PLSQL

Lesson 03: Cursors, Exception handling, Procedures, Functions, Packages, Adv Packages concepts.

Lesson Objectives

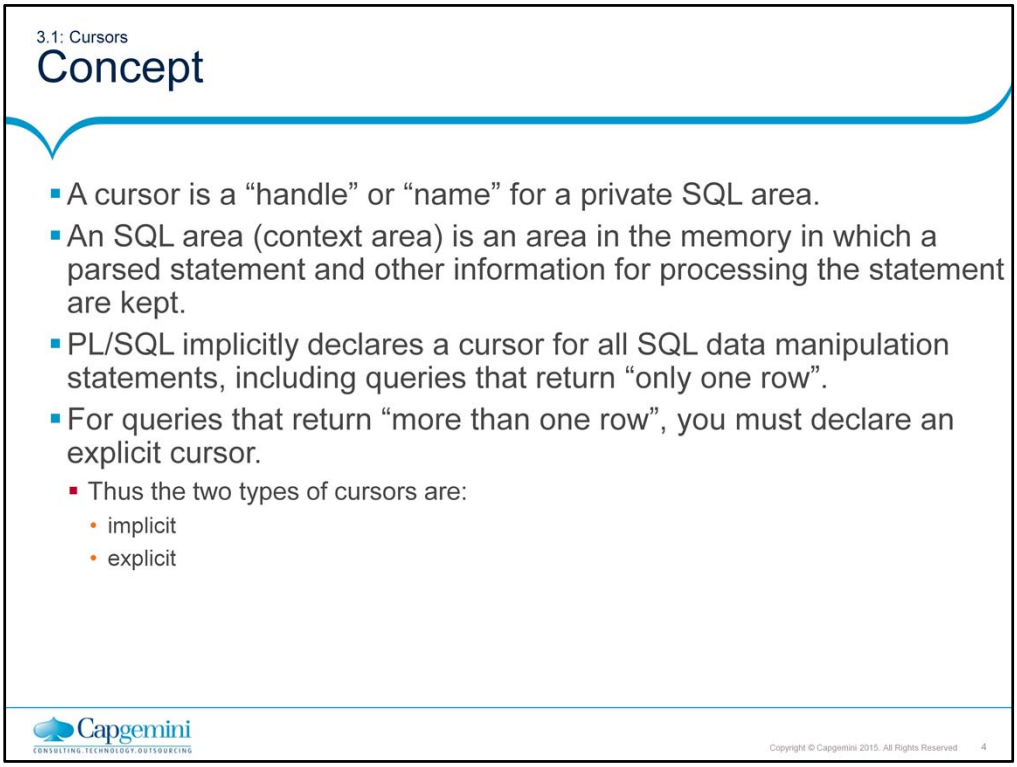
- To understand the following topics:
  - Introduction to Cursors
  - Implicit and Explicit Cursors
  - Cursor attributes
  - Processing Implicit Cursors and Explicit Cursors
  - Error Handling
    - Predefined Exceptions
    - Numbered Exceptions
    - User Defined Exceptions
  - Raising Exceptions
  - Control passing to Exception Handlers

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 2

Lesson Objectives

- To understand the following topics:
  - RAISE_APPLICATION_ERROR
  - Subprograms in PL/SQL
  - Anonymous blocks versus Stored Subprograms
  - Procedure
    - Subprogram Parameter modes
  - Functions
  - Packages
    - Package Specification and Package Body

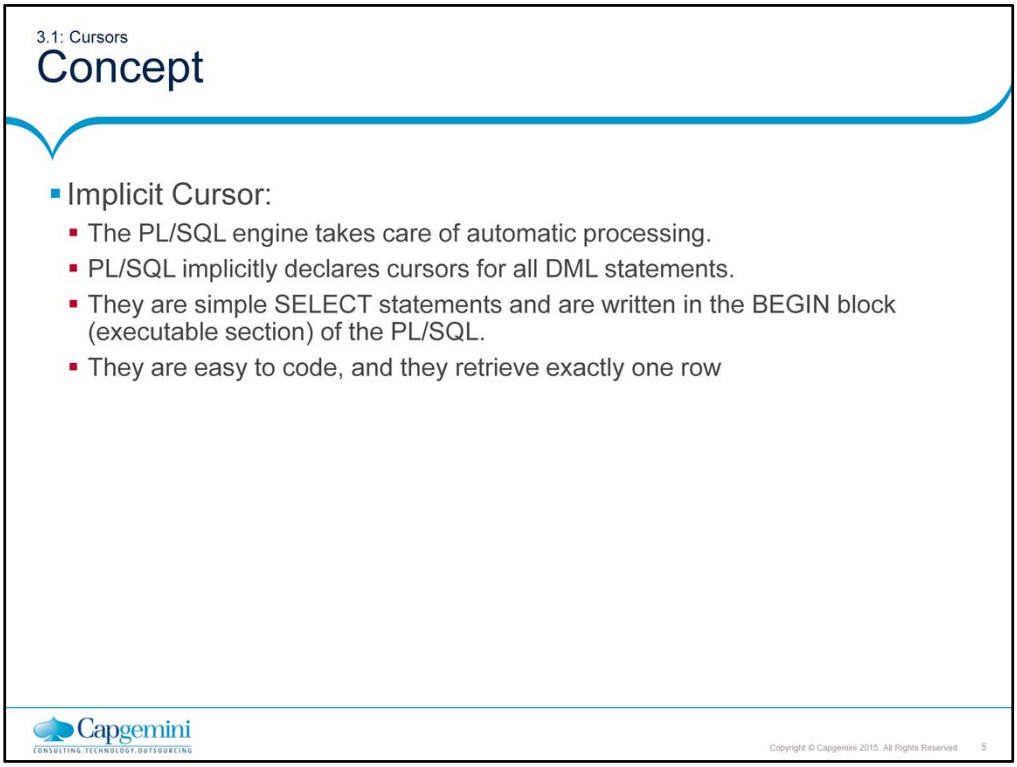


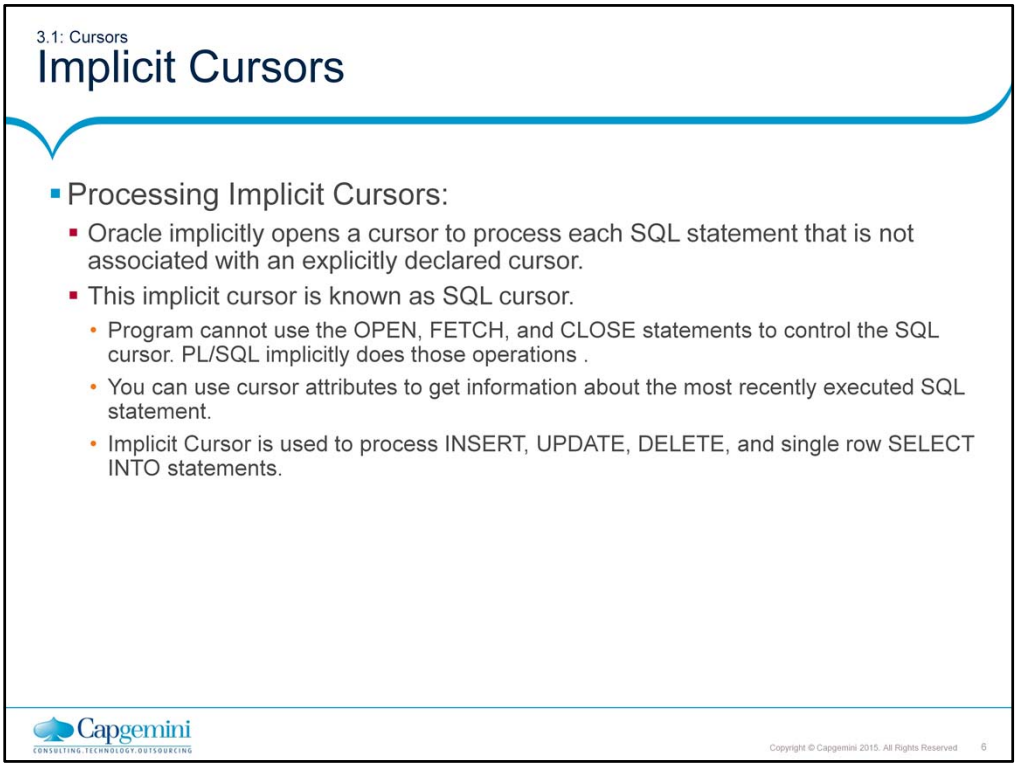
Introduction to Cursors:
ORACLE allocates memory on the Oracle server to process SQL statements. It is called as “context area”. Context area stores information like number of rows processed, the set of rows returned by a query, etc.
A Cursor is a “handle” or “pointer” to the context area. Using this cursor the PL/SQL program can control the context area, and thereby access the information stored in it.
There are two types of cursors - “explicit” and “implicit”.

In an explicit cursor, a cursor name is explicitly assigned to a SELECT statement through CURSOR IS statement.

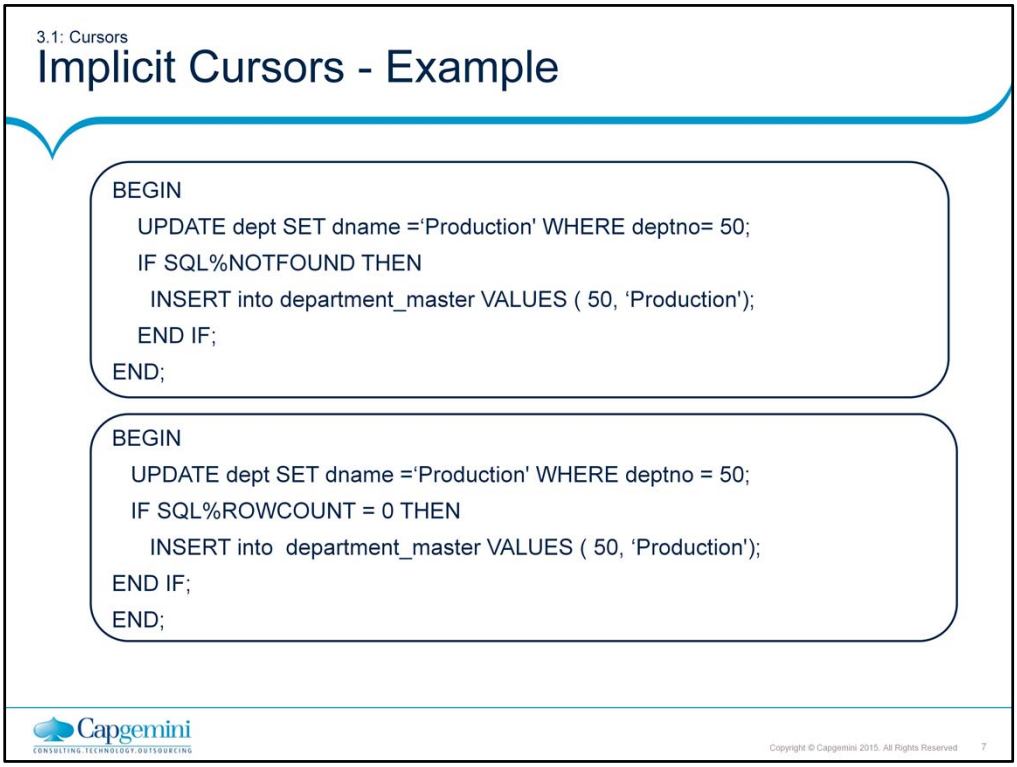
An implicit cursor is used for all other SQL statements.

Processing an explicit cursor involves four steps. In case of implicit cursors, the PL/SQL engine automatically takes care of these four steps.

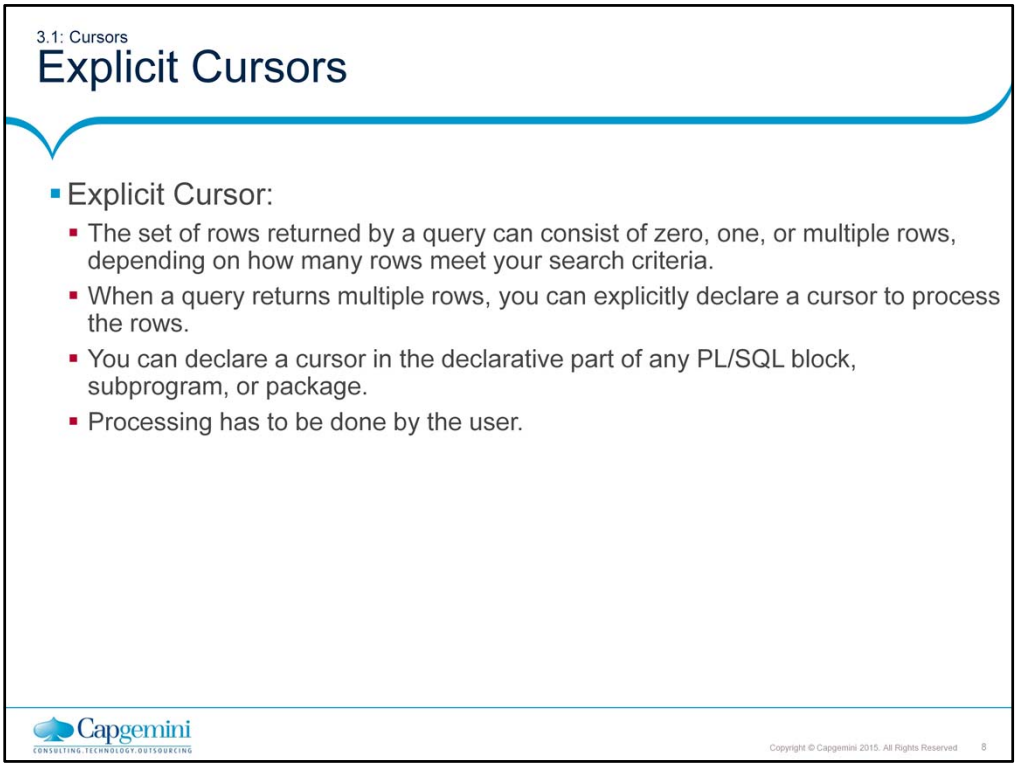

3.1: Cursors
Concept
Implicit Cursor:
The PL/SQL engine takes care of automatic processing.
PL/SQL implicitly declares cursors for all DML statements.
They are simple SELECT statements and are written in the BEGIN block (executable section) of the PL/SQL.
They are easy to code, and they retrieve exactly one row
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
5



Processing Implicit Cursors:
All SQL statements are executed inside a context area and have a cursor, which points to that context area. This implicit cursor is known as SQL cursor.
Implicit SQL cursor is not opened or closed by the program. PL/SQL implicitly opens the cursor, processes the SQL statement, and closes the cursor.
Implicit cursor is used to process INSERT, UPDATE, DELETE, and single row SELECT INTO statements.
The cursor attributes can be applied to the SQL cursor.



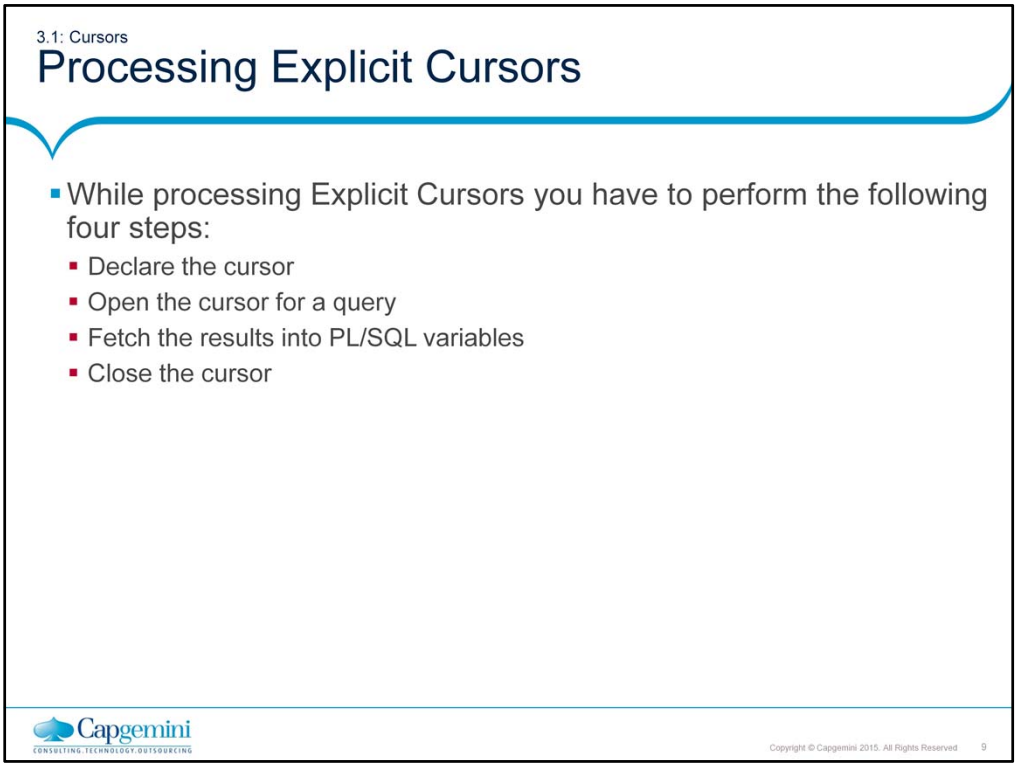
Processing Implicit Cursors:
Note:
SQL%NOTFOUND should not be used with SELECT INTO …. Statement.
This is because SELECT INTO…. Statement will raise an ORACLE error if no rows are selected, and
control will pass to exception * section (discussed later), and
SQL%NOTFOUND statement will not be executed at all
The slide shows two code snippets using Cursor attributes SQL%NOTFOUND and SQL%ROWCOUNT respectively.

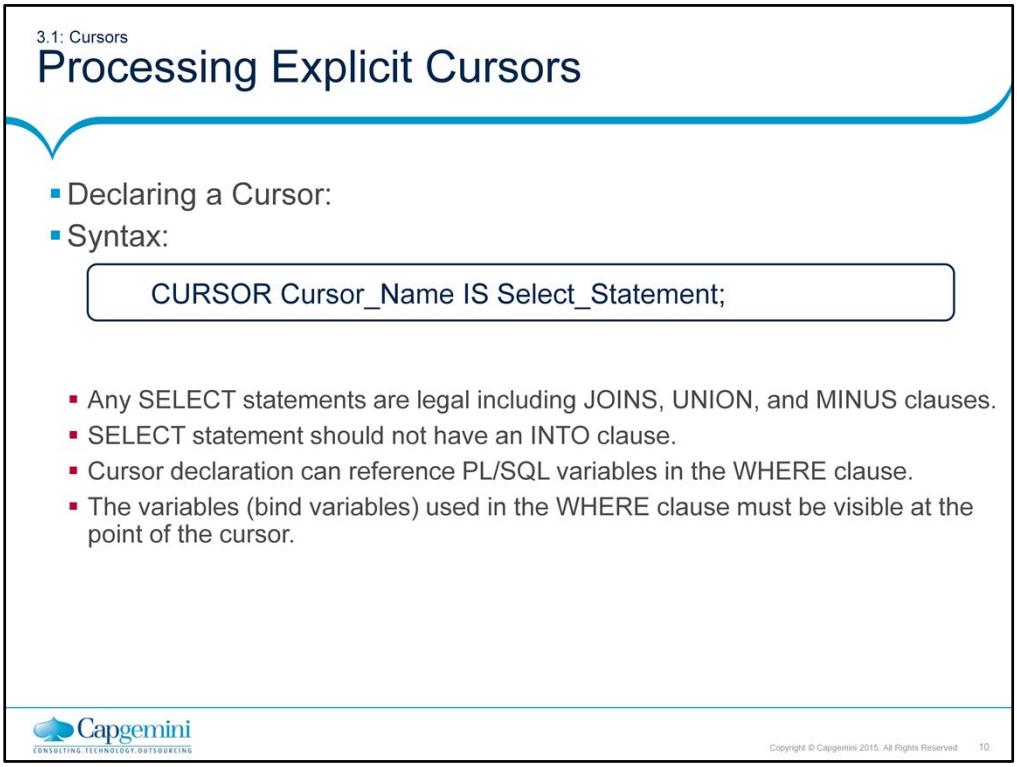


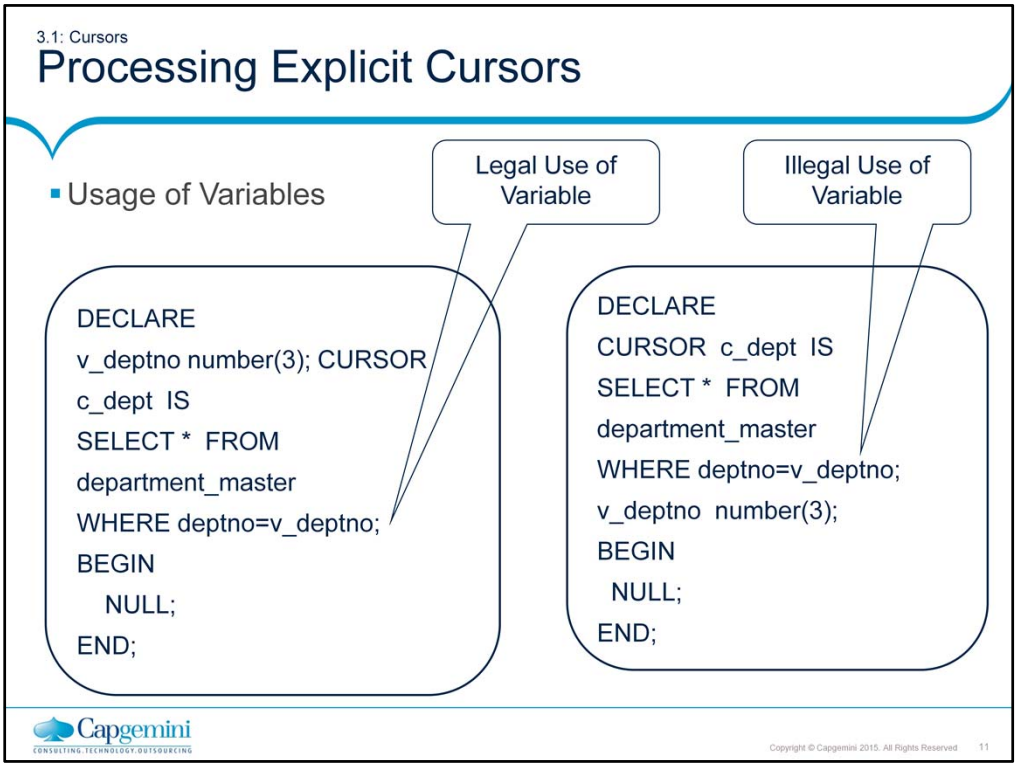
Explicit Cursor:
When you need precise control over query processing, you can explicitly declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.
This technique requires more code than other techniques such as the implicit cursor FOR loop. But it is beneficial in terms of flexibility. You can:

Process several queries in parallel by declaring and opening multiple cursors.

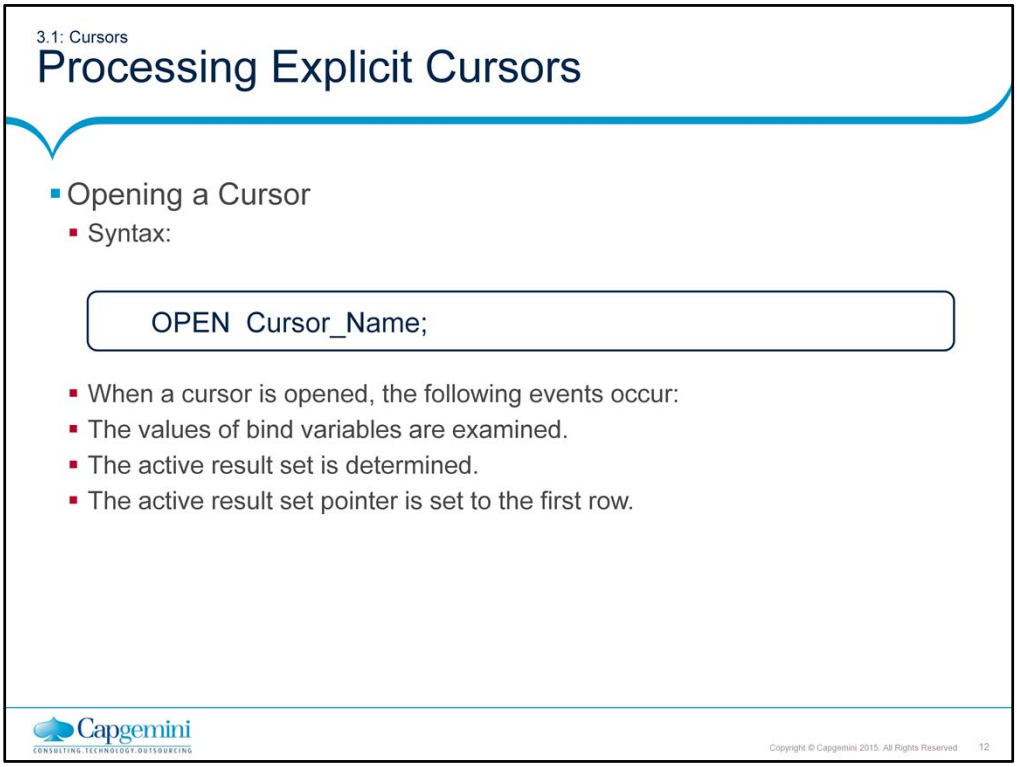
Process multiple rows in a single loop iteration, skip rows, or split the processing into more than one loop.


