same family (`NUMBER` and `DECIMAL` belong to the same family.)
- Two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same family (`VARCHAR` and `STRING` are PL/SQL subtypes of `VARCHAR2`.)
- Two functions that differ only in return type, even if the types are in different families

You get a run-time error when you overload subprograms with the preceding features.

Note: The preceding restrictions apply if the names of the parameters are also the same. If you use different names for the parameters, you can invoke the subprograms by using named notation for the parameters.

Resolving Calls

The compiler tries to find a declaration that matches the call. It searches first in the current scope and then, if necessary, in successive enclosing scopes. The compiler stops searching if it finds one or more subprogram declarations in which the name matches the name of the called subprogram. For similarly named subprograms at the same level of scope, the compiler needs an exact match in number, order, and data type between the actual and formal parameters.

Overloading Procedures Example: Creating the Package Specification

```
CREATE OR REPLACE PACKAGE dept_pkg IS
  PROCEDURE add_department
    (p_deptno departments.department_id%TYPE,
     p_name departments.department_name%TYPE :='unknown',
     p_loc  departments.location_id%TYPE := 1700);

PROCEDURE add_department
  (p_name departments.department_name%TYPE := 'unknown',
   p_loc  departments.location_id%TYPE := 1700);
END dept_pkg;
/
```

Overloading Procedures Example: Creating the Package Body

```
CREATE OR REPLACE PACKAGE BODY dept_pkg  IS
PROCEDURE add_department -- First procedure's declaration
  (p_deptno departments.department_id%TYPE,
   p_name   departments.department_name%TYPE := 'unknown',
   p_loc    departments.location_id%TYPE := 1700) IS
  BEGIN
    INSERT INTO departments(department_id,
      department_name, location_id)
    VALUES  (p_deptno, p_name, p_loc);
  END add_department;

PROCEDURE add_department -- Second procedure's declaration
  (p_name   departments.department_name%TYPE := 'unknown',
   p_loc    departments.location_id%TYPE := 1700) IS
  BEGIN
    INSERT INTO departments (department_id,
      department_name, location_id)
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);
  END add_department;
 END dept_pkg; /
```

Overloading and the STANDARD Package

```
--overloading the standard package
create or replace package override
is
/*we defined to_char function and this function already exist
as oracle bulit-in function
*/

function to_char( p1 number, p2 date )
return varchar2;

procedure print;

end;
```

```
create or replace package body override
is

      function to_char( p1 number, p2 date )
      return varchar2
      is
      begin
      return p1||p2;
      end;

      procedure print
      is
      begin
      dbms_output.put_line(to_char(1,'1-jan-81' ));
      dbms_output.put_line(standard.to_char(10));
      end;

end;
```

the oracle will understand that to_char is the function from the package, not the built-in

when you prefix standard, this to tell oracle to use the built-in

Note:
Standard only used in PL/SQL
You can not do this outside PL/SQL
Select standard.to_char….

Illegal Procedure Reference

- Block-structured languages such as PL/SQL must declare identifiers before referencing them.
- Example of a referencing problem:

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
  PROCEDURE award_bonus(. . .) IS
  BEGIN
    calc_rating(. . .);     --illegal reference
  END;

PROCEDURE calc_rating(. . .) IS
  BEGIN
    ...
  END;
END forward_pkg;
/
```

Using Forward Declarations to Solve Illegal Procedure Reference

In the package body, a forward declaration is a private subprogram specification terminated by a semicolon.

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
 PROCEDURE calc_rating (...);-- forward declaration

 -- Subprograms defined in alphabetical order

 PROCEDURE award_bonus(...) IS
 BEGIN
   calc_rating (...);        -- reference resolved!
   . . .
 END;

 PROCEDURE calc_rating (...) IS -- implementation
 BEGIN
   . . .
 END;
END forward_pkg;
```

Using Oracle-Supplied Packages

- The Oracle-supplied packages:
  - Are provided with the Oracle server
  - Extend the functionality of the database
  - Enable access to certain SQL features that are normally restricted for PL/SQL
- For example, the `DBMS_OUTPUT` package was originally designed to debug PL/SQL programs.

Examples of Some Oracle-Supplied Packages

Here is an abbreviated list of some Oracle-supplied packages:

- DBMS_OUTPUT
- UTL_FILE
- UTL_MAIL
- DBMS_ALERT
- DBMS_LOCK
- DBMS_SESSION
- HTP
- DBMS_SCHEDULER

How the DBMS_OUTPUT Package Works

The DBMS_OUTPUT package enables you to send messages from stored subprograms and triggers.

- PUT and PUT_LINE place text in the buffer.
- GET_LINE and GET_LINES read the buffer.
- Messages are not sent until the sending subprogram or trigger completes.
- Use SET SERVEROUTPUT ON to display messages in SQL Developer and SQL*Plus.

DBMS_OUTPUT

- PUT appends text from the procedure to the current line of the line output buffer.
- NEW_LINE places an end-of-line marker in the output buffer.
- PUT_LINE combines the action of PUT and NEW_LINE (to trim leading spaces).
- GET_LINE retrieves the current line from the buffer into a procedure variable.
- GET_LINES retrieves an array of lines into a procedure-array variable.
- ENABLE/DISABLE enables and disables calls to DBMS_OUTPUT procedures.

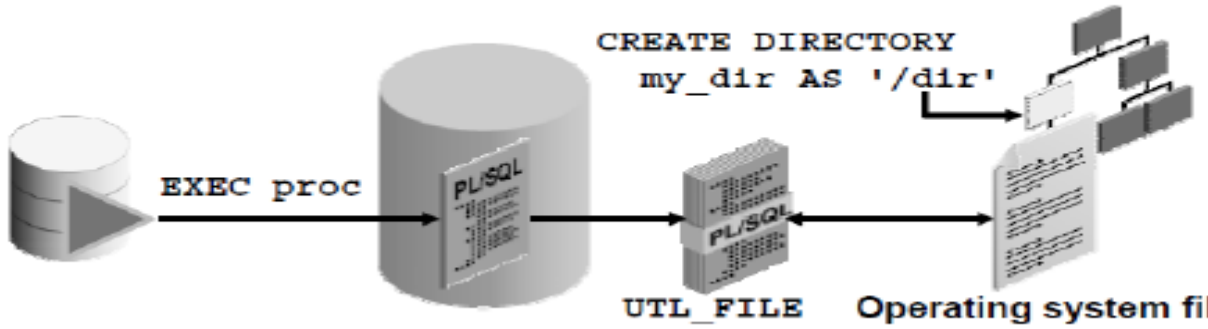
The buffer size
The minimum is 2,000 and the maximum is unlimited. The default is 20,000.
An integer parameter between 2,000 and 1,000,000 in the ENABLE procedure

Actually the DBMS_OUTPUT is not that important for developers
Because they deal with messages using the development applications features
Example: alert windows, console windows