3.1: Cursors
Processing Explicit Cursors
While processing Explicit Cursors you have to perform the following four steps:
Declare the cursor
Open the cursor for a query
Fetch the results into PL/SQL variables
Close the cursor
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
9


3.1: Cursors
Processing Explicit Cursors
Declaring a Cursor:
Syntax:
CURSOR Cursor_Name IS Select_Statement;
Any SELECT statements are legal including JOINS, UNION, and MINUS clauses.
SELECT statement should not have an INTO clause.
Cursor declaration can reference PL/SQL variables in the WHERE clause.
The variables (bind variables) used in the WHERE clause must be visible at the point of the cursor.
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
10



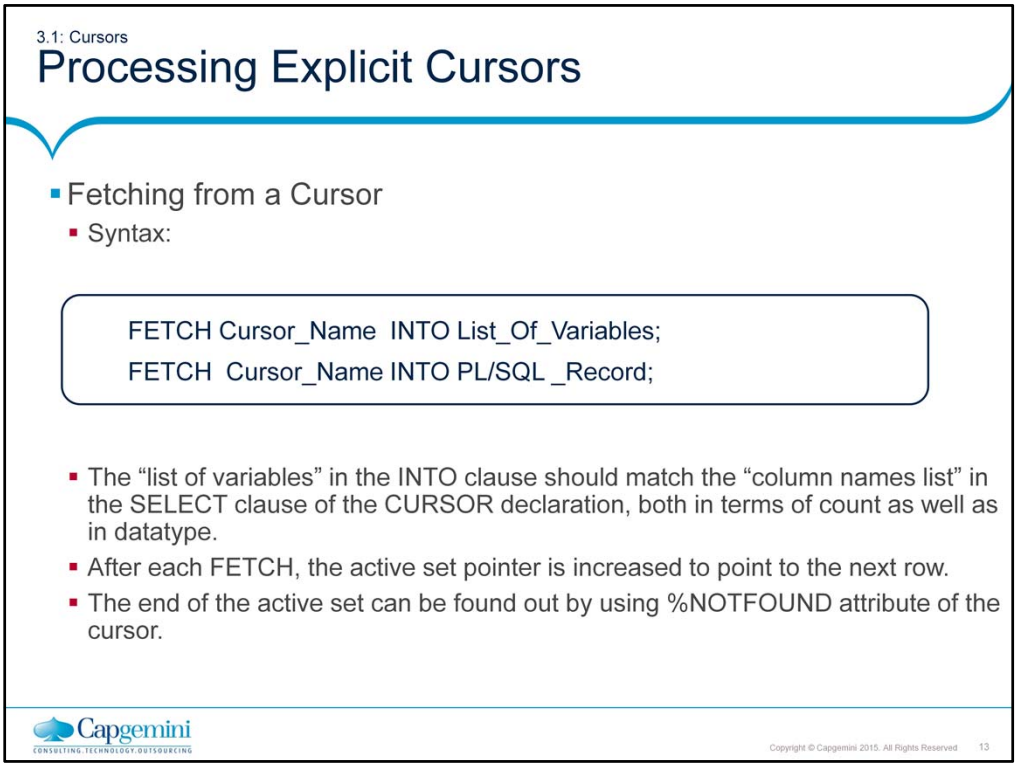
Processing Explicit Cursors: Declaring a Cursor:
The code snippets on the slide show the usage of variables in cursor declaration. You cannot use a variable before it has been declared. It will be illegal.



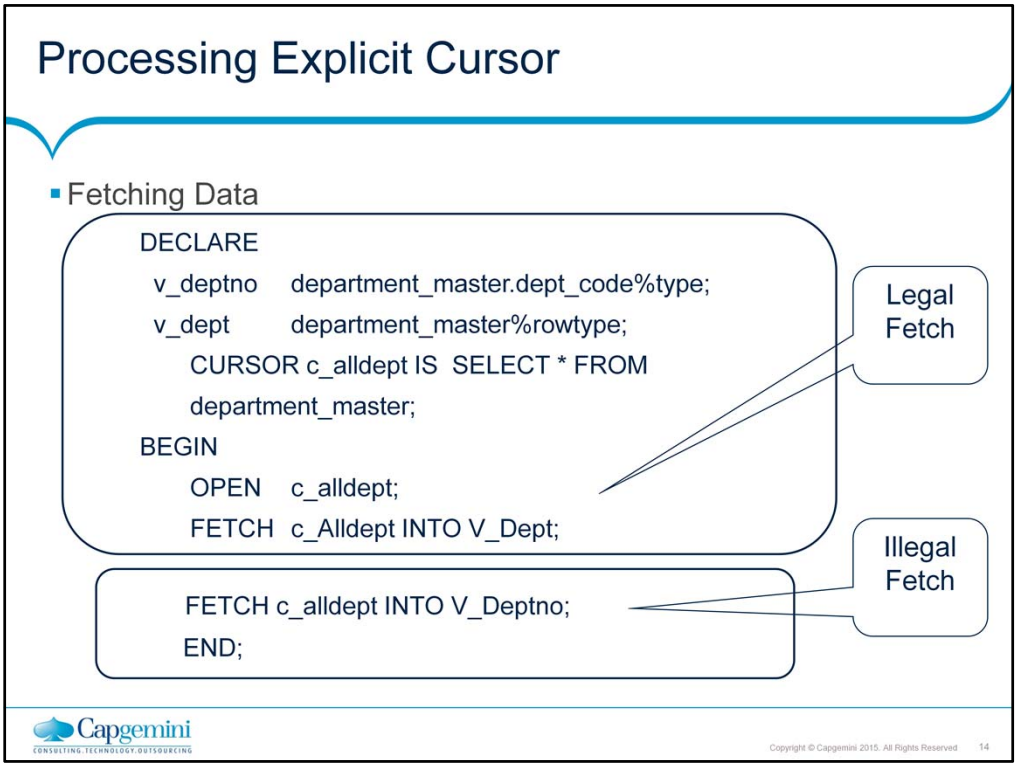
Processing Explicit Cursors: Opening a Cursor:
When a Cursor is opened, the following events occur:
- The values of “bind variables” are examined.
- Based on the values of bind variables , the “active result set” is determined.
- The active result set pointer is set to the first row.

“Bind variables” are evaluated only once at Cursor open time.
- Changing the value of Bind variables after the Cursor is opened will not make any changes to the active result set.
- The query will see changes made to the database that have been committed prior to the OPEN statement.

You can open more than one Cursor at a time.



Processing Explicit Cursors: Fetching from Cursor:



Processing Explicit Cursors: Fetching from Cursor:
The code snippets on the slide shows example of fetching data from cursor. The second snippet FETCH is illegal since SELECT * selects all columns of the table, and there is only one variable in INTO list.
For each column value returned by the query associated with the cursor, there must be a corresponding, type-compatible variable in the INTO list.

To change the result set or the values of variables in the query, you must close and reopen the cursor with the input variables set to their new values.

3.1: Cursors

Processing Explicit Cursors

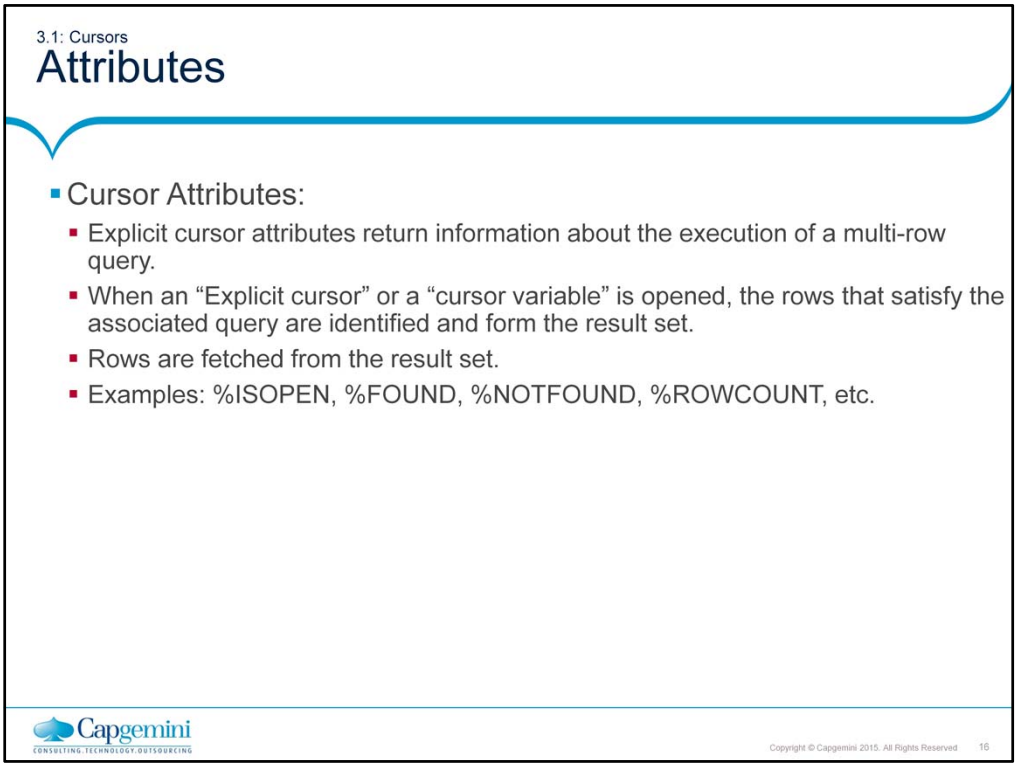
- Closing a Cursor
  - Syntax

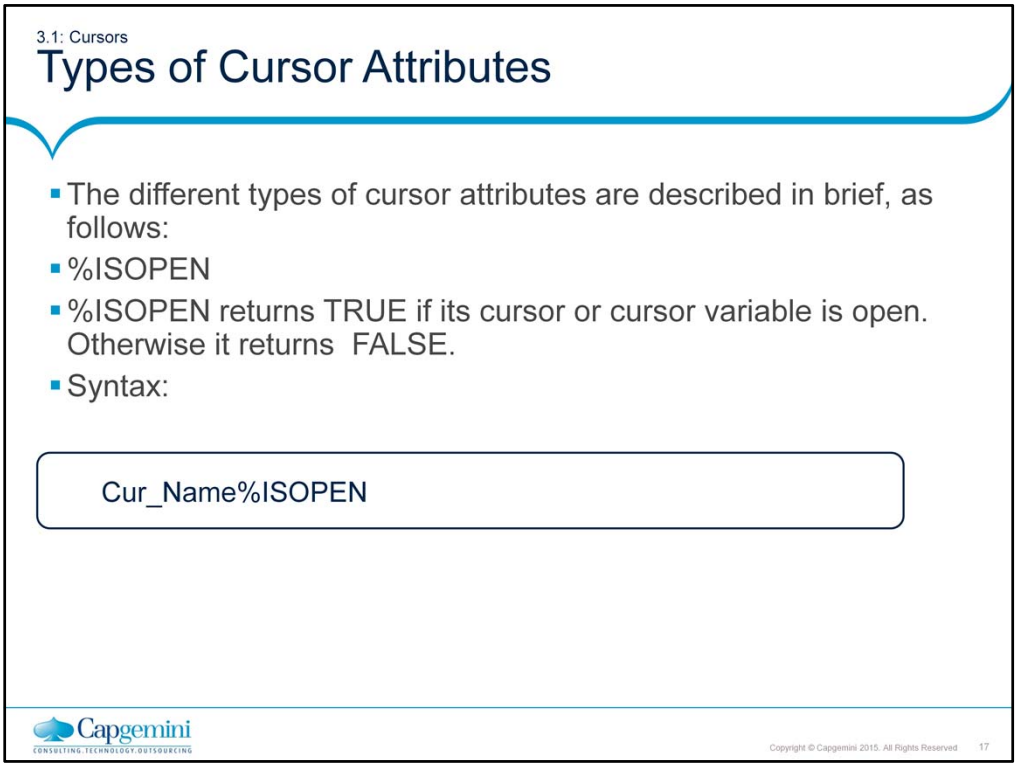
```
CLOSE  Cursor_Name;
```

  - Closing a Cursor frees the resources associated with the Cursor.
  - You cannot FETCH from a closed Cursor.
  - You cannot close an already closed Cursor.

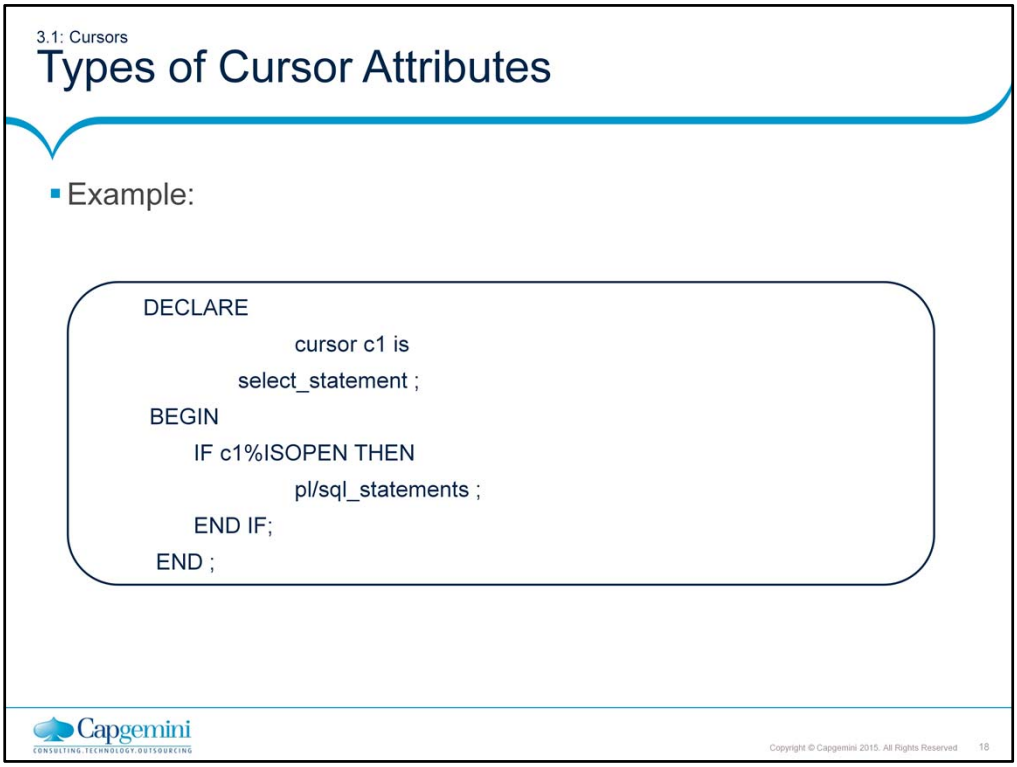
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 15

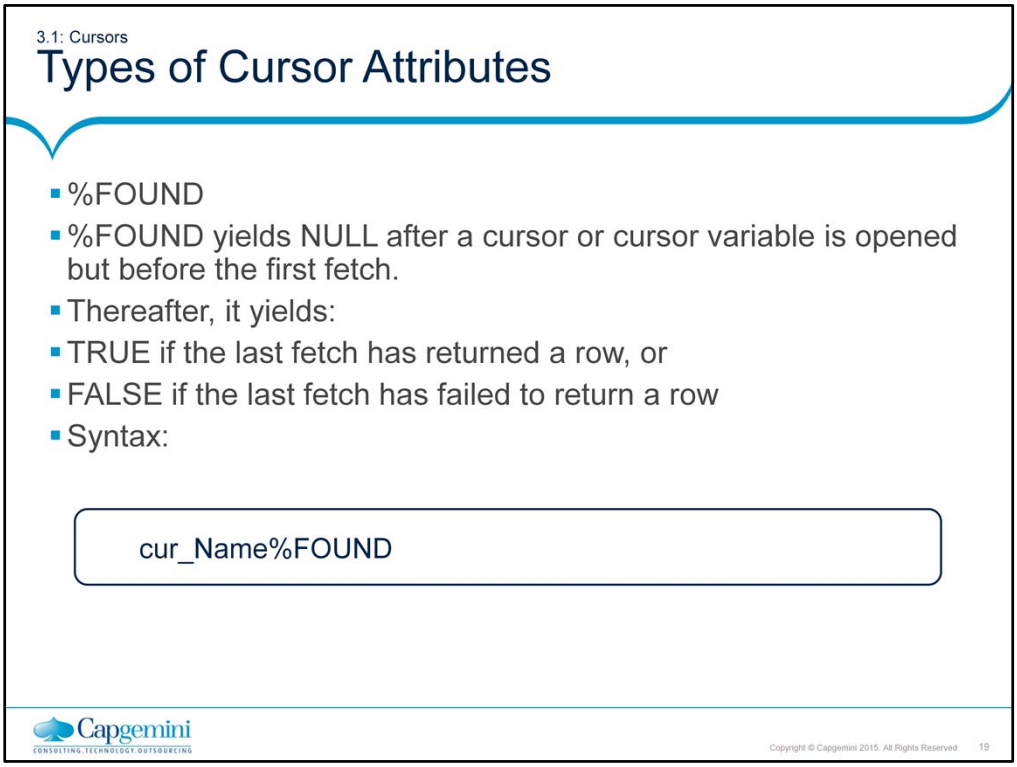

3.1: Cursors
Attributes
Cursor Attributes:
Explicit cursor attributes return information about the execution of a multi-row query.
When an “Explicit cursor” or a “cursor variable” is opened, the rows that satisfy the associated query are identified and form the result set.
Rows are fetched from the result set.
Examples: %ISOPEN, %FOUND, %NOTFOUND, %ROWCOUNT, etc.
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
16



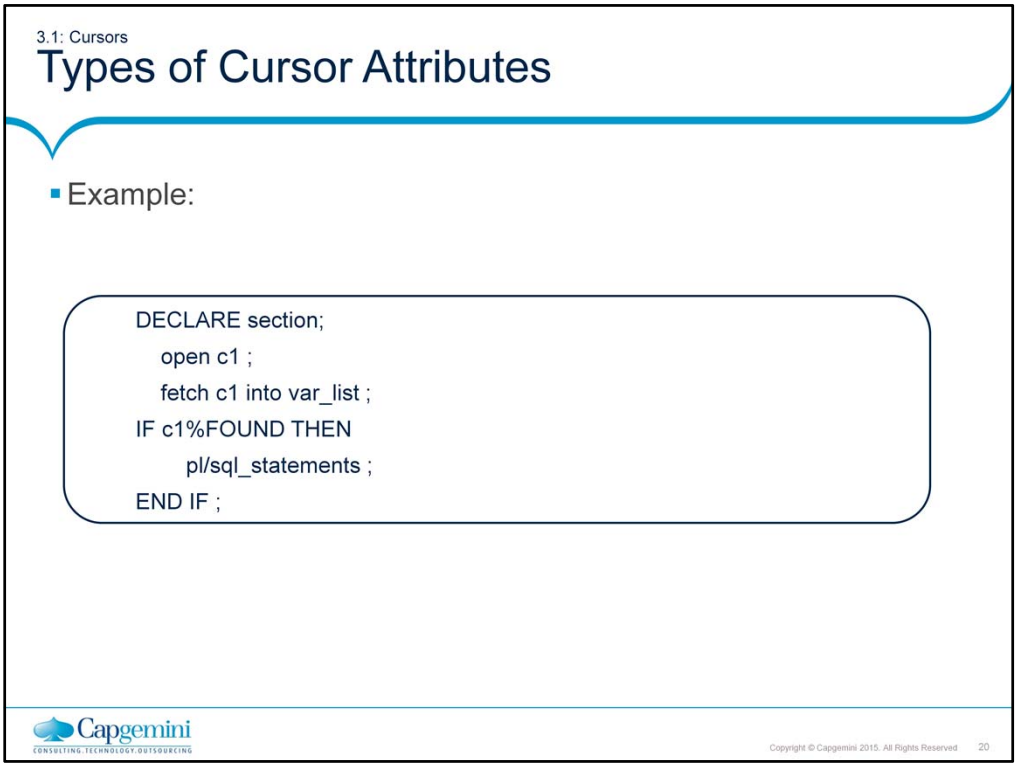
Cursor Attributes: %ISOPEN
This attribute is used to check the open/close status of a Cursor.
If the Cursor is already open, the attribute returns TRUE.
Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %ISOPEN always yields FALSE for Implicit cursor.

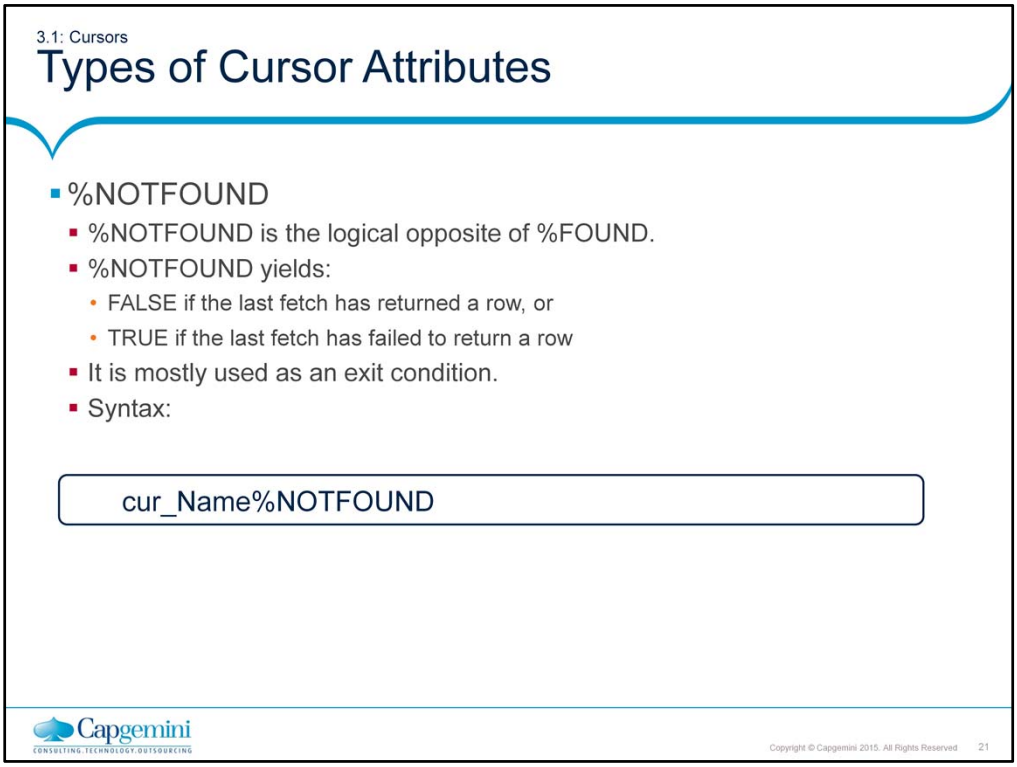


Cursor Attributes: %ISOPEN (contd.)
Note:
In the example shown in the slide, C1%ISOPEN returns FALSE as the cursor is yet to be opened.
Hence, the PL/SQL statements within the IF...END IF are not executed.