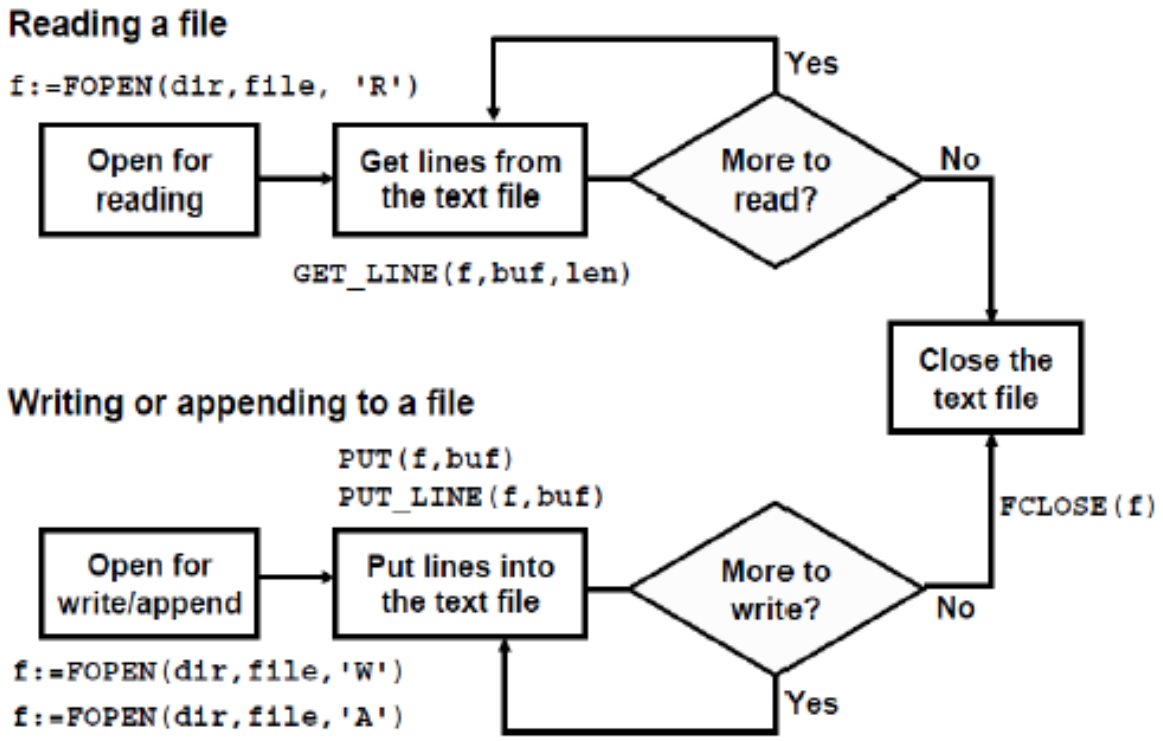
Using the UTL_FILE Package to Interact with Operating System Files

The UTL_FILE package extends PL/SQL programs to read and write operating system text files:

- Provides a restricted version of operating system stream file I/O for text files
- Can access files in operating system directories defined by a CREATE DIRECTORY statement



File Processing Using the UTL_FILE Package: Overview



Using the Available Declared Exceptions in the UTL_FILE Package

Exception Name	Description
INVALID_PATH	File location invalid
INVALID_MODE	The open_mode parameter in FOPEN is invalid
INVALID_FILEHANDLE	File handle invalid
INVALID_OPERATION	File could not be opened or operated on as requested
READ_ERROR	Operating system error occurred during the read operation
WRITE_ERROR	Operating system error occurred during the write operation
INTERNAL_ERROR	Unspecified PL/SQL error

The FOPEN and IS_OPEN Functions: Example

- This FOPEN function opens a file for input or output.

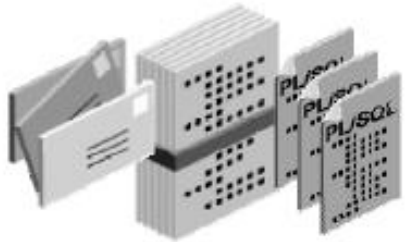
```
FUNCTION FOPEN (p_location  IN VARCHAR2,
                p_filename  IN VARCHAR2,
                p_open_mode IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

- The IS_OPEN function determines whether a file handle refers to an open file.

```
FUNCTION IS_OPEN (p_file IN FILE_TYPE)
RETURN BOOLEAN;
```

What Is the UTL_MAIL Package?

- A utility for managing email
- Requires the setting of the SMTP_OUT_SERVER database initialization parameter
- Provides the following procedures:
  - SEND for messages without attachments
  - SEND_ATTACH_RAW for messages with binary attachments
  - SEND_ATTACH_VARCHAR2 for messages with text attachments



How UTL_MAIL work ?????

1- THE DBA install UTL_MAIL package

@$ORACLE_HOME/rdbms/admin/utlmail.sql
@$ORACLE_HOME/rdbms/admin/prvtmail.plb

2- define the SMTP_OUT_SERVER (init.ora)

3- the DBA grant Execute on UTL_MAIL to a USER/ Public

4- The DBA add Record in ACL (Access Control List)
Using package called DBMS_NETWORK_ACL_ADMIN

5- now the USER can use UTL_MAIL

Note: No need to know details for steps 1-4

Summary of UTL_MAIL Subprograms

Subprogram	Description
SEND procedure	Packages an email message, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients
SEND_ATTACH_RAW Procedure	Represents the SEND procedure overloaded for RAW attachments
SEND_ATTACH_VARCHAR2 Procedure	Represents the SEND procedure overloaded for VARCHAR2 attachments

The SEND Procedure Syntax

Packages an email message into the appropriate format, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients.