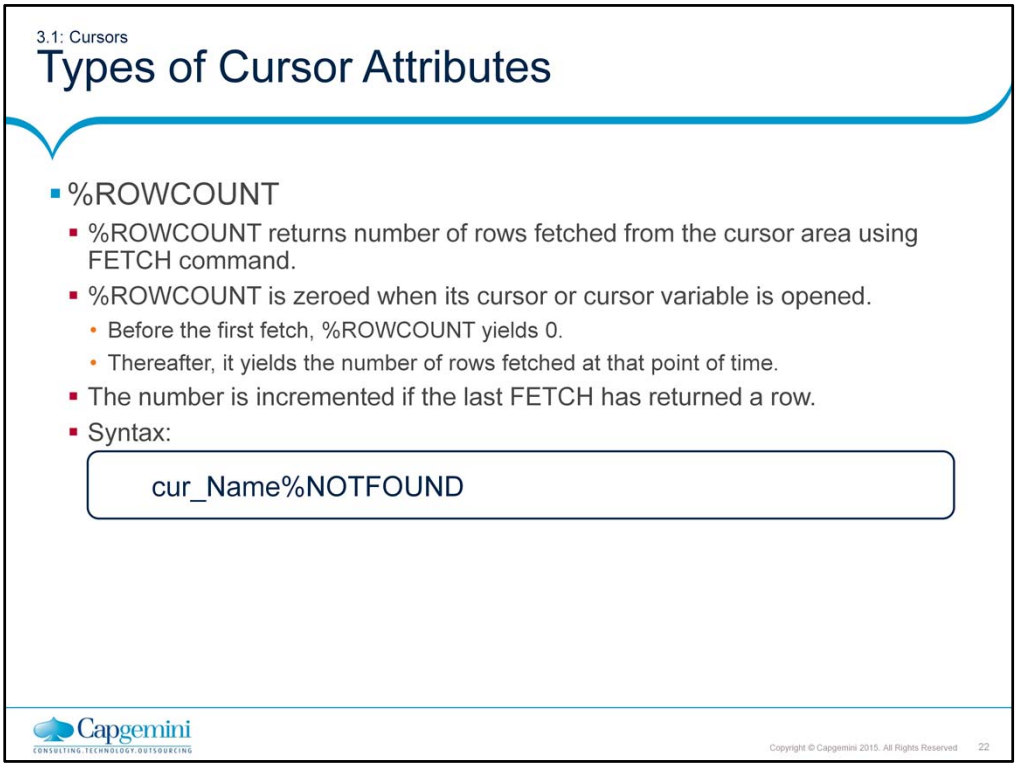


Cursor Attributes: %FOUND:
Until a SQL data manipulation statement is executed, %FOUND yields NULL. Thereafter, %FOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows, or a SELECT INTO statement returned one or more rows. Otherwise, %FOUND yields FALSE


3.1: Cursors
Types of Cursor Attributes
Example:
DECLARE section;
open c1 ;
fetch c1 into var_list ;
IF c1%FOUND THEN
pl/sql_statements ;
END IF ;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
20



Cursor Attributes: %NOTFOUND:
%NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, %NOTFOUND yields FALSE.



Cursor Attributes: %ROWCOUNT
For example: To give a 10% raise to all employees earning less than Rs. 2500.

```
DECLARE
        V_Salary   emp.sal%TYPE;
        V_Empno  emp.empno%TYPE;
        CURSOR C_Empsal  IS
        SELECT empno, sal FROM emp WHERE sal
        <2500;
BEGIN
        IF  NOT C_Empsal%ISOPEN THEN
        OPEN  C_Empsal  ;
        END  IF ;
        LOOP
        FETCH  C_Empsal  INTO
        V_Empno,V_Salary;
        EXIT WHEN  C_Empsal %NOTFOUND ;
                --Exit out of block when no rows
        UPDATE emp SET sal = 1.1 * V_Salary
        WHERE empno = V_Empno;
        END LOOP ;
        CLOSE  C_Empsal ;
        COMMIT ;
END ;
```

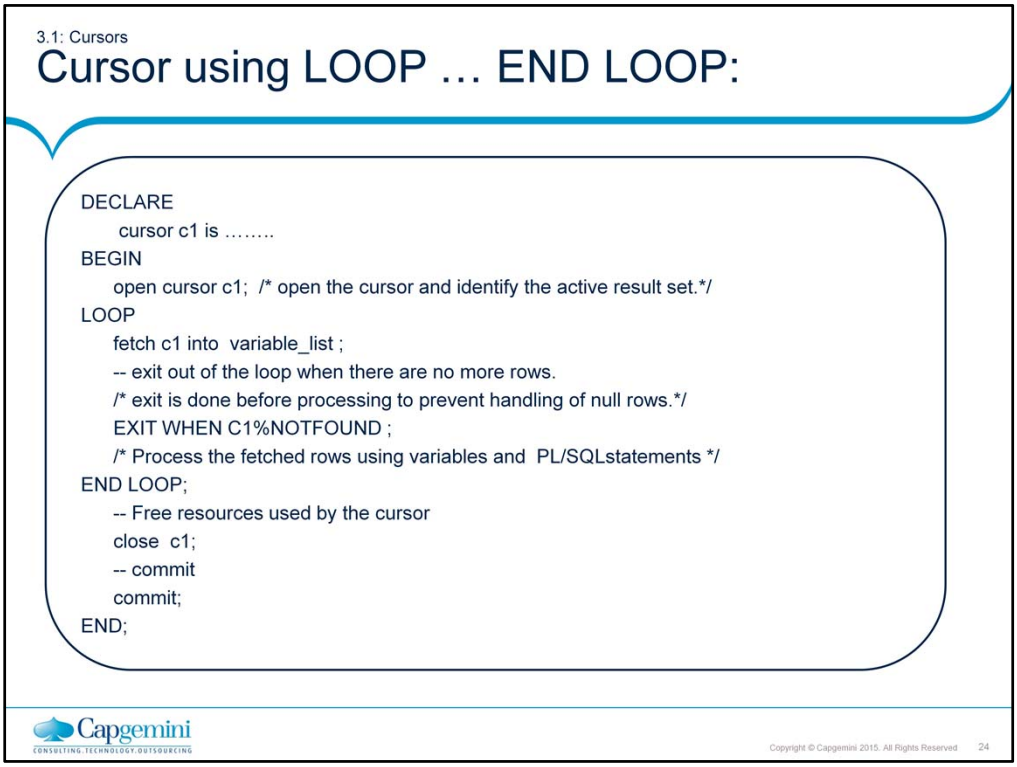
3.1: Cursors

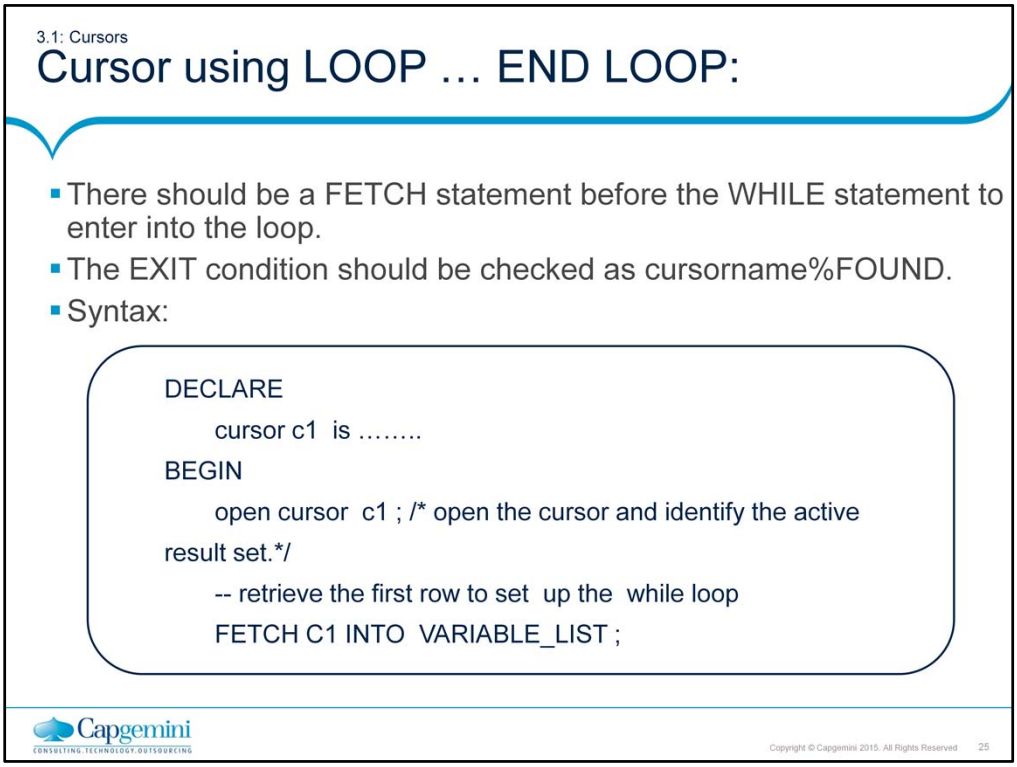
Cursor FETCH loops

- They are examples of simple loop statements.
- The FETCH statement should be followed by the EXIT condition to avoid infinite looping.
- Condition to be checked is cursor%NOTFOUND.
- Examples: LOOP .. END LOOP, WHILE LOOP, etc

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 23


3.1: Cursors
Cursor using LOOP … END LOOP:
DECLARE
cursor c1 is ……..
BEGIN
open cursor c1; /* open the cursor and identify the active result set.*/
LOOP
fetch c1 into variable_list ;
-- exit out of the loop when there are no more rows.
/* exit is done before processing to prevent handling of null rows.*/
EXIT WHEN C1%NOTFOUND ;
/* Process the fetched rows using variables and PL/SQLstatements */
END LOOP;
-- Free resources used by the cursor
close c1;
-- commit
commit;
END;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
24



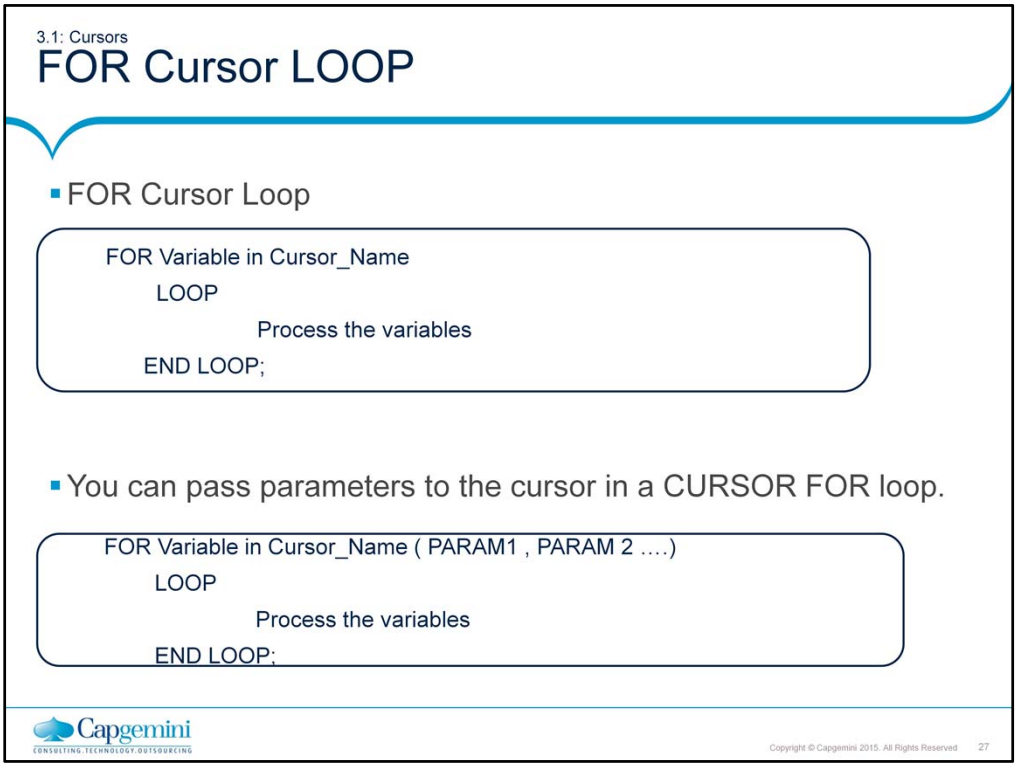
Processing Explicit Cursors: Cursor using WHILE loops:
Note:
FETCH statement appears twice, once before the loop and once after the loop processing.
This is necessary so that the loop condition will be evaluated for each iteration of the loop.

Cursor using WHILE loops…contd

```
/*Continue  looping ,  processing & fetching till last row is
retrieved.*/
WHILE C1%FOUND
LOOP
      fetch c1 into variable_list;
END LOOP;
   CLOSE C1; -- Free resources used by the cursor.
   commit;   -- commit
END;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 26



Processing Explicit Cursors: FOR CURSOR Loop:
For all other loops we had to go through all the steps of OPEN, FETCH, AND CLOSE statements for a cursor.
PL/SQL provides a shortcut via a CURSOR FOR Loop. The CURSOR FOR Loop implicitly handles the cursor processing.

```
DECLARE
CURSOR  C1  IS ……..
BEGIN
-- An implict Open  of the  cursor C1 is done here.
-- Record variable should not be declared in
declaration section
FOR  Record_Variable   IN  C1 LOOP
-- An implicit Fetch is done here
-- Process the fetched rows using variables  and
PL/SQL statements
-- Before the loop is continued an implicit
C1%NOTFOUND test is done by PL/SQL
END LOOP;
-- An automatic close  C1 is issued after termination of
the loop
-- Commit
COMMIT ;
END;
/
```

In this snippet, the record variable is implicitly declared by PL/SQL and is of the type C1%ROWTYPE and the scope of Record_Variable is only for the cursor FOR LOOP.

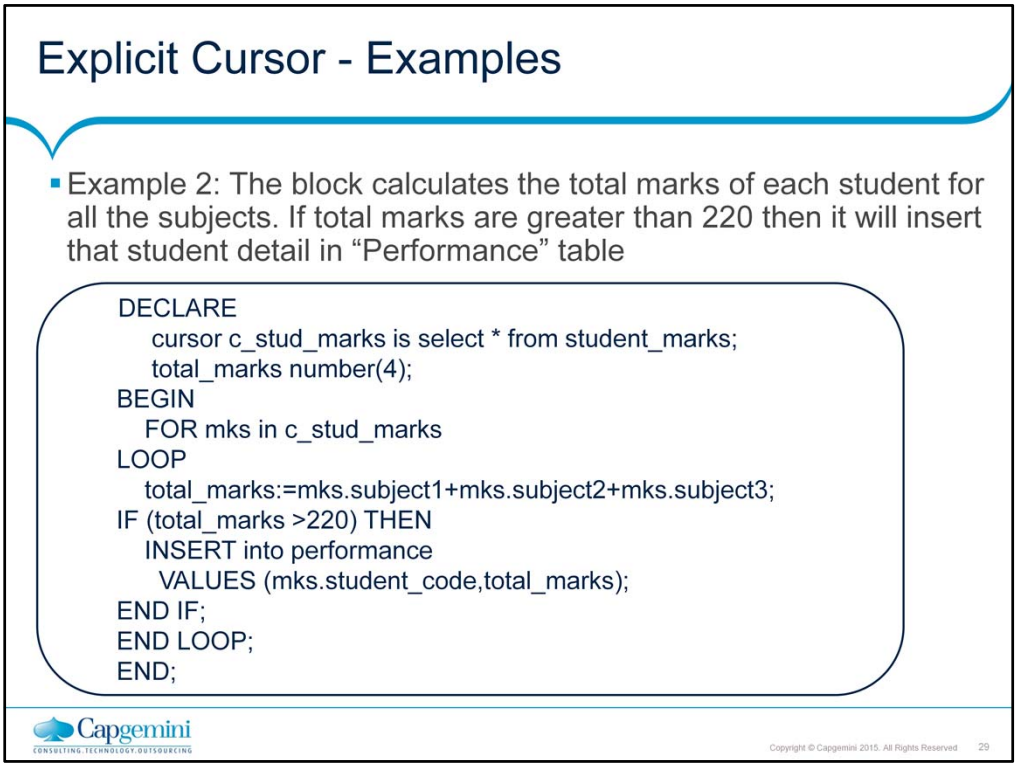
Explicit Cursor - Examples

- Example 1: To add 10 marks in subject3 for all students

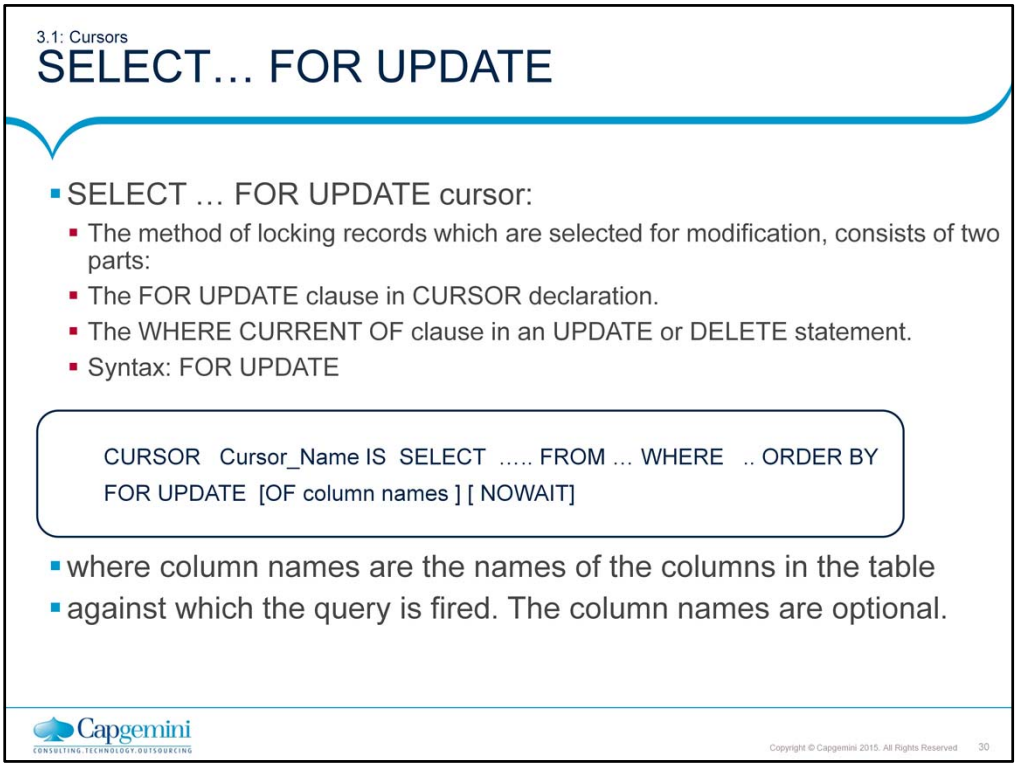
```
DECLARE
      v_sno      student_marks.student_code%type;
      cursor c_stud_marks is
     select student_code from student_marks;
BEGIN
      OPEN c_stud_marks;
      FETCH c_stud_marks into v_sno;
      WHILE c_stud_marks%found
        LOOP
          UPDATE student_marks SET subject3 =subject3+10
          WHERE student_code = v_sno ;
   FETCH c_stud_marks into v_empno;
   END LOOP ;
   CLOSE c_stud_marks;
END;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 28



In the above example “Performance” is a user defined table.



Processing Explicit Cursors: Using FOR UPDATE

The FOR UPDATE clause is part of the SELECT statement of the cursor. It is the last clause of the SELECT statement after the ORDER BY clause (if present). Normally, a SELECT operation does not lock the rows being accessed. This allows others to change the data selected by the SELECT statement. Besides, at OPEN time the active set consists of changes which were committed. Any changes made after OPEN even if they are committed, are not reflected in the active result set unless the cursor is reopened. This results in Data Inconsistency.

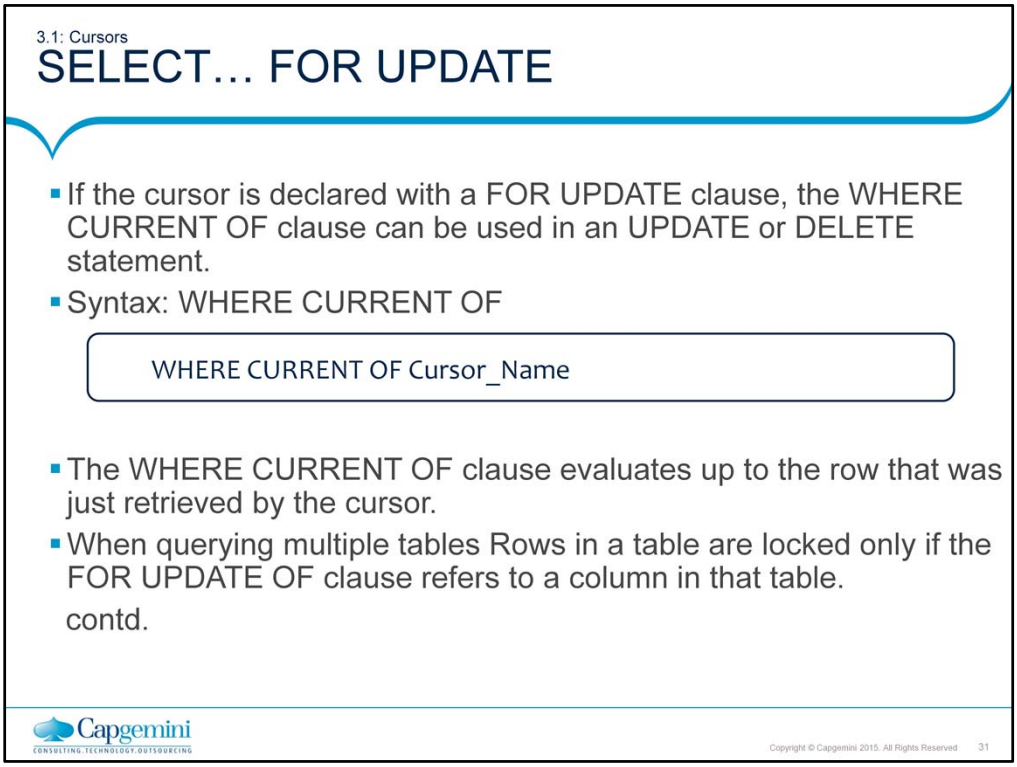
If the FOR UPDATE clause is present, exclusive “row locks” are taken on the rows in the active set.

These locks prevent other sessions / users from changing these rows unless the changes are committed.

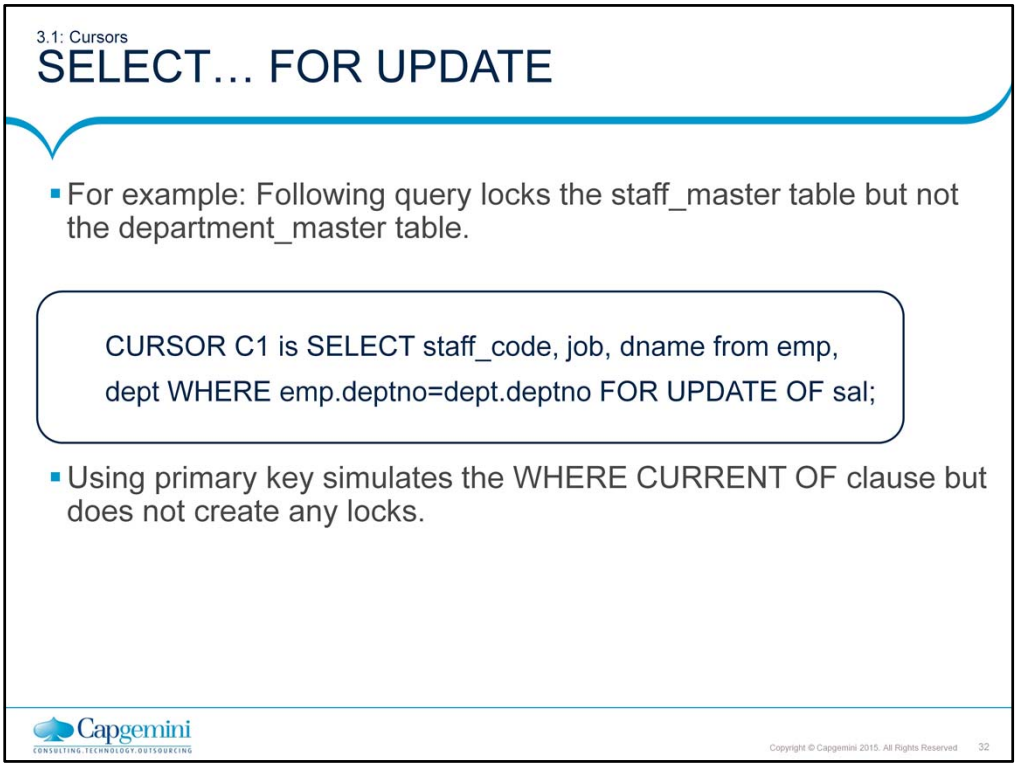
If another session / user already has locks on the rows in the active set, then the SELECT FOR UPDATE will wait indefinitely for these locks to be released. This statement will hang the system till the locks are released.

To avoid this NOWAIT clause is used.

With the NOWAIT clause SELECT FOR UPDATE will not wait for the locks acquired by previous sessions to be released and will return immediately with an ORACLE error.



Processing Cursors: Using WHERE CURRENT OF:
Note:
When querying multiple tables you can use the FOR UPDATE OF column to confine row locking for a particular table.



Processing Cursors: Using WHERE CURRENT OF (contd.):
Note:
If you are using NOWAIT clause, then OF Column_List is essential for syntax purpose.
Any COMMIT should be done after the processing is over in a CURSOR LOOP which has FOR UPDATE clause. This is because after commit locks will be released, the cursor will become invalid, and further FETCH will result in an error. This problem can be solved by using primary key. Using primary key simulates the WHERE CURRENT of clause, however it does not create any locks.

3.1: Cursors

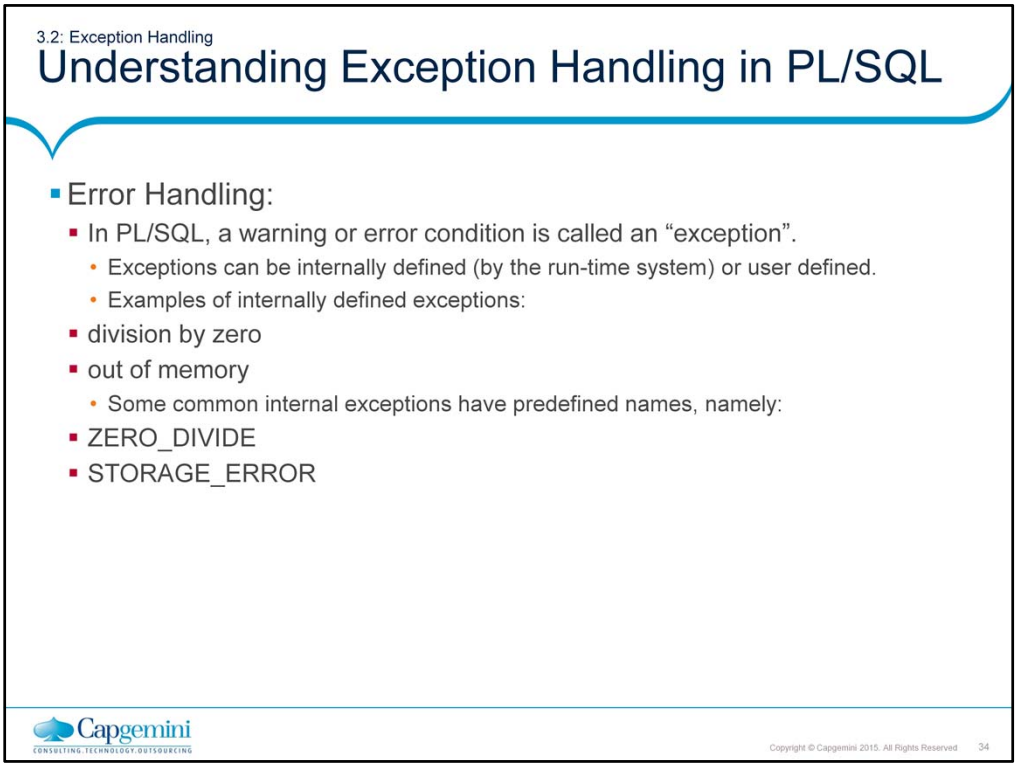
SELECT… FOR UPDATE - Examples

- To promote professors who earn more than 20000

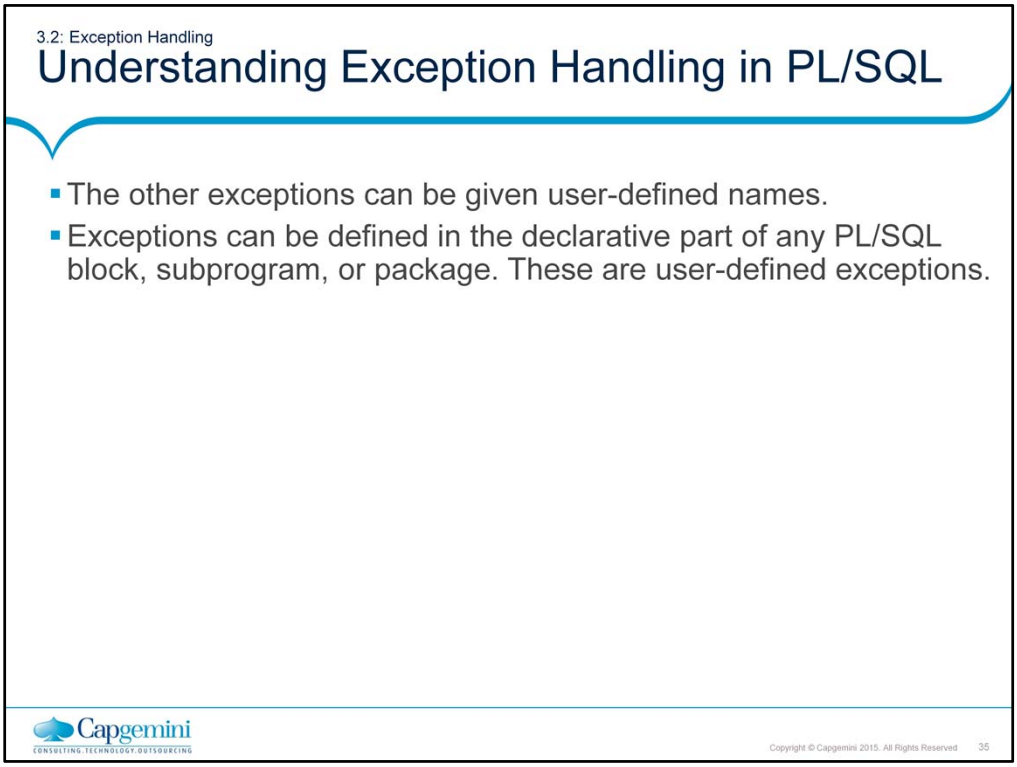
```
DECLARE
CURSOR c_staff  is SELECT staff_code, staff_master.design_code
 FROM staff_master,designation_master
 WHERE design_name = 'Professor' and staff_sal > 20000
and staff_master.design_code =designation_master.design_code
FOR UPDATE OF design_code NOWAIT;
d_code  designation_master.design_code%type;
BEGIN
      SELECT design_code into d_code FROM designation_master
           WHERE design_name='Director';
          FOR v_rec in c_staff
          LOOP
              UPDATE staff_master SET design_code = d_code
               WHERE current of c_staff;
          END LOOP;
  END;
```

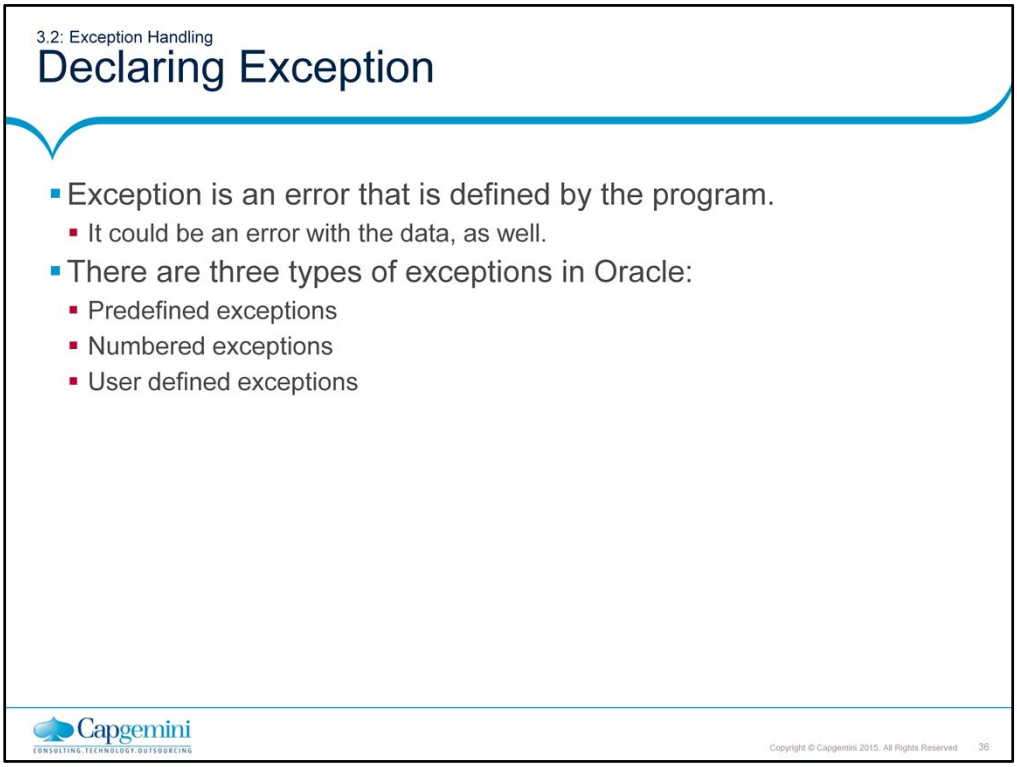
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 33

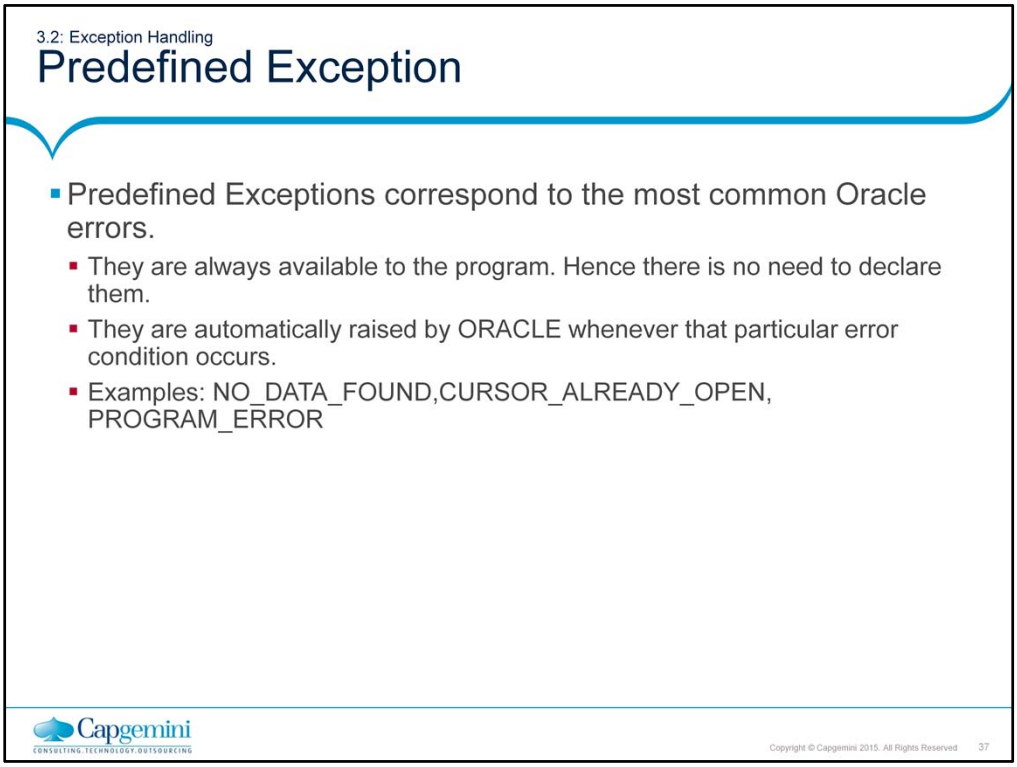


Error Handling:
A good programming language should provide capabilities of handling errors and recovering from them if possible.
PL/SQL implements Error Handling via “exceptions” and “exception handlers”.
Types of Errors in PL/SQL
Compile Time errors: They are reported by the PL/SQL compiler, and you have to correct them before recompiling.
Run Time errors: They are reported by the run-time engine. They are handled programmatically by raising an exception, and catching it in the Exception section.


3.2: Exception Handling
Understanding Exception Handling in PL/SQL
The other exceptions can be given user-defined names.
Exceptions can be defined in the declarative part of any PL/SQL block, subprogram, or package. These are user-defined exceptions.
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
35



Declaring Exceptions:
Exceptions are declared in the Declaration section, raised in the Executable section, and handled in the Exception section.



Predefined Exceptions:
An internal exception is raised implicitly whenever your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. So, PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception NO_DATA_FOUND if a SELECT INTO statement returns no rows.
Given below are some Predefined Exceptions:

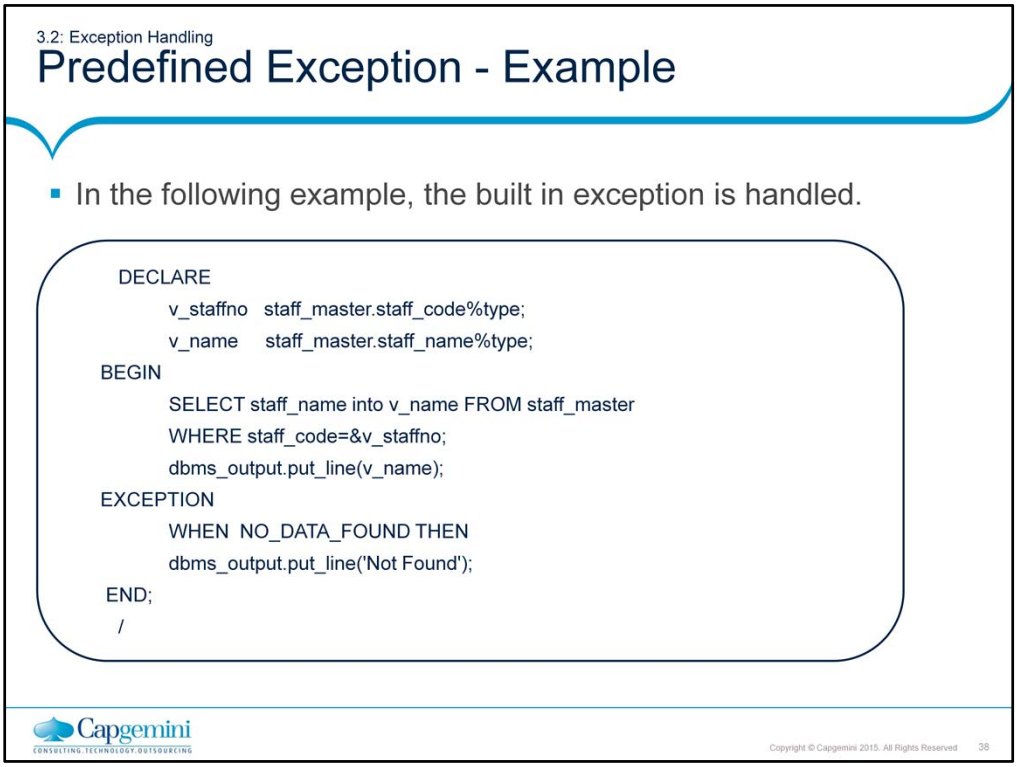
NO_DATA_FOUND
: This exception is raised when SELECT INTO …. statement does not return any rows.

TOO_MANY_ROWS
: This exception is raised when SELECT INTO …. statement returns more than one row.

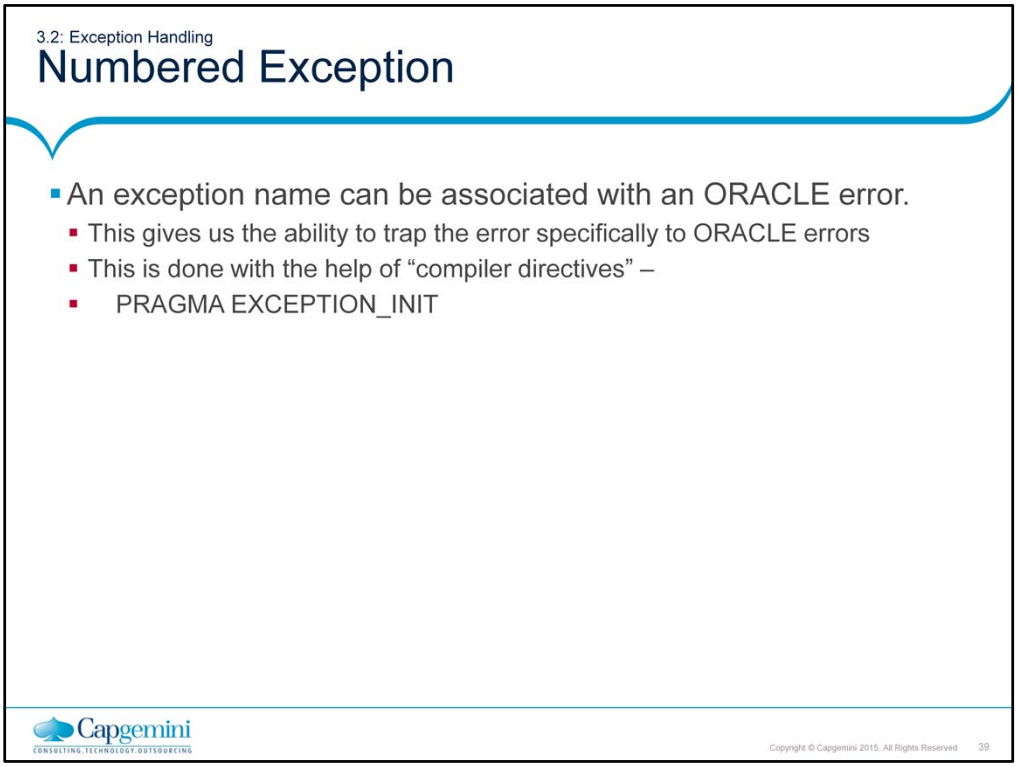
INVALID_CURSOR
: This exception is raised when an illegal cursor operation is performed such as closing an already closed cursor.

VALUE_ERROR
: This exception is raised when an arithmetic, conversion, truncation, or constraint error occurs in a procedural statement.

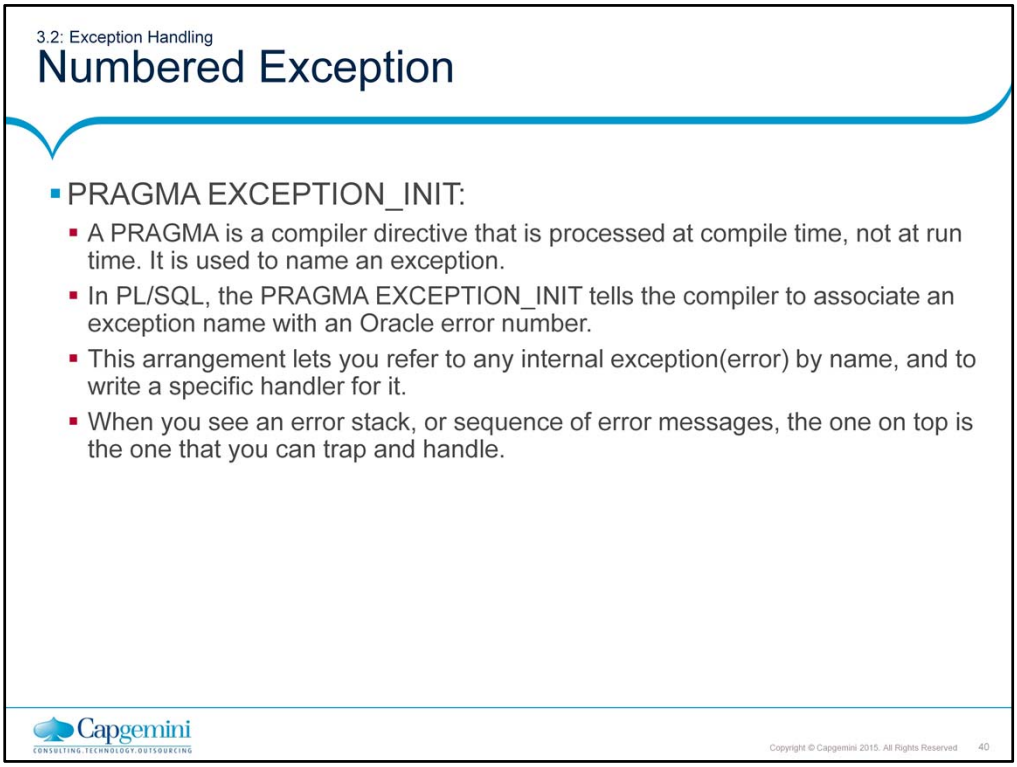
DUP_VAL_ON_INDEX
: This exception is raised when the UNIQUE CONSTRAINT is violated.



Predefined Exceptions:
In the example shown on the slide, the NO_DATA_FOUND built in exception is handled. It is automatically raised if the SELECT statement does not fetch any value and populate the variable.



Numbered Exception:
The Numbered Exceptions are Oracle errors bound to a user defined exception name.


3.2: Exception Handling
Numbered Exception
PRAGMA EXCEPTION_INIT:
A PRAGMA is a compiler directive that is processed at compile time, not at run time. It is used to name an exception.
In PL/SQL, the PRAGMA EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number.
This arrangement lets you refer to any internal exception(error) by name, and to write a specific handler for it.
When you see an error stack, or sequence of error messages, the one on top is the one that you can trap and handle.
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
40

3.2: Exception Handling

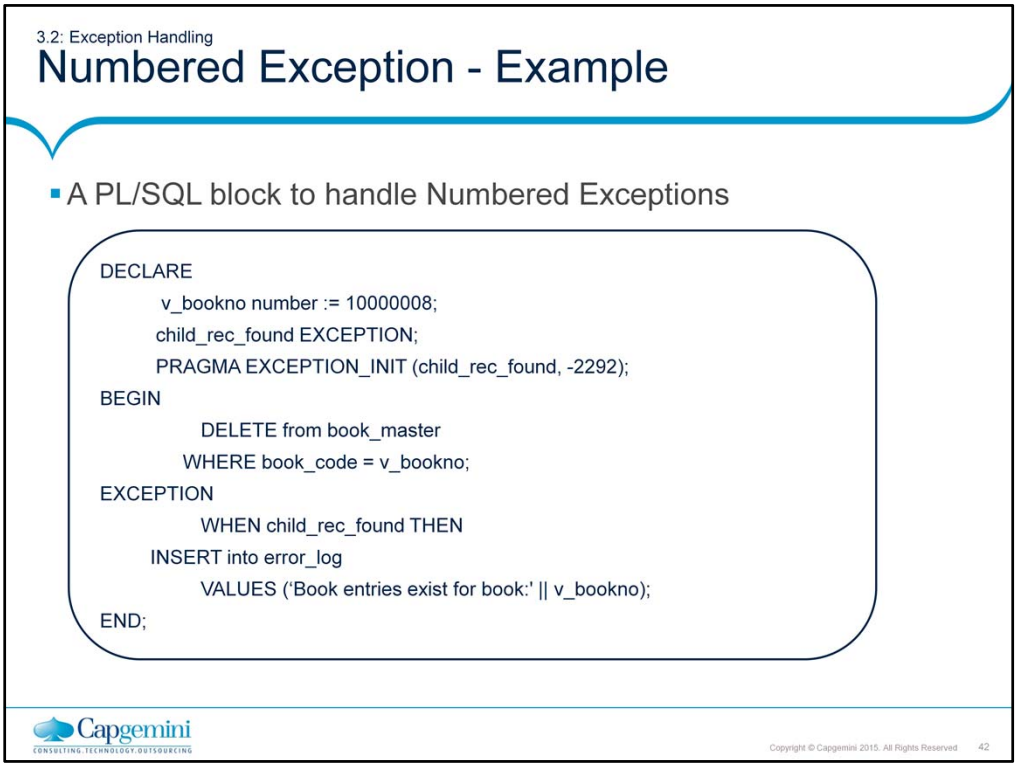
Numbered Exception (Contd.)