```
UTL_MAIL.SEND (
   sender      IN    VARCHAR2 CHARACTER SET ANY_CS,
   recipients  IN    VARCHAR2 CHARACTER SET ANY_CS,
   cc          IN    VARCHAR2 CHARACTER SET ANY_CS
                       DEFAULT NULL,
   bcc         IN    VARCHAR2 CHARACTER SET ANY_CS
                       DEFAULT NULL,
   subject     IN    VARCHAR2 CHARACTER SET ANY_CS
                       DEFAULT NULL,
   message     IN    VARCHAR2 CHARACTER SET ANY_CS,
   mime_type   IN    VARCHAR2
                       DEFAULT 'text/plain; charset=us-ascii',
   priority    IN    PLS_INTEGER DEFAULT NULL);
```

MIME:Multipurpose Internet Mail Extensions

The SEND_ATTACH_RAW Procedure

This procedure is the SEND procedure overloaded for RAW attachments.

```
UTL_MAIL.SEND_ATTACH_RAW (
   sender           IN    VARCHAR2 CHARACTER SET ANY_CS,
   recipients       IN    VARCHAR2 CHARACTER SET ANY_CS,
   cc               IN    VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
   bcc              IN    VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
   subject          IN    VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
   message          IN    VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
   mime_type        IN    VARCHAR2 DEFAULT CHARACTER SET ANY_CS
                            DEFAULT 'text/plain; charset=us-ascii',
   priority         IN    PLS_INTEGER DEFAULT 3,
   attachment       IN    RAW,
   att_inline       IN    BOOLEAN DEFAULT TRUE,
   att_mime_type    IN    VARCHAR2 CHARACTER SET ANY_CS
                            DEFAULT 'text/plain; charset=us-ascii',
   att_filename     IN    VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL);
```

MIME:Multipurpose Internet Mail Extensions

The SEND_ATTACH_VARCHAR2 Procedure

This procedure is the SEND procedure overloaded for VARCHAR2 attachments.

```
UTL_MAIL.SEND_ATTACH_VARCHAR2 (
   sender            IN    VARCHAR2 CHARACTER SET ANY_CS,
   recipients        IN    VARCHAR2 CHARACTER SET ANY_CS,
   cc                IN    VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
   bcc               IN    VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
   subject           IN    VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
   message           IN    VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
   mime_type         IN    VARCHAR2 CHARACTER SET ANY_CS
                             DEFAULT 'text/plain; charset=us-ascii',
   priority          IN    PLS_INTEGER DEFAULT 3,
   attachment        IN    VARCHAR2 CHARACTER SET ANY_CS,
   att_inline        IN    BOOLEAN DEFAULT TRUE,
   att_mime_type     IN    VARCHAR2 CHARACTER SET ANY_CS
                             DEFAULT 'text/plain; charset=us-ascii',
   att_filename      IN    VARCHAR2CHARACTER SET ANY_CS DEFAULT NULL);
```

USO DEL SQL DINÁMICO

SQL DINAMICO

Execution Flow of SQL

- All SQL statements go through some or all of the following stages:
  - Parse
  - Bind
  - Execute
  - Fetch

- Parse : check the statement syntax , validating the statement
Ensure all referencing objects are correct , The privileges exists
- Bind : check the bind variable if the statement contains Bind Var.
- Execute: execute the statement(non Queries statements)
- Fetch : retrieve the rows (Queries statements)

SQL DINÁMICO

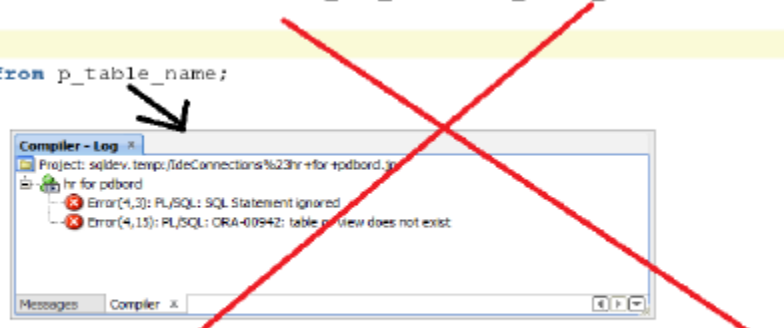
What Is Dynamic SQL???

The SQL statements are created dynamically at run time
(not compile time)

Example: creating a procedure for deleting any table

You may think to do this:

```
create or replace procedure delete_any_table ( p_table_name varchar2)
is
begin
  delete from p_table_name;
  commit;
end;
```



The solution is to use dynamic SQL

- execute immediate (native Dynamic)
- DBMS_SQL

What Is Dynamic SQL?

Use dynamic SQL to create a SQL statement whose structure may change during run time. Dynamic SQL:

- Is constructed and stored as a character string, string variable, or string expression within the application.
- Is a SQL statement with varying column data, or different conditions with or without placeholders (bind variables).
- Enables DDL, DCL, or session-control statements to be written and executed from PL/SQL.
- Is executed with Native Dynamic SQL statements or the `DBMS_SQL` package.

What Is Dynamic SQL?

- The full text of the dynamic SQL statement might be unknown until run time; therefore, its syntax is checked at *run time* rather than at *compile time*.
- You can use dynamic SQL statements to make your PL/SQL programs more general and flexible.

SQL DINÁMICO

Execute immediate as dynamic string

```
create or replace procedure delete_any_table
( p_table_name varchar2)
is
v_no_rec number;
begin
execute immediate 'delete from '||p_table_name;
commit; --same rules for commit and rollback
v_no_rec:=sql%rowcount;
dbms_output.put_line(v_no_rec ||' record(s) deleted form '||p_table_name );
end;
```

```
execute delete_any_table('empl');
select * from empl;
```

SQL DINÁMICO

dynamic SQL with multi row query

```
create or replace procedure emp_list ( p_dept_id number default null )
is
type  c_emp_dept is ref cursor;
d_cursor c_emp_dept;
v_empno employees.employee_id%type;
v_first_name employees.first_name%type;
v_sql varchar2(1000):='select employee_id, first_name from employees';
begin
   if p_dept_id is null then
   open d_cursor for v_sql;
   else
   v_sql:=v_sql||'  where department_id=:id';
   open d_cursor for v_sql using p_dept_id;
   end if;

     loop
      fetch d_cursor into v_empno, v_first_name;
      exit when d_cursor%notfound;
      dbms_output.put_line(v_empno||' '||v_first_name);
    end loop;
close d_cursor;

end;
```

```
--to get all the employees
execute emp_list;

--to get all the employees in specific dept
execute emp_list (30);
```

SQL DINÁMICO

dynamic SQL to execute anonymous Block

```
declare
     v_code varchar2(100):=
    'begin
    dbms_output.put_line(''welcome'');
    end;
    ';
begin
execute immediate v_code;
end;
```

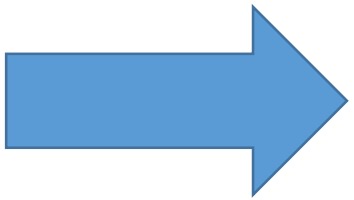
CONSIDERACIONES DE DISEÑO PLSQL CODE

CONSIDERACIONES DE DISEÑO – PLSQL CODE

Standardizing Constants and Exceptions

Constants and exceptions are typically implemented using a bodiless package (that is, a package specification).

- Standardizing helps to:
  - Develop programs that are consistent
  - Promote a higher degree of code reuse
  - Ease code maintenance
  - Implement company standards across entire applications
- Start with standardization of:
  - Exception names
  - Constant definitions



CONSIDERACIONES DE DISEÑO – PLSQL CODE