- User defined exceptions can be named with error number between -20000 and -20999.
- The naming is declared in Declaration section.
- It is valid within the PL/SQL blocks only.
- Syntax is:

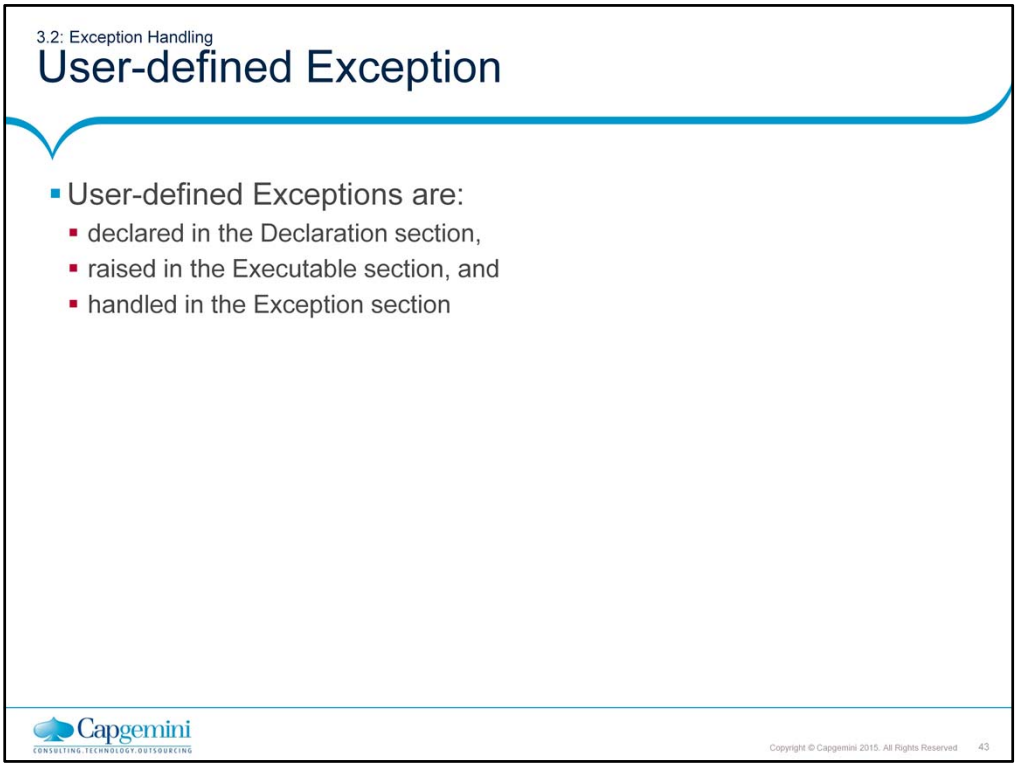
```
PRAGMA EXCEPTION_INIT(Exception Name,Error_Number);
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 41



If a user tries to delete record from the parent table wherein child records exist an error is raised by Oracle. We would want to handle this error through the PL/SQL block which is deleting records from a parent table. The example on the slide demonstrates this. In the PL/SQL block we are binding the constraint exception raised by Oracle to user defined exception name.
All oracle errors are negative i.e prefixed with a minus symbol.
In the example we are mapping error-2292 which occurs when referential integrity rule is violated.



User-Defined Exceptions:
These exception are entirely user defined based on the application. The programmer is responsible for declaring, raising and handling them.

3.2: Exception Handling

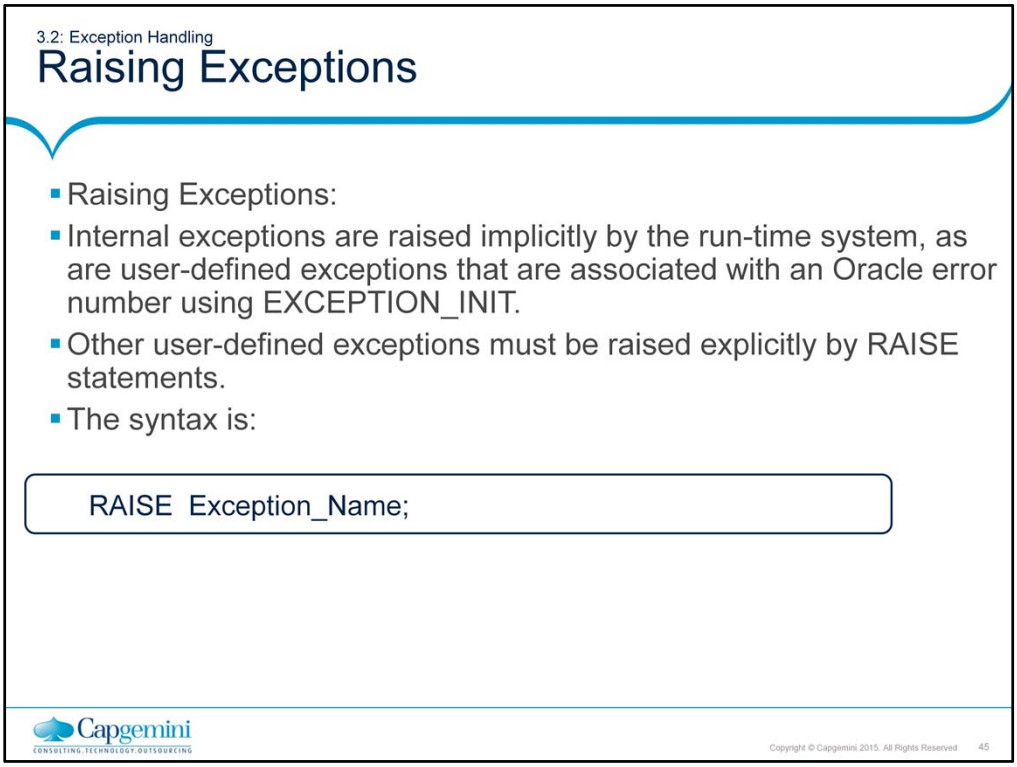
User-defined Exception - Example

- Here is an example of User Defined Exception:

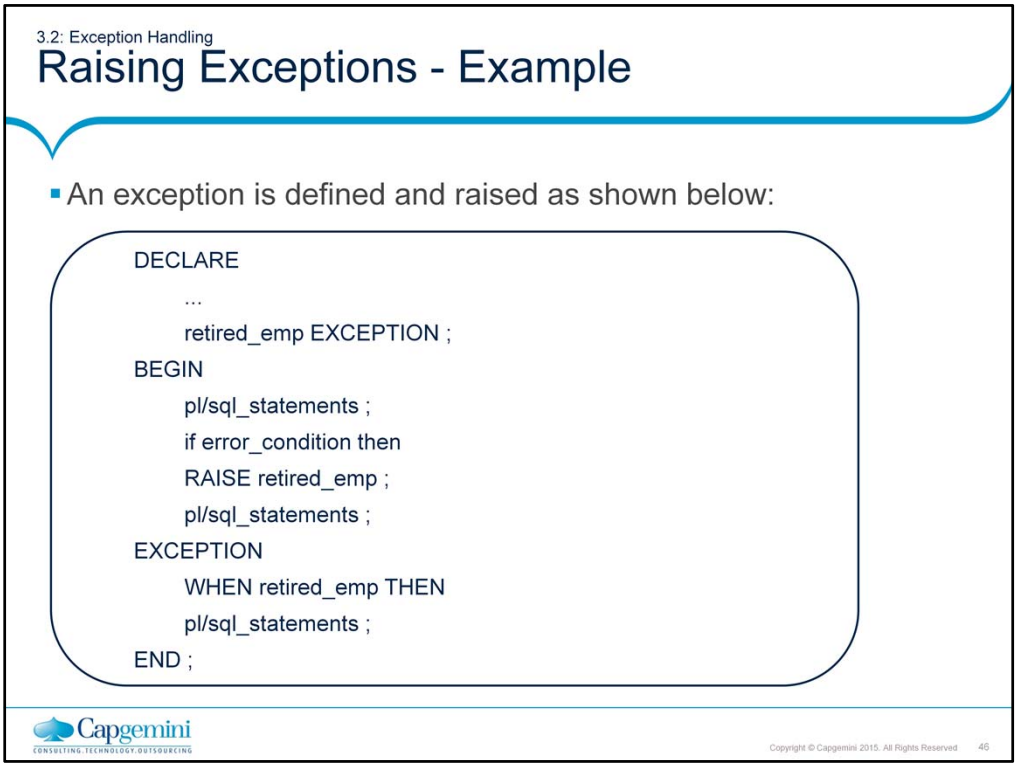
```
DECLARE
    E_Balance_Not_Sufficient EXCEPTION;
    E_Comm_Too_Larage EXCEPTION;
    …
BEGIN
    NULL;
END;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 44



Raising Exceptions:
When the error associated with an exception occurs, the exception is raised. This is done through the RAISE command.


3.2: Exception Handling
Raising Exceptions - Example
An exception is defined and raised as shown below:
DECLARE
...
retired_emp EXCEPTION ;
BEGIN
pl/sql_statements ;
if error_condition then
RAISE retired_emp ;
pl/sql_statements ;
EXCEPTION
WHEN retired_emp THEN
pl/sql_statements ;
END ;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
46

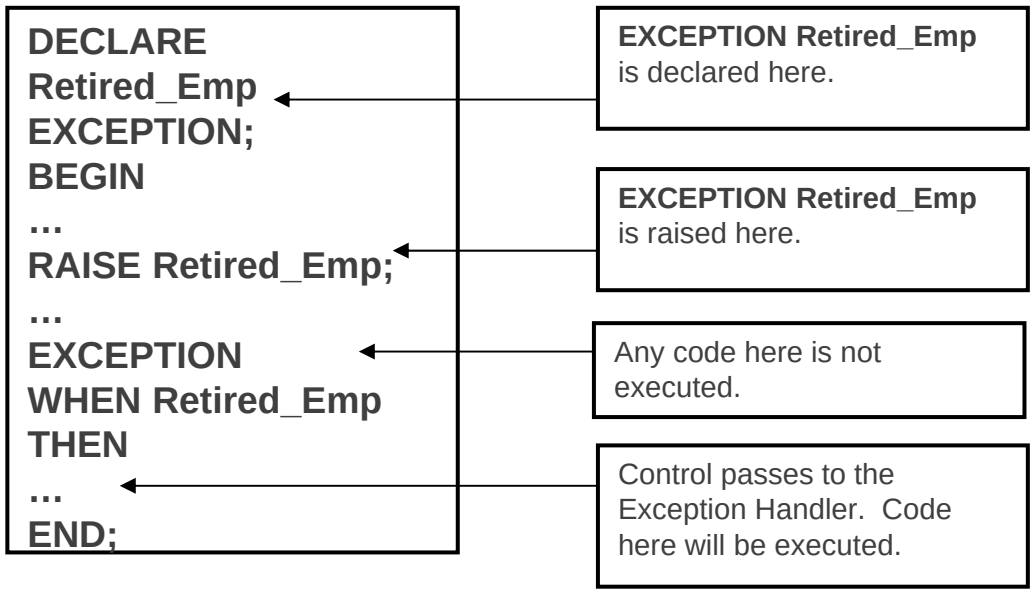
Control passing to Exception Handler

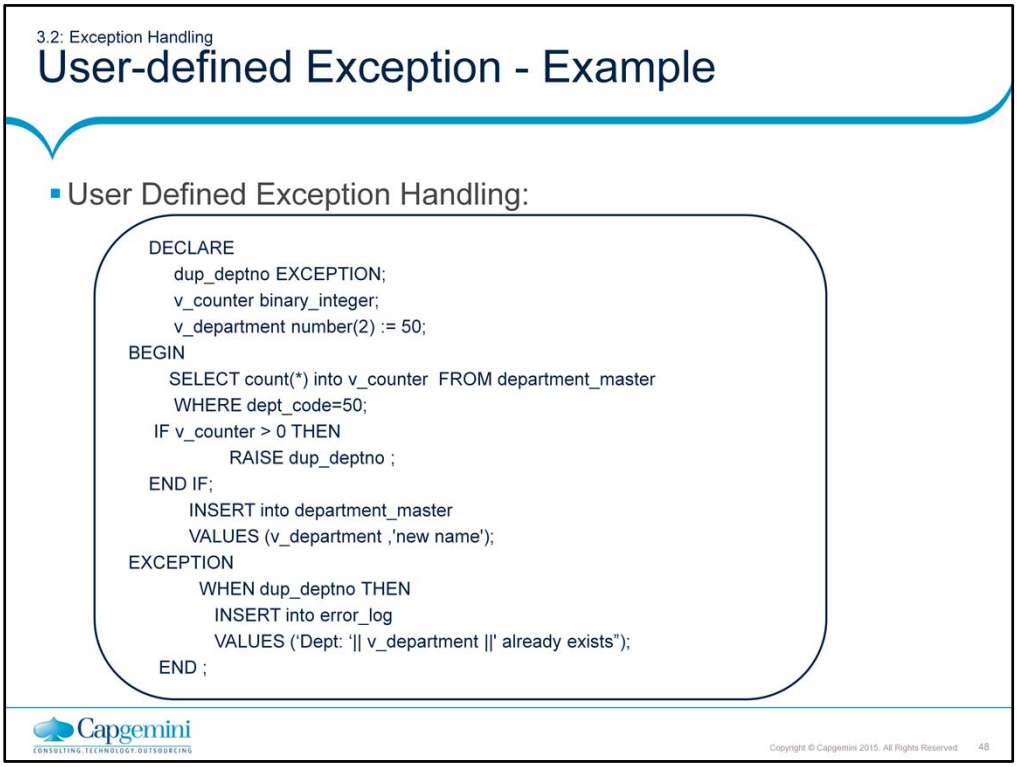
- Control passing to Exception Handler :
  - When an exception is raised, normal execution of your PL/SQL block or subprogram stops, and control passes to its exception-handling part.
  - To catch the raised exceptions, you write “exception handlers”.
  - Each exception handler consists of a WHEN clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised.
  - These statements complete execution of the block or subprogram, however, the control does not return to where the exception was raised. In other words, you cannot resume processing where you left off.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 47

Control passing to Exception Handler:





The example on the slide demonstrates user-defined exceptions. It checks for department no value to be inserted in the table. If the value is duplicated it will raise an exception.

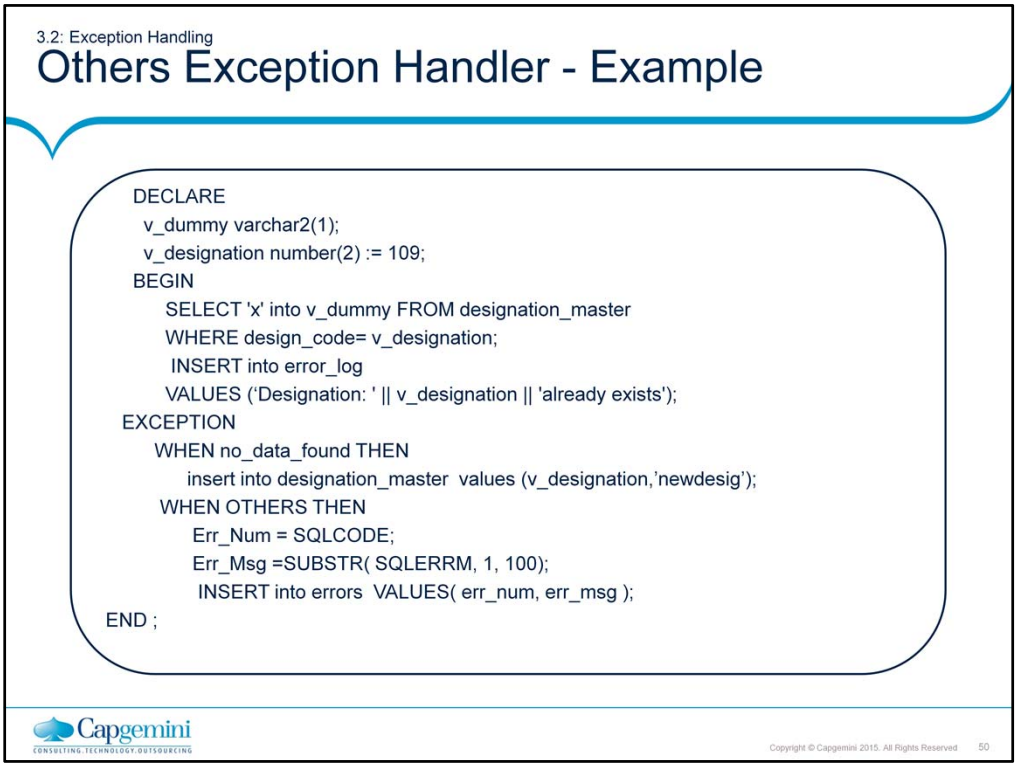
3.2: Exception Handling

Others Exception Handler

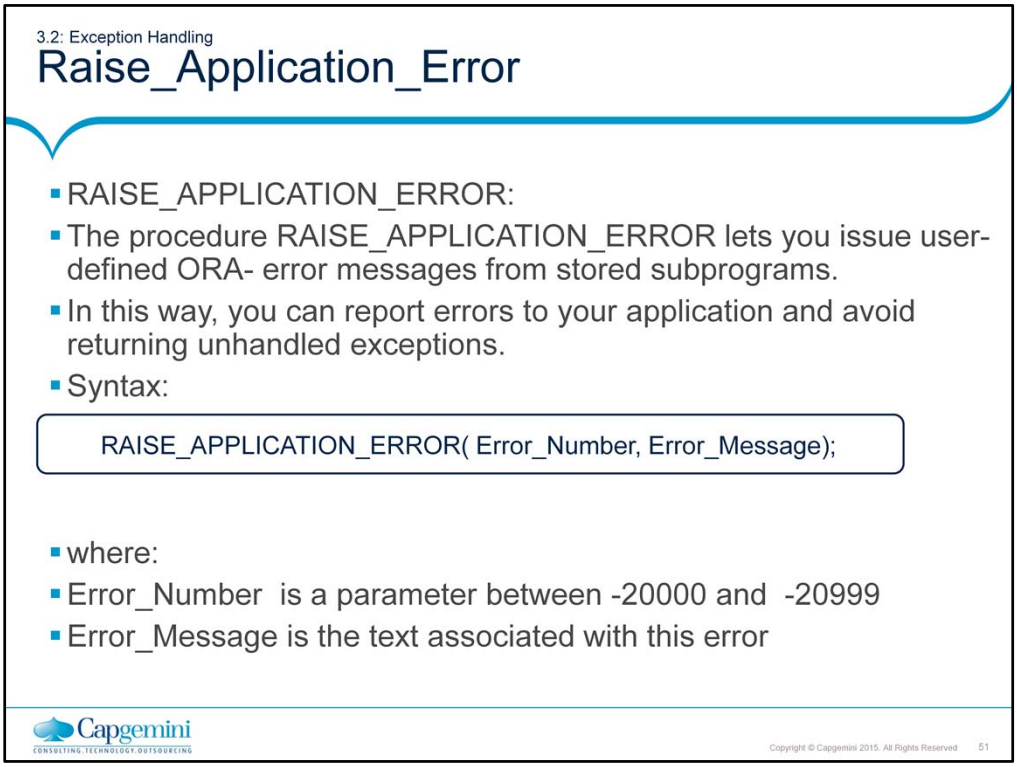
- OTHERS Exception Handler:
  - The optional OTHERS exception handler, which is always the last handler in a block or subprogram, acts as the handler for all exceptions that are not specifically named in the Exception section.
  - A block or subprogram can have only one OTHERS handler.
  - To handle a specific case within the OTHERS handler, predefined functions SQLCODE and SQLERRM are used.
  - SQLCODE returns the current error code. And SQLERRM returns the current error message text.
  - The values of SQLCODE and SQLERRM should be assigned to local variables before using it within a SQL statement.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 49



The example on the slide uses OTHERS Exception handler. If the exception that is raised by the code is not NO_DATA_FOUND, then it will go to the OTHERS exception handler since it will notice that there is no appropriate exception handler defined.
Also observe that the values of SQLCODE and SQLERRM are assigned to variables defined in the block.



Raise Application Error:
The built-in function RAISE_APPLICATION_ERROR is used to create our own error messages, which can be more descriptive and user friendly than Exception Names.

3.2: Exception Handling

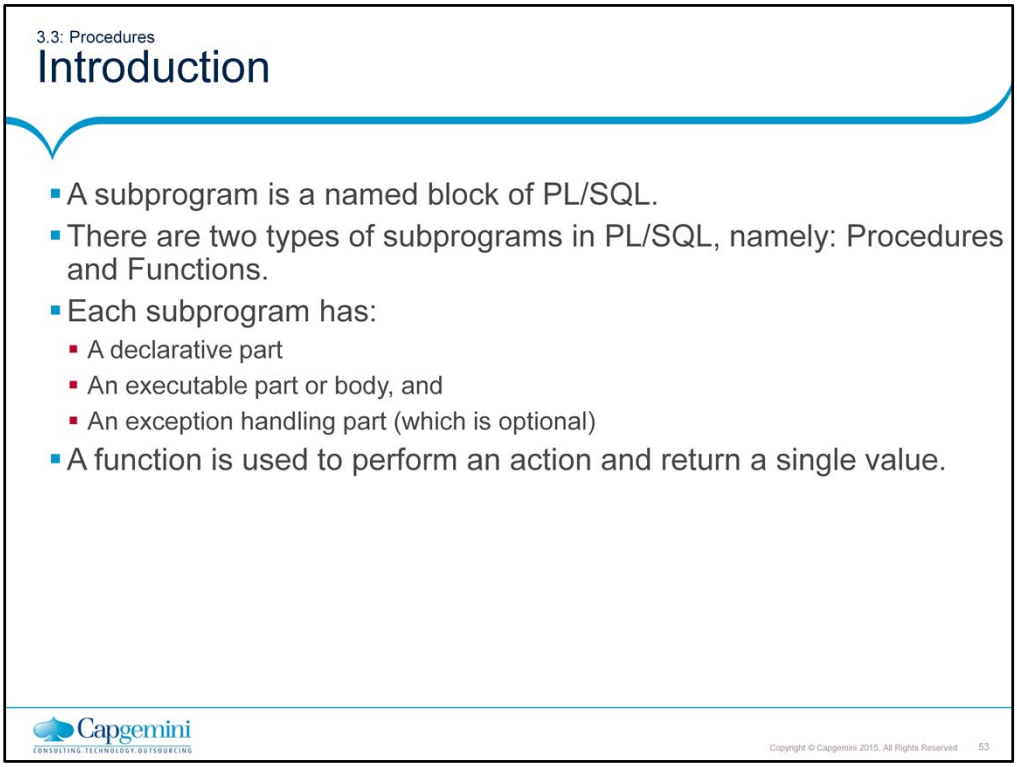
Raise_Application_Error - Example

- Here is an example of Raise Application Error:

```
DECLARE
    /* VARIABLES */
BEGIN
    ……..
    ……..
EXCEPTION
    WHEN OTHERS THEN
    -- Will transfer the error to the calling environment
    RAISE_APPLICATION_ERROR( -20999 ,'Contact DBA');
END ;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 52



Subprograms in PL/SQL:
The subprograms are compiled and stored in the Oracle database as “stored programs”, and can be invoked whenever required. As the subprograms are stored in a compiled form, when called they only need to be executed. Hence this arrangement saves time needed for compilation.
When a client executes a procedure or function, the processing is done in the server. This reduces the network traffic.
Subprograms provide the following advantages:

They allow you to write a PL/SQL program that meets our need.
They allow you to break the program into manageable modules.
They provide reusability and maintainability for the code.

3.3: Procedures

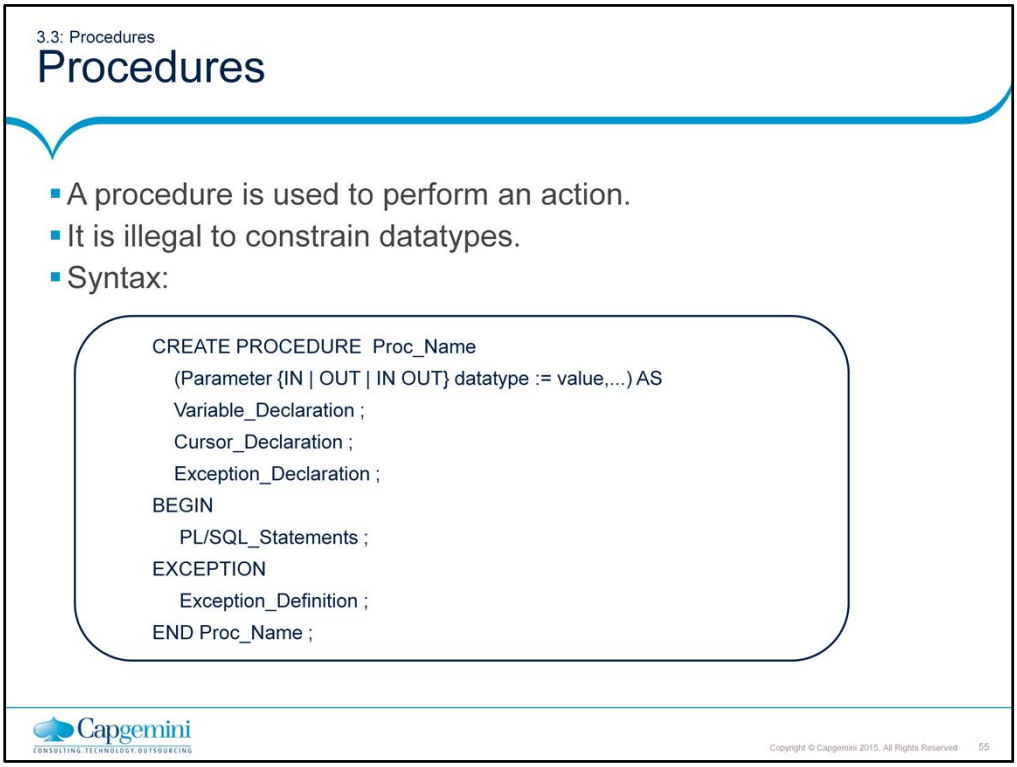
Comparison

- Anonymous Blocks & Stored Subprograms Comparison

Anonymous Blocks	Stored Subprograms/Named Blocks
1. Anonymous Blocks do not have names.	1. Stored subprograms are named PL/SQL blocks.
2. They are interactively executed. The block needs to be compiled every time it is run.	2. They are compiled at the time of creation and stored in the database itself. Source code is also stored in the database.
3. Only the user who created the block can use the block.	3. Necessary privileges are required to execute the block.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 54



Procedures:
A procedure is a subprogram used to perform a specific action.
A procedure contains two parts:
the specification, and
the body
The procedure specification begins with CREATE and ends with procedure name or parameters list. Procedures that do not take parameters are written without a parenthesis.
The procedure body starts after the keyword IS or AS and ends with keyword END.
contd.

3.3: Procedures

Subprogram Parameter Modes

IN	OUT	IN OUT
The default	Must be specified	Must be specified
Used to pass values to the procedure.	Used to return values to the caller.	Used to pass initial values to the procedure and return updated values to the caller.
Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an uninitialized variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression, but should be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, literal, initialized variable, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.
Actual parameter is passed by reference (a pointer to the value is passed in).	Actual parameter is passed by value (a copy of the value is passed out) unless NOCOPY is specified.	Actual parameter is passed by value (a copy of the value is passed in and out) unless NOCOPY is specified.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 56

Subprogram Parameter Modes:
You use “parameter modes” to define the behavior of “formal parameters”. The three parameter modes are IN (the default), OUT, and INOUT. The characteristics of the three modes are shown in the slide.
Any parameter mode can be used with any subprogram.
Avoid using the OUT and INOUT modes with functions.
To have a function return multiple values is a poor programming practice. Besides functions should be free from side effects, which change the values of variables that are not local to the subprogram.
Example1:

```
CREATE PROCEDURE split_name
(
  phrase IN VARCHAR2, first OUT VARCHAR2, last OUT
    VARCHAR2
)
IS
  first := SUBSTR(phrase, 1, INSTR(phrase, ' ')-1);
  last := SUBSTR(phrase, INSTR(phrase, ' ')+1);
  IF first = 'John' THEN
    DBMS_OUTPUT.PUT_LINE('That is a common first
    name.');
  END IF;
END;
```

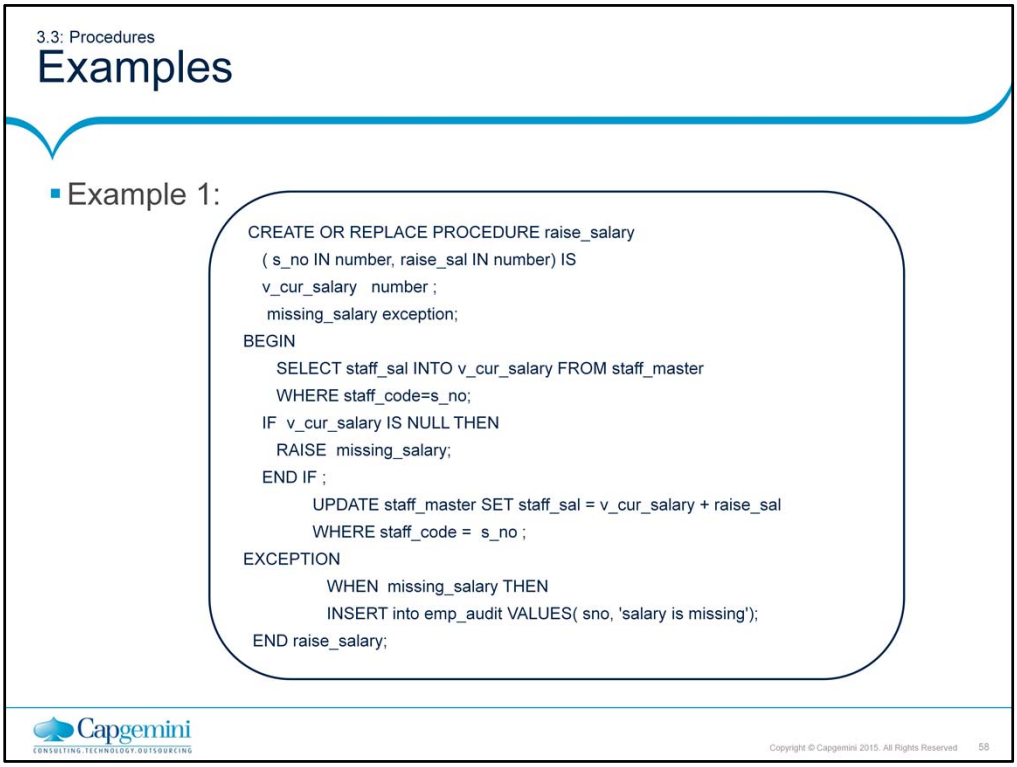
Subprogram Parameter Modes (contd.):
Examples:

Example 2:

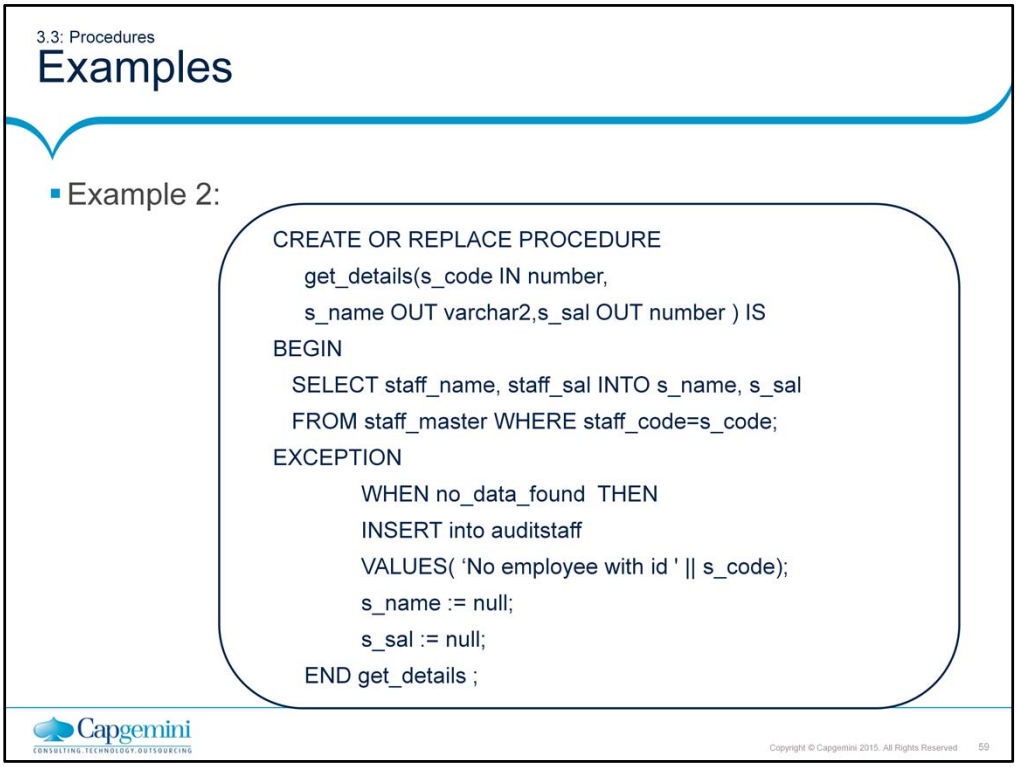
```
SQL > SET SERVEROUTPUT ON
SQL > CREATE OR REPLACE PROCEDRE PROC1 AS
      2    BEGIN
      3         DBMS_OUTPUT.PUT_LINE('Hello from procedure …');
      4    END;
      5    /
Procedure created.
SQL > EXECUTE PROC1
Hello from procedure …
PL/SQL procedure successfully created.
```

Example 3:

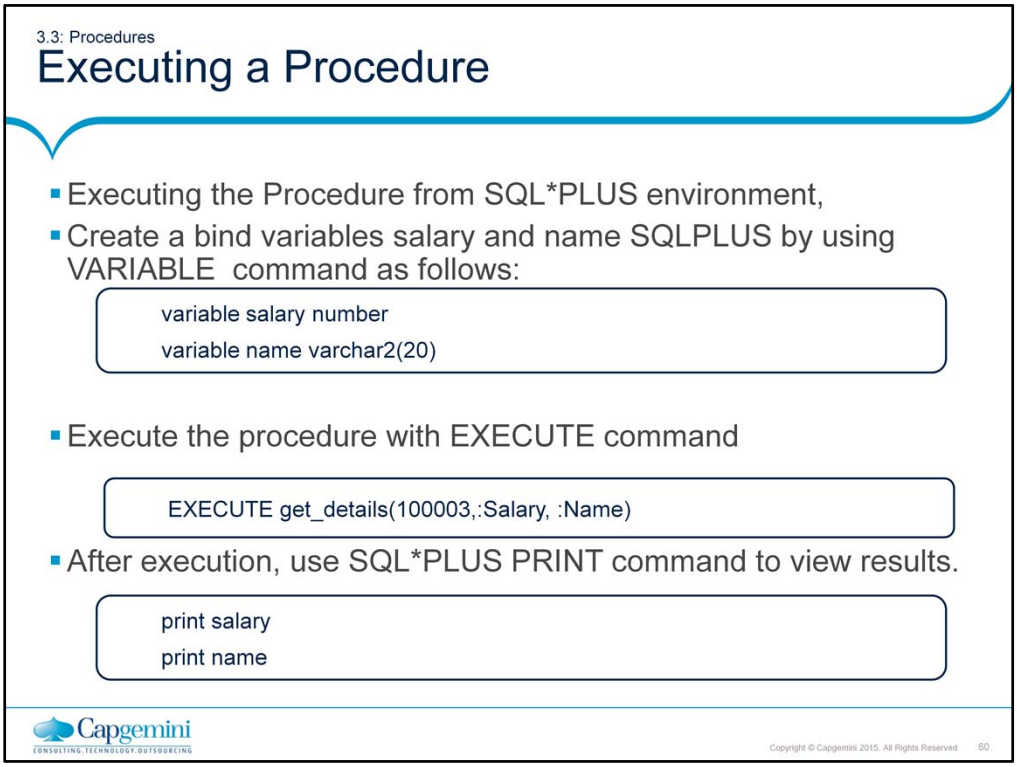
```
SQL > CREATE OR REPLACE                                    PROCEDURE PRO
    2     (N1 IN NUMBER, N2 IN NUMBER, TOT OUT NUMBER) IS
    3   BEGIN
    4     TOT := N1 + N2;
    5   END;
    6   /

Procedure created.

SQL > VARIABLE T NUMBER
SQL > EXEC PROC2(33, 66, :T)

PL/SQL procedure successfully completed.

SQL > PRINT T

          T
-- - - - - - - - - -
          99
```



The procedure example on the slide modifies the salary of staff member. It also handles exceptions appropriately. In addition to the above shown exception you can also handle “NO_DATA_FOUND” exception. The procedure accepts two parameters which is the staff_code and amount that has to be given as raise to the staff member.



The procedure on the slide accept three parameters, one is IN mode and other two are OUT mode. The procedure retrieves the name and salary of the staff member based on the staff_code passed to the procedure. The S_NAME and S_SAL are the OUT parameters that will return the values to the calling environment



Procedures can be executed through command line as shown on the slide or can be called from other procedures/functions/Anonymous PL/SQL blocks.

On the slide the first snippet declares two variables viz. salary and name. The second snippet calls the procedure and passes the actual parameters. The first is a literal string and the next two parameters are empty variables which will be assigned with values within the procedure.

Calling the procedure from an anonymous PL/SQL block

```
DECLARE
      s_no  number(10):=&sno;
      sname varchar2(10);
      sal  number(10,2);
BEGIN
      get_details(s_no,sname,sal);
      dbms_output.put_line('Name:'||sname||'Salary'||sal);
END;
```

```
PROCEDURE Create_Dept( New_Deptno IN NUMBER,
   New_Dname IN  VARCHAR2  DEFAULT 'TEMP') IS
BEGIN
  INSERT INTO  department_master
         VALUES ( New_Deptno, New_Dname, New_Loc) ;
END ;
```

Parameter default values:
Like variable declarations, the formal parameters to a procedure or function can have default values.
If a parameter has default values, it does not have to be passed from the calling environment.
If it is passed, actual parameter will be used instead of default.
Only IN parameters can have default values.

Examples:
Example 1:

Example 2:
Now consider the following calls to Create_Dept.

```
BEGIN
Create_Dept( 50);
--  Actual  call will be  Create_Dept ( 50, 'TEMP',
'TEMP')

Create_Dept ( 50, 'FINANCE');
-- Actual call will be  Create_Dept ( 50, 'FINANCE'
,'TEMP')

Create_Dept( 50, 'FINANCE', 'BOMBAY') ;
-- Actual  call will be  Create_Dept(50, 'FINANCE',
'BOMBAY' )

END;
```

Procedures (contd.):
Using Positional, Named, or Mixed Notation for Subprogram Parameters:
When calling a subprogram, you can write the actual parameters by using either Positional notation, Named notation, or Mixed notation.

Positional notation: You specify the same parameters in the same order as they are declared in the procedure. This notation is compact, but if you specify the parameters (especially literals) in the wrong order, the bug can be hard to detect. You must change your code if the procedure's parameter list changes.

Named notation: You specify the name of each parameter along with its value. An arrow (=>) serves as the “association operator”. The order of the parameters is not significant.

Mixed notation: You specify the first parameters with “Positional notation”, and then switch to “Named notation” for the last parameters. You can use this notation to call procedures that have some “required parameters”, followed by some “optional parameters”.

We have already seen a few examples of calling procedures with Positional notation.
The example shows calling the Create_Dept procedure with named notations

```
Create_Dept (New_Deptno=> 50, New_Dname=>'FINANCE');
```

Example - 2

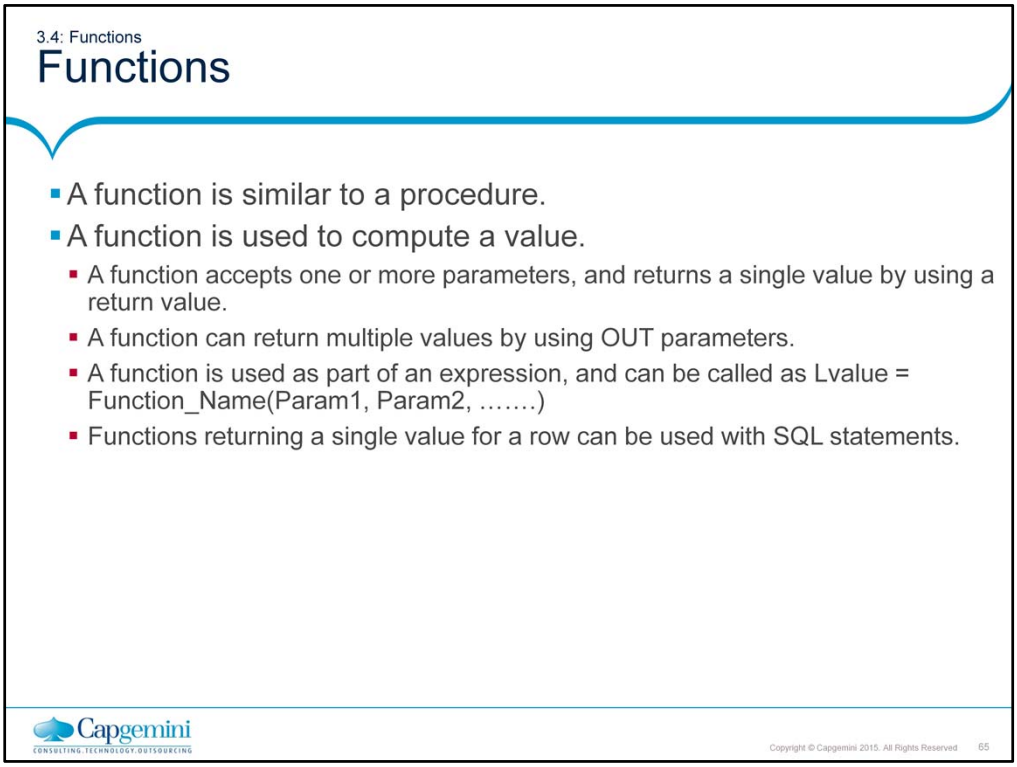
```
CREATE OR REPLACE PROCEDURE spEmp
   (nEmpno IN employee.empno%TYPE,
    nSal IN OUT NUMBER)
   AS
   nMinSal NUMBER;
   BEGIN
      SELECT min(sal) INTO nMinSal FROM employee;
      IF nSal<nMinSal
      THEN
            nSal:=nSal+nSal*.3;
     END IF;
  END spEmp;
  /
```

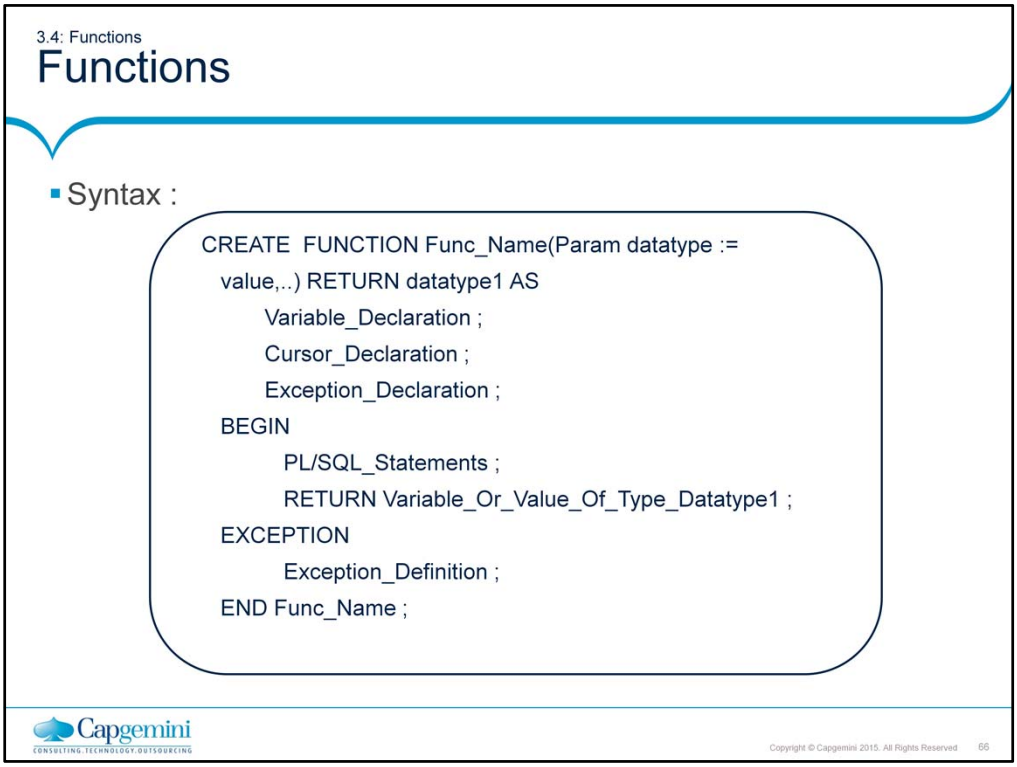
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 63

Example - 2

```
DECLARE
    salno NUMBER;
  BEGIN
    salno:=&salno;
    spEmp(&empno,salno);
    DBMS_OUTPUT.PUT_LINE(salno);
   END;
   /
```



3.4: Functions
Functions
A function is similar to a procedure.
A function is used to compute a value.
A function accepts one or more parameters, and returns a single value by using a return value.
A function can return multiple values by using OUT parameters.
A function is used as part of an expression, and can be called as Lvalue = Function_Name(Param1, Param2, …….)
Functions returning a single value for a row can be used with SQL statements.
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
65


3.4: Functions
Functions
Syntax :
CREATE FUNCTION Func_Name(Param datatype :=
value,..) RETURN datatype1 AS
Variable_Declaration ;
Cursor_Declaration ;
Exception_Declaration ;
BEGIN
PL/SQL_Statements ;
RETURN Variable_Or_Value_Of_Type_Datatype1 ;
EXCEPTION
Exception_Definition ;
END Func_Name ;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
66

3.4: Functions

Examples

- Example 1:

```
CREATE FUNCTION crt_dept(dno number,
  dname varchar2)  RETURN number AS
BEGIN
      INSERT into department_master
      VALUES (dno,dname) ;
                return 1 ;
EXCEPTION
      WHEN others THEN
                return 0 ;
END crt_dept ;
```

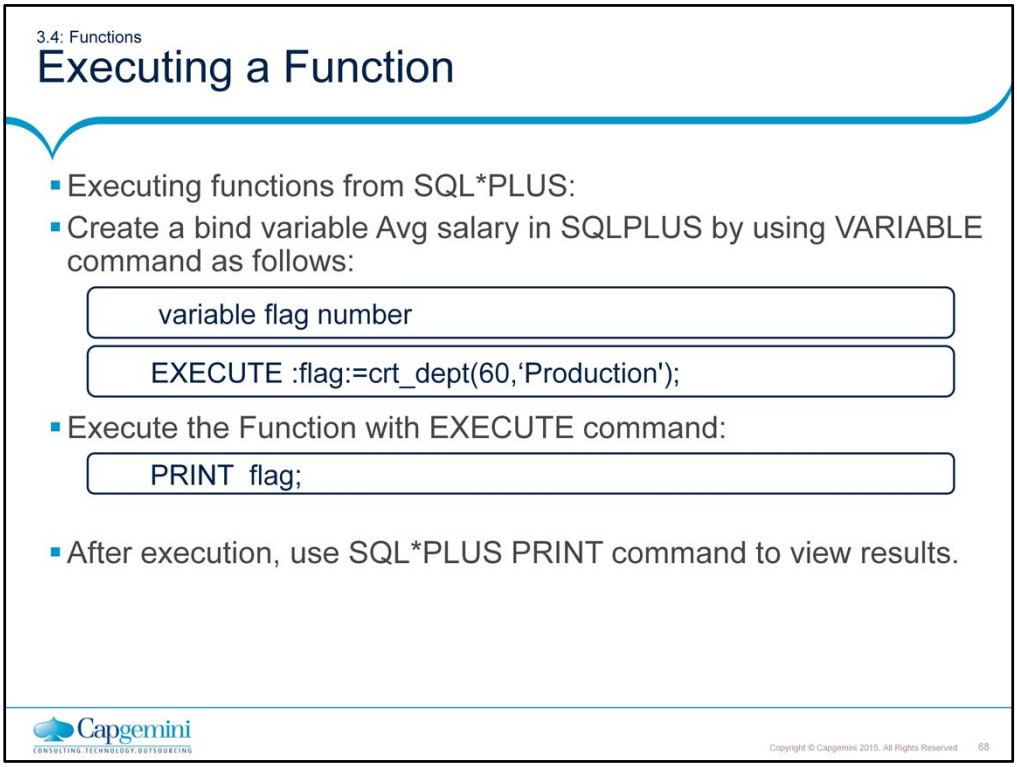
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 67

Example 2:
Function to calculate average salary of a department:

Function returns average salary of the department
Function returns -1, in case no employees are there in the department.
Function returns -2, in case of any other error.