```
Begin
Delete form departments where
Department_id=p_dept_id
…
…
Exception
when  error_pkg.e_fk_err Then…
….
End;
```

Standardizing Exceptions

Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.

```
CREATE OR REPLACE PACKAGE error_pkg IS
  e_fk_err      EXCEPTION;
  e_seq_nbr_err EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_fk_err, -2292);
  PRAGMA EXCEPTION_INIT (e_seq_nbr_err, -2277);
  ...
END error_pkg;
/
```

```
delete from departments;

Script Output
Task completed in 0.053 seconds
Error starting at line 1 in command:
delete from departments
Error report:
SQL Error: ORA-02292: integrity constraint (HR.EMP_DEPT_FK) violated - child record found
02292. 00000 - "integrity constraint (%s.%s) violated - child record found"
*Cause:    attempted to delete a parent key value that had a foreign
           dependency.
*Action:   delete dependencies first then parent or disable constraint.
```

CONSIDERACIONES DE DISEÑO – PLSQL CODE

Standardizing Exception Handling

Consider writing a subprogram for common exception handling to:

- Display errors based on `SQLCODE` and `SQLERRM` values for exceptions
- Track run-time errors easily by using parameters in your code to identify:
  - The procedure in which the error occurred
  - The location (line number) of the error
  - `RAISE_APPLICATION_ERROR` using stack trace capabilities, with the third argument set to `TRUE`

```
RAISE_APPLICATION_ERROR(-20001, 'My first error', TRUE);
```

Standardizing Constants

For programs that use local variables whose values should not change:

- Convert the variables to constants to reduce maintenance and debugging
- Create one central package specification and place all constants in it

```
create or replace package global_Measurement
is

  c_mile_to_km   constant number:=1.6093;
  c_kilo_to_mile constant number:=0.6214;

end;
```

CONSIDERACIONES DE DISEÑO – PLSQL CODE

Local Subprograms

- A local subprogram is a PROCEDURE or FUNCTION defined at the end of the declarative section.

```
CREATE PROCEDURE employee_sal(p_id NUMBER) IS
   v_emp employees%ROWTYPE;
   FUNCTION tax(p_salary VARCHAR2) RETURN NUMBER IS
   BEGIN
     RETURN p_salary * 0.825;
   END tax;
BEGIN
   SELECT * INTO v_emp
   FROM EMPLOYEES WHERE employee_id = p_id;
   DBMS_OUTPUT.PUT_LINE('Tax: ' || tax(v_emp.salary));
END;
/
EXECUTE employee_sal(100)
```

Local Function →

1- can be accessible only from the block owner
2- compiled as a part of the owner block

Why using local subprogram?
1- reduction of repetitive code
2- code readability
3- easy maintenance

CONSIDERACIONES DE DISEÑO – PLSQL CODE

Definer's Rights Versus Invoker's Rights

Definer's rights:

- Used prior to Oracle8*i*
- Programs execute with the privileges of the creating user.
- User does not require privileges on underlying objects that the procedure accesses. User requires privilege only to execute a procedure.

Invoker's rights:

- Introduced in Oracle8*i*
- Programs execute with the privileges of the calling user.
- User requires privileges on the underlying objects that the procedure accesses.

CONSIDERACIONES DE DISEÑO – PLSQL CODE

Specifying Invoker's Rights: Setting AUTHID to CURRENT_USER