```
CREATE OR REPLACE FUNCTION get_avg_sal(p_deptno
in number) RETURN number as
   V_Sal number;
BEGIN
        SELECT Trunc(Avg(staff_sal)) INTO V_Sal
        FROM staff_master
        WHERE deptno=P_Deptno;
        IF v_sal is null THEN
                v_sal := -1 ;
         END IF;
                return v_sal;
EXCEPTION
       WHEN others THEN
              return -2;  --signifies any other errors
END get_avg_sal;
```

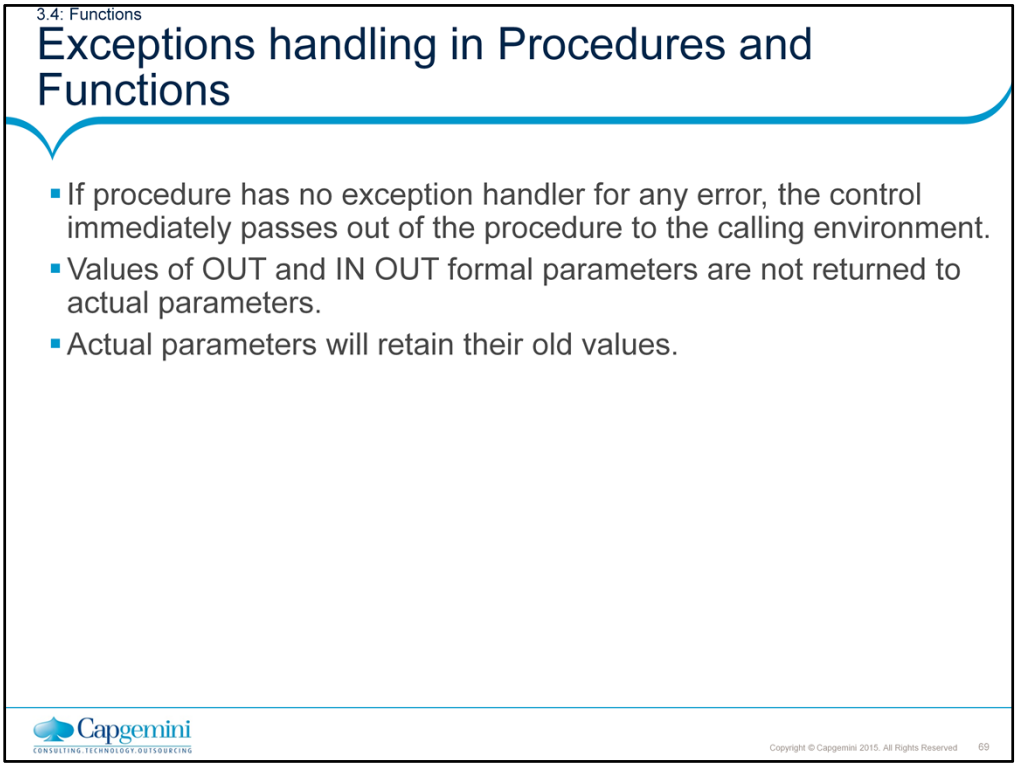


Functions can also be executed through command line as shown on the slide or can be called from other procedures/functions/Anonymous PL/SQL blocks.
The second snippet calls the function and passes the actual parameters. The variable declared earlier is used for collecting the return value from the function
Calling the function from an anonymous PL/SQL block

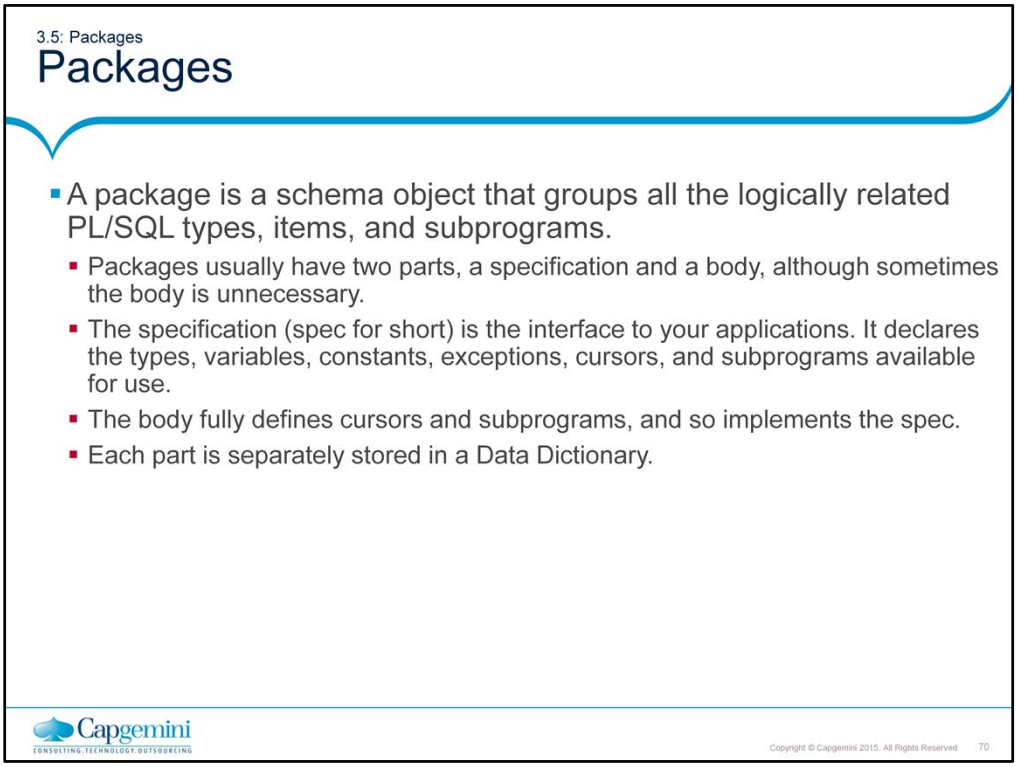
```
DECLARE
  avgsalary  number;
BEGIN
    avgsalary:=_get_avg_sal(20);
    dbms_output.put_line('The average salary of Dept 20 is'||
    avgsalary);
END;
```

Calling function using a Select statement

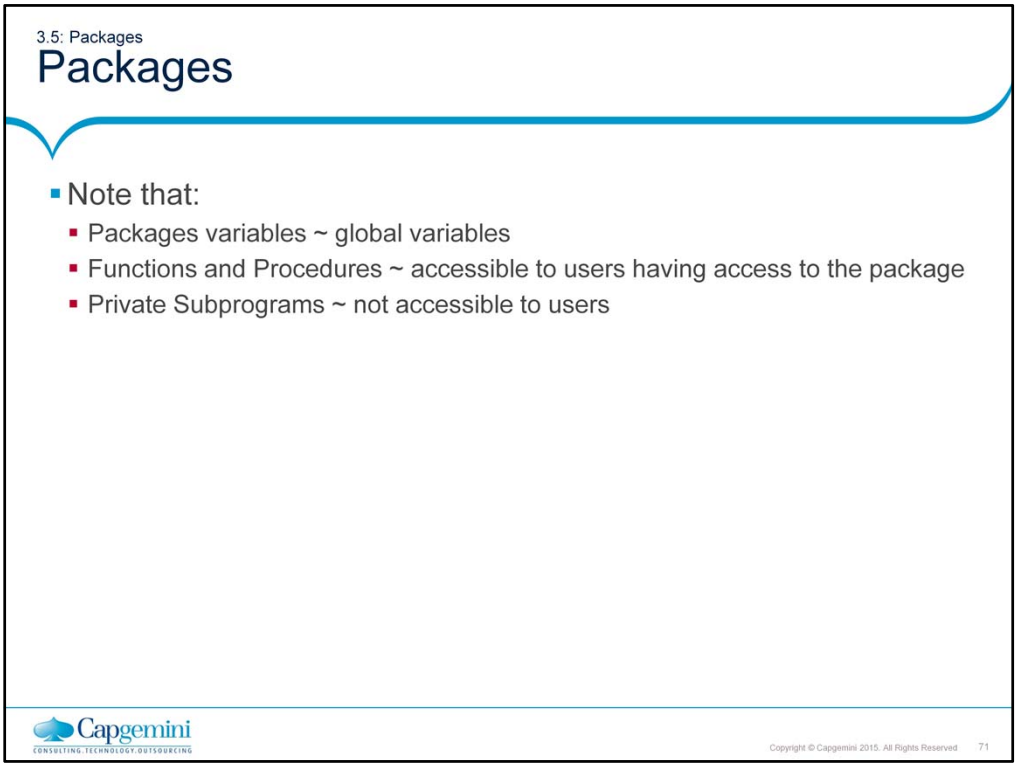
```
SELECT get_avg_sal(30) FROM staff_master;
```

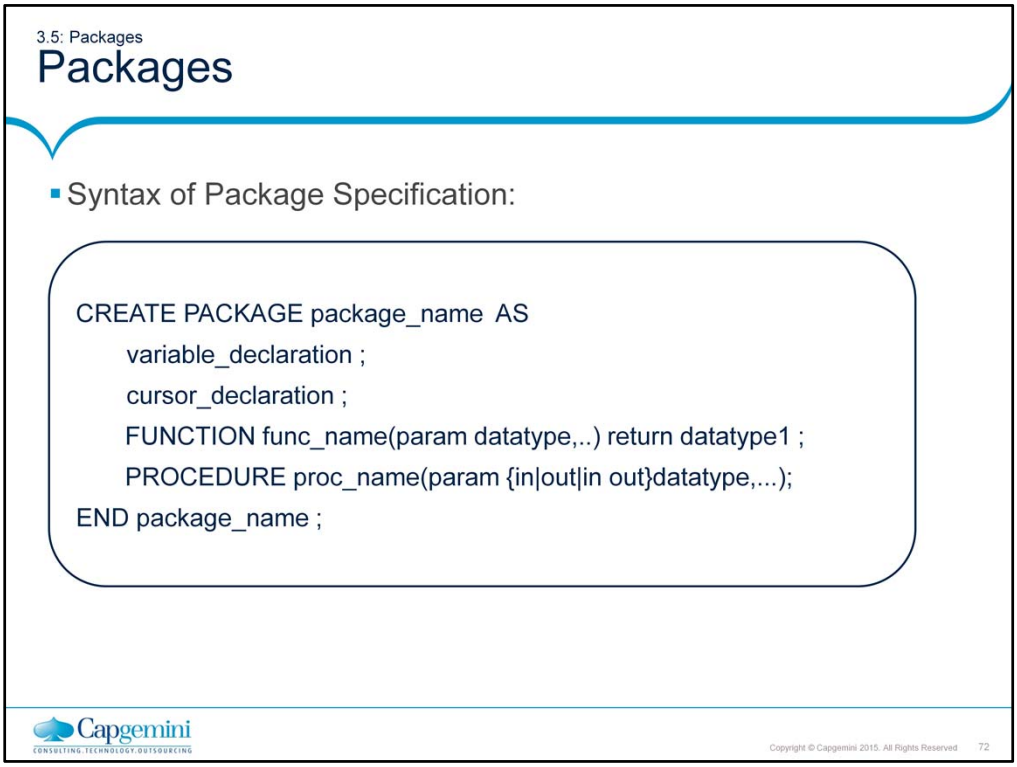


Exceptions raised inside Procedures and Functions:
If an error occurs inside a procedure, an exception (pre-defined or user-defined) is raised.

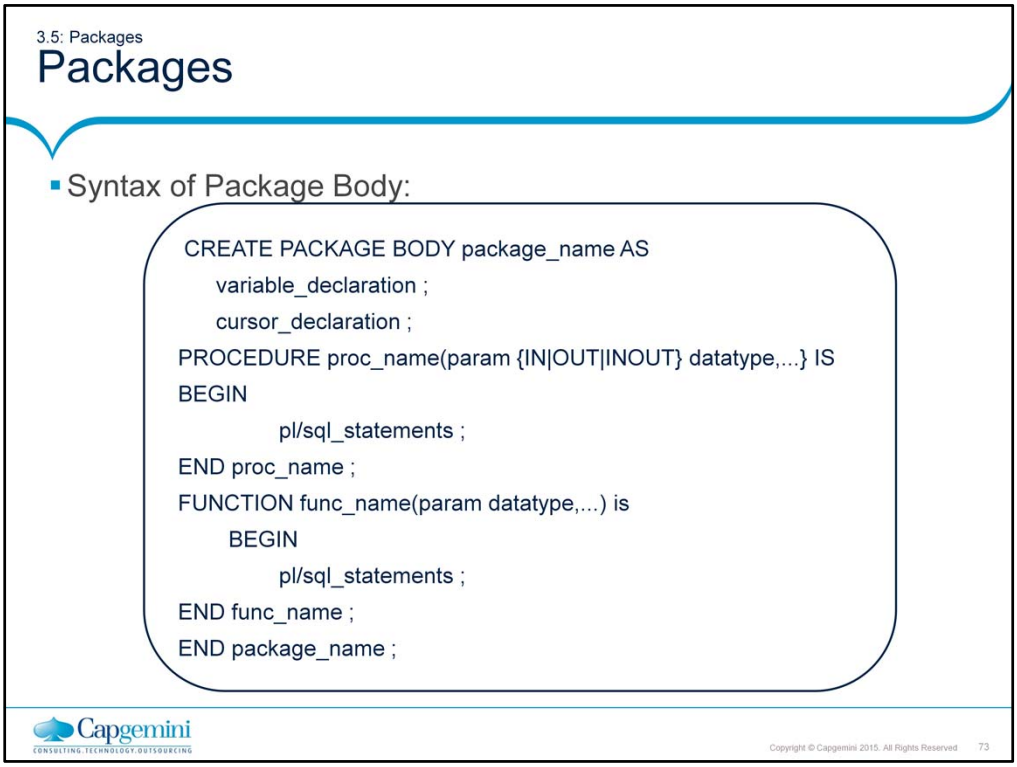


Packages:
Packages are PL/SQL constructs that allow related objects to be stored together. A Package consists of two parts, namely “Package Specification” and “Package Body”. Each of them is stored separately in a “Data Dictionary”.
Package Specification: It is used to declare functions and procedures that are part of the package. Package Specification also contains variable and cursor declarations, which are used by the functions and procedures. Any object declared in a Package Specification can be referenced from other PL/SQL blocks. So Packages provide global variables to PL/SQL.
Package Body: It contains the function and procedure definitions, which are declared in the Package Specification. The Package Body is optional. If the Package Specification does not contain any procedures or functions and contains only variable and cursor declarations then the body need not be present.
All functions and procedures declared in the Package Specification are accessible to all users who have permissions to access the Package. Users cannot access subprograms, which are defined in the Package Body but not declared in the Package Specification. They can only be accessed by the subprograms within the Package Body. This facility is used to hide unwanted or sensitive information from users.
A Package generally consists of functions and procedures, which are required by a specific application or a particular module of an application.

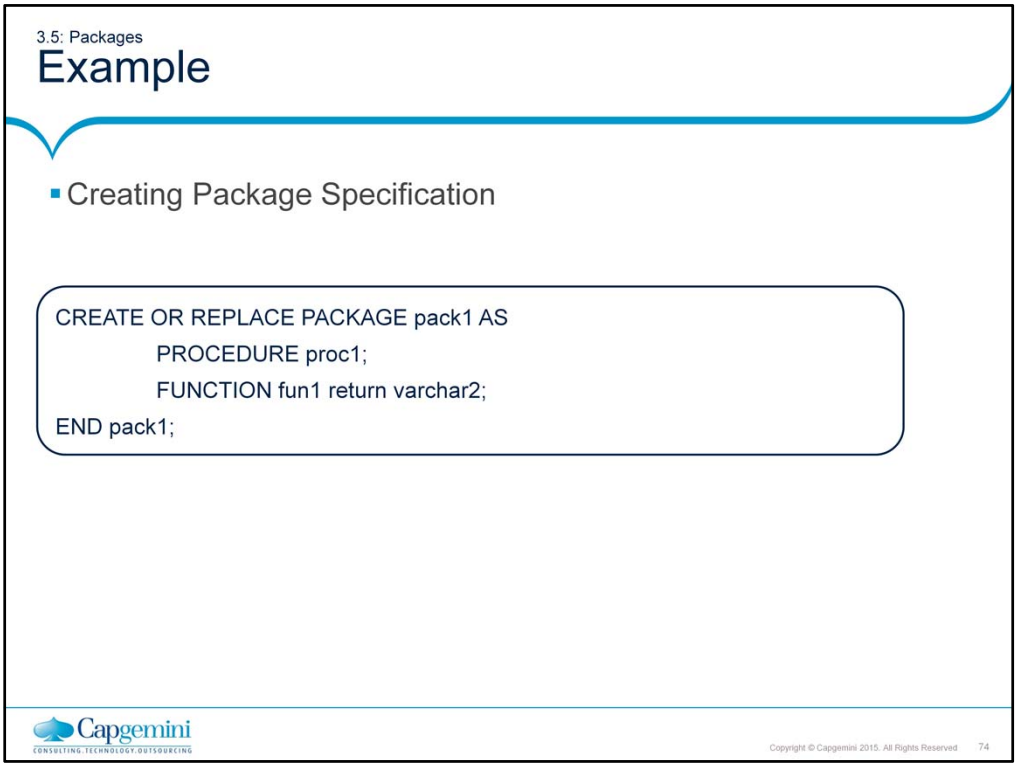

3.5: Packages
Packages
Note that:
Packages variables ~ global variables
Functions and Procedures ~ accessible to users having access to the package
Private Subprograms ~ not accessible to users
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
71

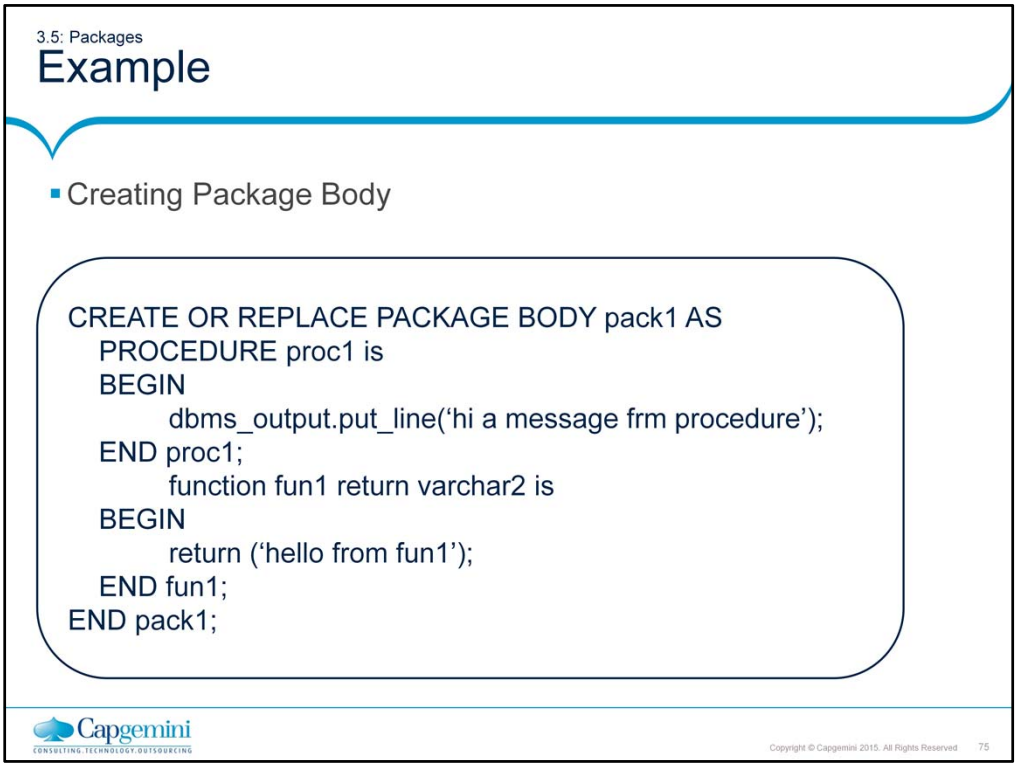


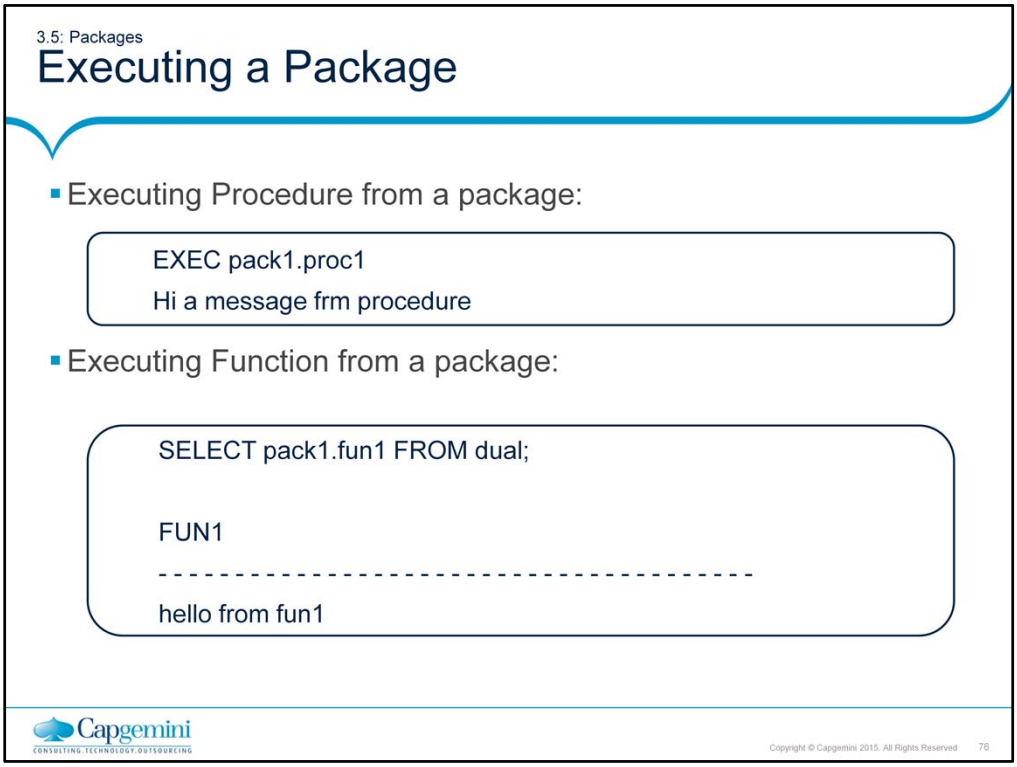
The package specification can contain variables,cursors, procedure and functions. Whatever is specified within the packages are global by default and are accessible to users who have the privileges on the package



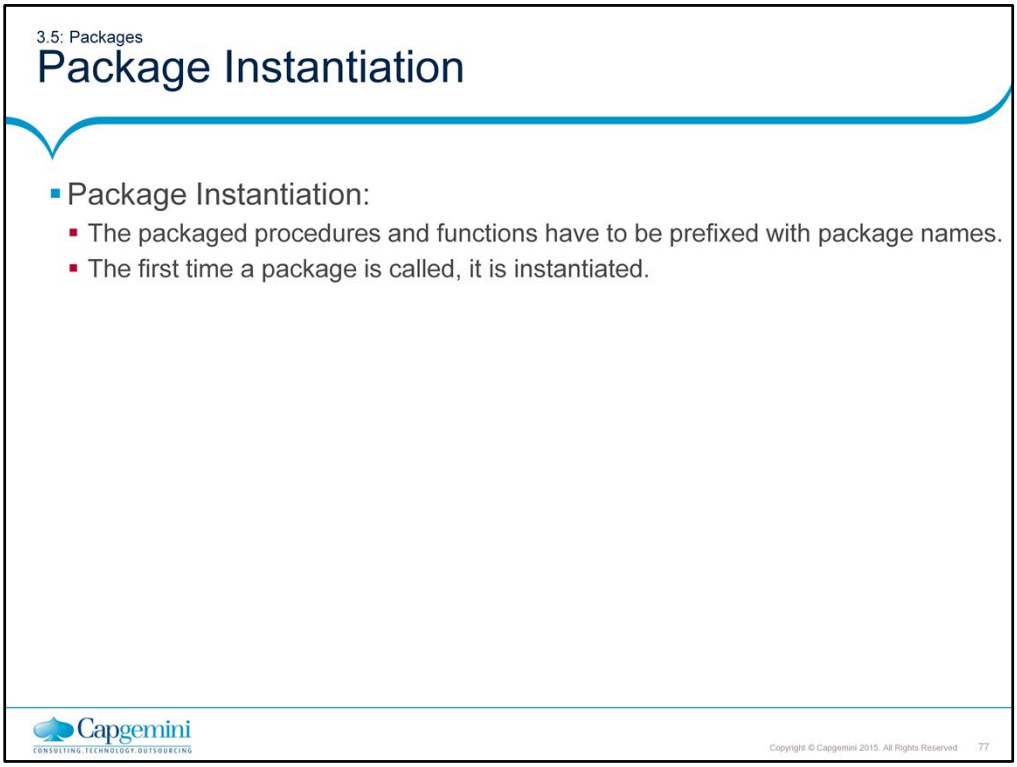
The package body should contain all the procedures and function declared in the package specification. Any variables and cursors declared within the package body are local to the package body and are accessible only within the package. The package body can contain additional procedures and functions apart from the ones declared in package body. The procedures/functions are local to the package and cannot be accessed by any user outside the package.


3.5: Packages
Example
Creating Package Specification
CREATE OR REPLACE PACKAGE pack1 AS
PROCEDURE proc1;
FUNCTION fun1 return varchar2;
END pack1;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
74


3.5: Packages
Example
Creating Package Body
CREATE OR REPLACE PACKAGE BODY pack1 AS
PROCEDURE proc1 is
BEGIN
dbms_output.put_line('hi a message frm procedure');
END proc1;
function fun1 return varchar2 is
BEGIN
return ('hello from fun1');
END fun1;
END pack1;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
75



Note:
If the specification of the package declares only types, constants, variables, and exceptions, then the package body is not required there. This type of packages only contains global variables that will be used by subprograms or cursors.

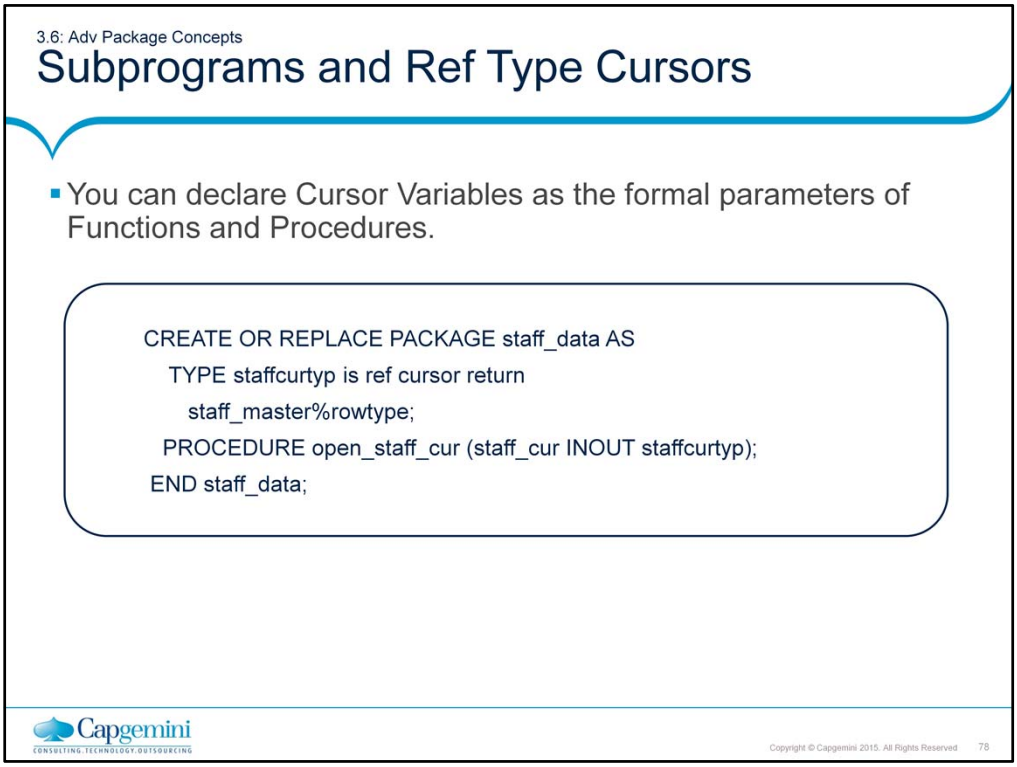


Package Instantiation:
The procedure and function calls are the same as in standalone subprograms.
The packaged procedures and functions have to be prefixed with package names.
The first time a package is called, it is instantiated.

This means that the package is read from disk into memory, and P-CODE is run.

At this point, the memory is allocated for any variables defined in the package.

Each session will have its own copy of packaged variables, so there is no problem of two simultaneous sessions accessing the same memory locations.



Subprograms and Ref Type Cursors:
You can declare Cursor Variables as the formal parameters of Functions and Procedures.
In the following example, you define the REF CURSOR type staffCurTyp, then declare a Cursor Variable of that type as the formal parameter of a procedure:

```
DECLARE
TYPE staffCurTyp IS REF CURSOR RETURN
staff_master%ROWTYPE;
PROCEDURE open_staff_cv (staff_cv IN OUT
staffCurTyp) IS
```

Typically, you open a Cursor Variable by passing it to a stored procedure that declares a Cursor Variable as one of its formal parameters.
The packaged procedure shown in the slide, for example, opens the cursor variable emp_cur.

3.6: Adv Package Concepts

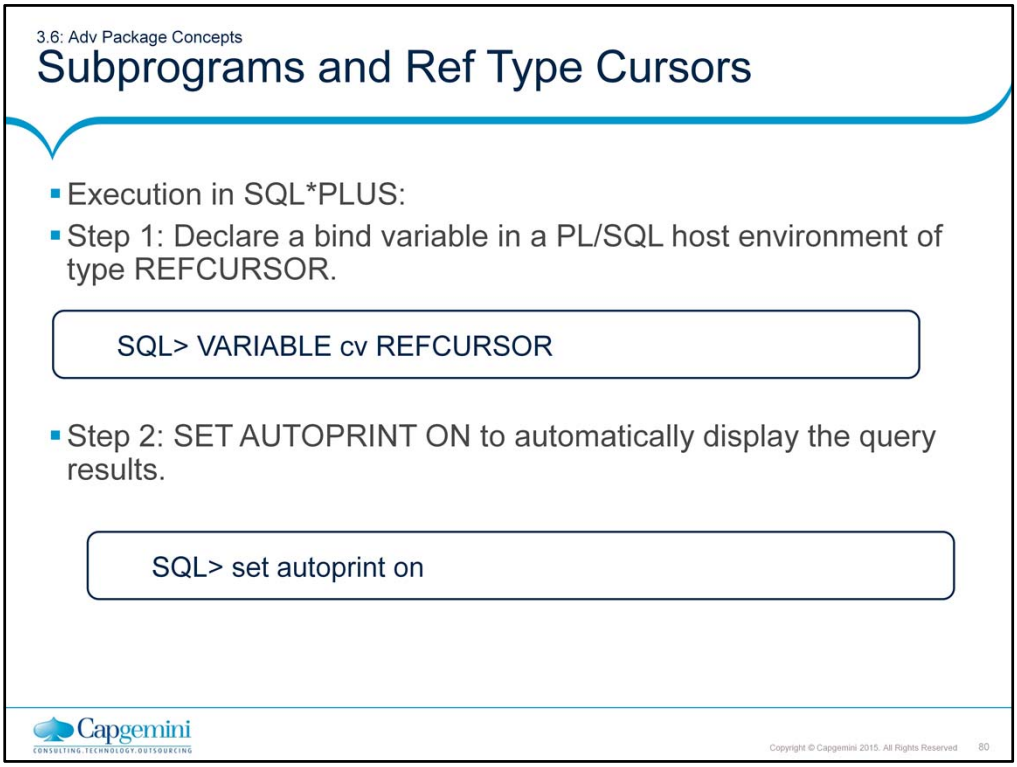
Subprograms and Ref Type Cursors

- Note: Cursor Variable as the formal parameter should be in IN OUT mode.

```
CREATE OR REPLACE PACKAGE BODY staff_data AS
PROCEDURE open_staff_cur (staff_cur INOUT staffcurtyp) IS
BEGIN
    OPEN staff_cur  for SELECT * FROM staff_master;
    end open_staff_cur;
END emp_data;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 79



Subprograms and Ref Type Cursors: Execution in SQL*PLUS:
When you declare a Cursor Variable as the formal parameter of a subprogram that opens the cursor variable, you must specify the IN OUT mode. That way, the subprogram can pass an open cursor back to the caller.
To see the value of the Cursor Variable on the SQL prompt, you need to do following:

Declare a bind variable in a PL/SQL host environment of type REFCURSOR as shown below. The SQL*Plus datatype REFCURSOR lets you declare Cursor Variables, which you can use to return query results from stored subprograms.

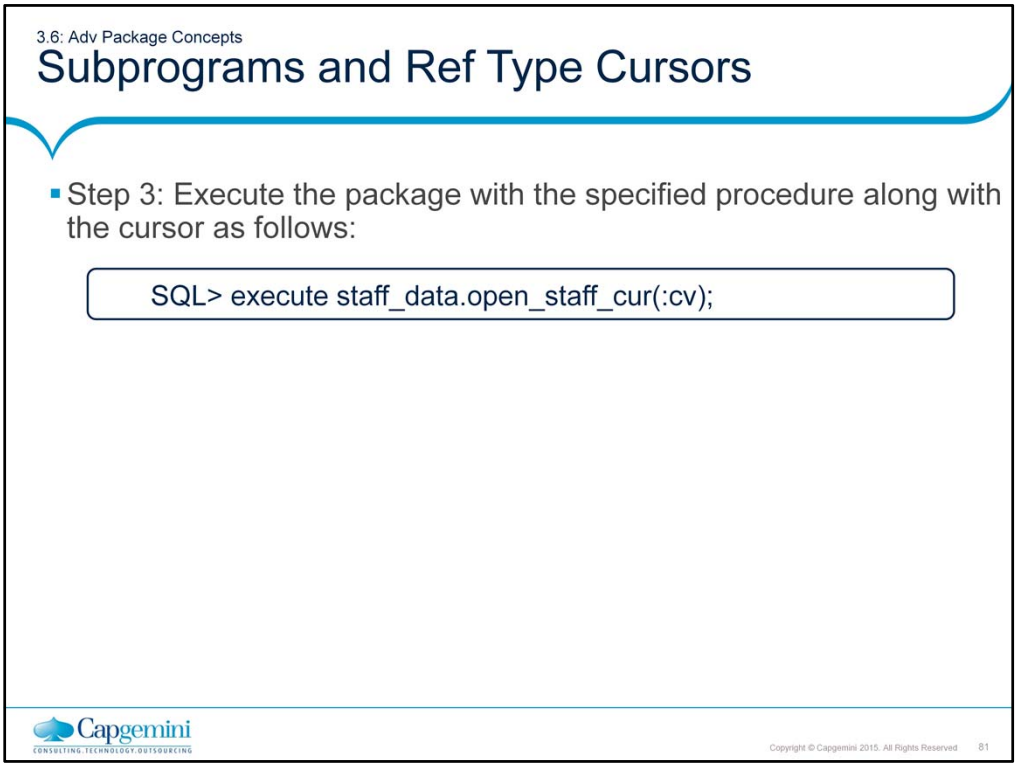
```
SQL> VARIABLE cv REFCURSOR
```

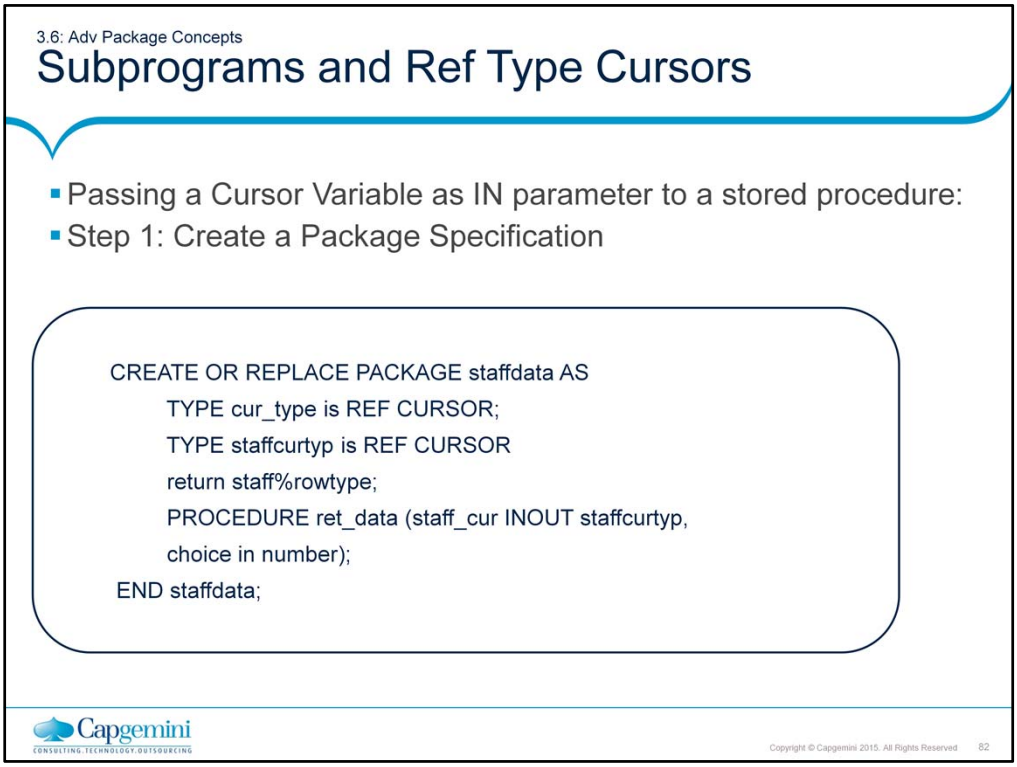
Use the SQL*Plus command SET AUTOPRINT ON to automatically display the query results.

```
SQL> set autoprint on
```

Now execute the package with the specified procedure along with the cursor as follows :

```
SQL> execute emp_data.open_emp_cur(:cv);
```


3.6: Adv Package Concepts
Subprograms and Ref Type Cursors
Step 3: Execute the package with the specified procedure along with the cursor as follows:
SQL> execute staff_data.open_staff_cur(:cv);
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
81



Subprograms and Ref Type Cursors: Passing a Cursor Variable:
You can pass a Cursor Variable and an IN parameter to a stored procedure, which will execute the queries with different return types.
In the example shown in the slide, you are passing the cursor as well as the number variable as choice. Depending on the choice you can write multiple queries, and retrieve the output from the cursor.
When called, the procedure opens the cursor variable emp_cur for the chosen query.

3.6: Adv Package Concepts

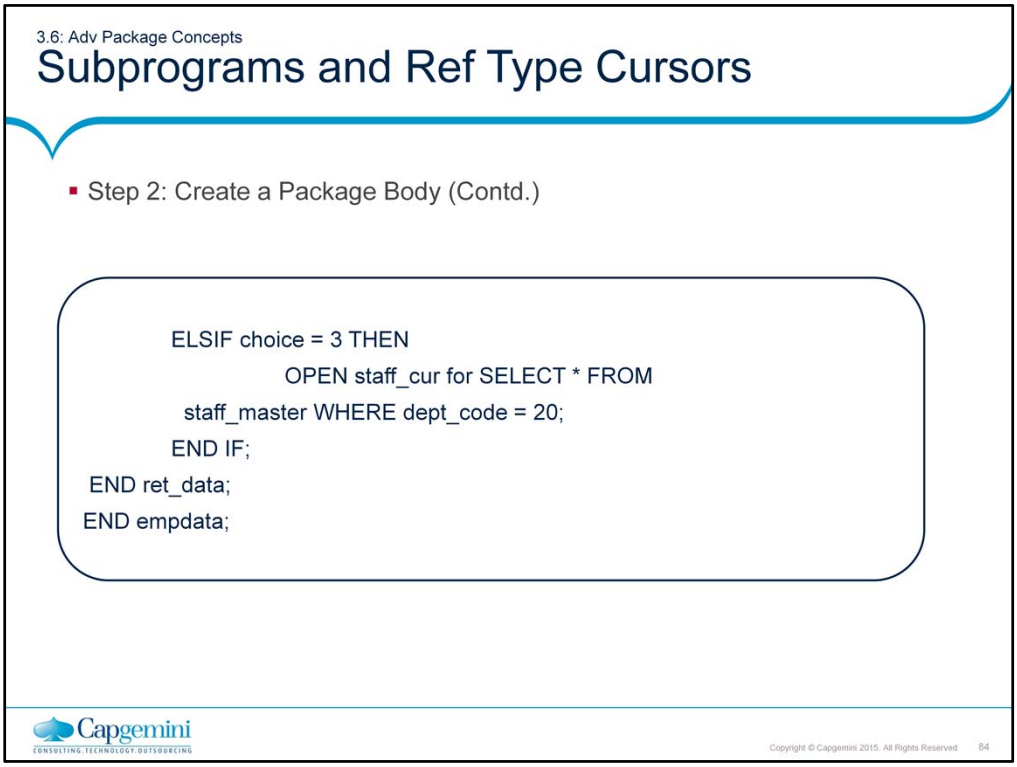
Subprograms and Ref Type Cursors (Contd.)

- Step 2: Create a Package Body:

```
CREATE OR REPLACE PACKAGE BODY staffdata AS
    PROCEDURE ret_data (staff_cur INOUT staffcurtyp,
          choice IN number) is
BEGIN
    IF choice = 1 THEN
              OPEN staff_cur for select * FROM staff_master
                WHERE staff_dob is not null;
    ELSIF choice = 2 THEN
               OPEN staff_cur for SELECT * FROM staff_master
              WHERE staff_sal > 2500;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 83



Step 3: To retrieve the values from the cursor:
Define a variable in SQL *PLUS environment using variable command.
Set the autoprint command on the SQL prompt.
Call the procedure with the package name and the relevant parameters.

```
SQL> variable cur refcursor
SQL> set autoprint on
SQL> execute staffdata.ret_data(:cur,1);
```

```
SQL> CREATE or replace PACKAGE cv_types AS
                              TYPE GenericCurTyp IS REF CURSOR;
TYPE staffCurTyp IS REF CURSOR RETURN
staff_master%ROWTYPE;                    TYPE deptCurTyp IS
REF CURSOR RETURN department_master%ROWTYPE;
 END cv_types;
 /
Package created.
```

Subprograms and Ref Type Cursors: Passing a Cursor Variable (contd.):
In a similar manner, you can pass the Cursor Variable as (: cur) and '2' number for second choice in the EmpData.ret_data procedure. This will give you the output for all the employees who have salary above 2500.
To see the output of the third cursor, use the same package.procedure name with the ': cur' host variable, and choice value which shows all the employees having department number as 20.
We can also create a package with the different REF CURSOR TYPES available (that is define the REF CURSOR type in a separate package), and then reference that type in the standalone procedure.
Example 1: Create a package as shown below:

Example 2: Create a standalone procedure that references the REF CURSOR type GenericCurTyp, which is defined in the package cv_types. Hence create a procedure as shown below:

contd.

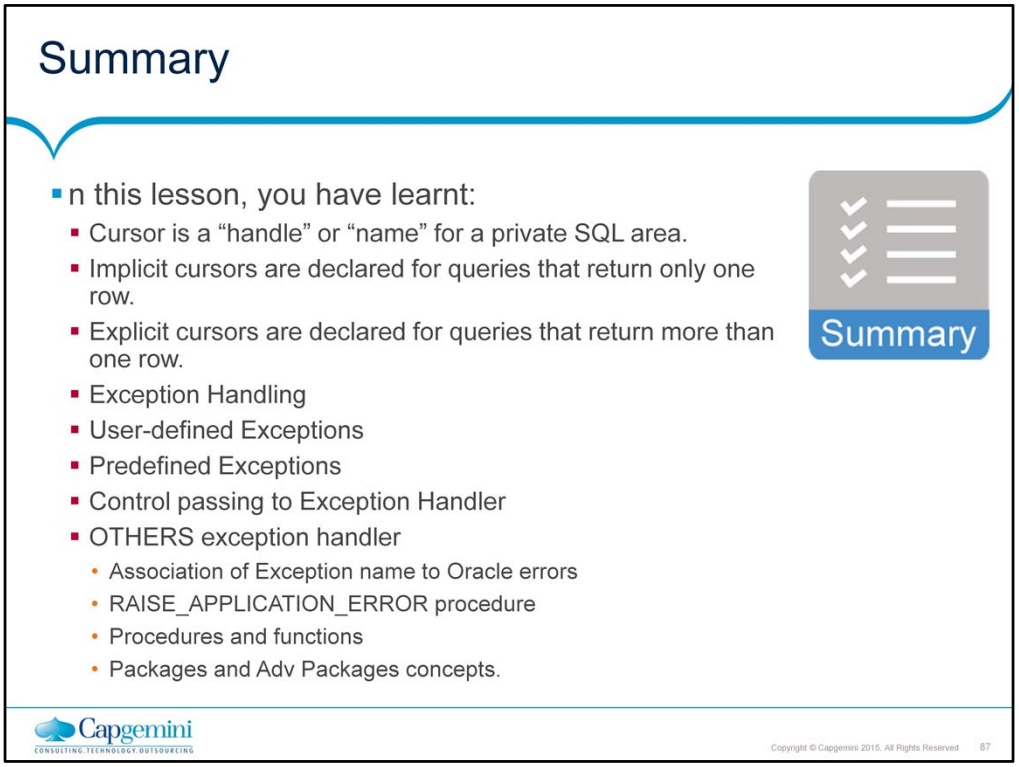
```
SQL> CREATE or REPLACE PROCEDURE open_pro (generic_cv
IN OUT
      cv_types.GenericCurTyp,choice IN NUMBER) IS
 BEGIN
 IF choice = 1 THEN
OPEN generic_cv FOR SELECT * FROM staff_master;
 ELSIF choice = 2 THEN
      OPEN generic_cv FOR SELECT * FROM
department_master;
 ELSIF choice = 3 THEN
           OPEN generic_cv FOR SELECT * FROM item;
   END IF;
END open_pro;
/
Package created.
```

Subprograms and Ref Type Cursors: Passing a Cursor Variable (contd.):

- Open_ procedure, which has a cursor parameter generic pro is an independent _cv, which refers to type REF CURSOR defined in the cv_types package. You can pass a Cursor Variable and Selector to a stored procedure that executes queries with different return types (that is what you have done in the Open_pro procedure). When you call this procedure with the Generic_cv cursor along with the Selector value, the generic_cv cursor gets open and it retrieves the values from the different tables.
- To execute this procedure you need to create the variable of type REFCURSOR, and pass that variable in the Open_pro procedure to see the output.
- For example:

This output is that for the choice number 2, that is the Cursor Variable will show all the rows from the Dept table.

```
SQL> execute open_pro(:cv,2);
```



Add the notes here.

Review Question

- Question 1: %COUNT returns number of rows fetched from the cursor area by using FETCH command.
  - True / False
- Question 2: Implicit SQL cursor is opened or closed by the program.
  - True / False
- Question 3: PL/SQL provides a shortcut via a ___ Loop, which implicitly handles the cursor processing.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 88

Add the notes here.

Review Question

- Question 4: The procedure ___ lets you issue user-defined ORA-error messages from stored subprograms
- Question 5: The ___ tells the compiler to associate an exception name with an Oracle error number.
- Question 6: ___ returns the current error code. And ___ returns the current error message text.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 89

Add the notes here.

Review Question

- Question 7: Anonymous Blocks do not have names.
  - True / False
- Question 8: A function can return multiple values by using OUT parameters
  - True / False
- Question 9: A Package consists of "Package Specification" and "Package Body", each of them is stored in a Data Dictionary named DBMS_package.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Add the notes here.