```
CREATE OR REPLACE PROCEDURE add_dept(
  p_id NUMBER, p_name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
  INSERT INTO departments
  VALUES (p_id, p_name, NULL, NULL);
END;
```

The default is AUTHID DEFINER.

When used with stand-alone functions, procedures, or packages:

- **Names used in queries, DML, Native Dynamic SQL, and DBMS_SQL package are resolved in the invoker's schema**
- **Calls to other packages, functions, and procedures are resolved in the definer's schema**

Specify CURRENT_USER to indicate that the package executes with the privileges of CURRENT_USER.

CONSIDERACIONES DE DISEÑO – PLSQL CODE

Autonomous Transactions

- Are independent transactions started by another main transaction
- Are specified with `PRAGMA AUTONOMOUS_TRANSACTION`

```
CREATE OR REPLACE PROCEDURE parent_block IS
BEGIN
   INSERT INTO t(test_value)
   VALUES('Parent block insert');

child_block;
ROLLBACK;
END parent_block;
 /
```

```
CREATE OR REPLACE PROCEDURE child_block IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO t(test_value)
   VALUES('Child block insert');
   COMMIT;
END child_block;

 -- empty the test table
TRUNCATE TABLE t;
```

This mean this procedure is independent
The commit not affect the caller procedure

CONSIDERACIONES DE DISEÑO – PLSQL CODE

Features of Autonomous Transactions

- Are independent of the main transaction
- Suspend the calling transaction until the autonomous transactions are completed
- Are not nested transactions
- Do not roll back if the main transaction rolls back
- Enable the changes to become visible to other transactions upon a commit
- Are started and ended by individual subprograms and not by nested or anonymous PL/SQL blocks

```
create or replace
package body area
is
PRAGMA AUTONOMOUS_TRANSACTION;
      function square_area( p_side number )
      return number
      is
      begin
      return p_side*p_side;
      end;

      function rectangle_area( p_l number,p_w number )
      return number
      is
      begin
      return p_l*p_w;
      end;

end;
```

CONSIDERACIONES DE DISEÑO – PLSQL CODE

Using the PARALLEL_ENABLE Hint

- Can be used in functions as an optimization hint
- Indicates that a function can be used in a parallelized query or parallelized DML statement

```
CREATE OR REPLACE FUNCTION f2 (p_p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p_p1 * 2;
END f2;
```

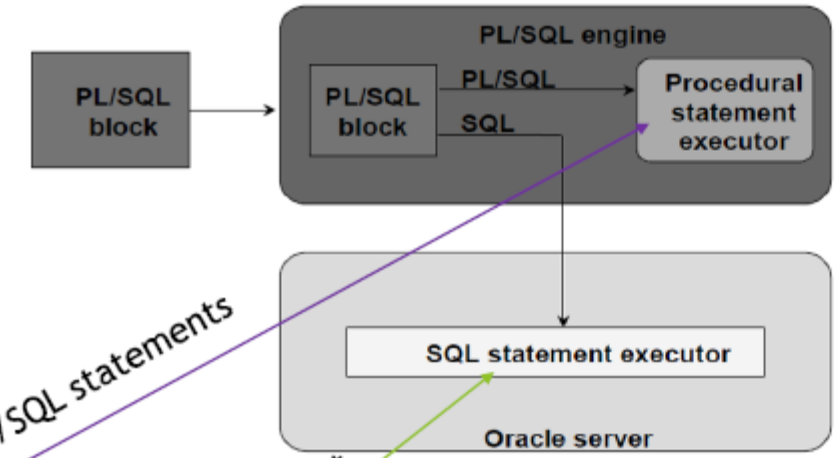
```
SELECT /*+ PARALLEL(4) */
First_name, f2(employee_id)
From
employees
```

Oracle will open 4 processes to execute this statement
Each process take subset of data

CONSIDERACIONES DE DISEÑO – PLSQL CODE

BULK Binding

PL/SQL Environment




```
Begin
...
..
For I in (select * from x )
Loop
Insert into....
End loop;
...
```

When dealing with collections this make the performance less
So Oracle introduce something called BULK Binding
(send the SQL statements using bulk)


GALAXY
TRAINING