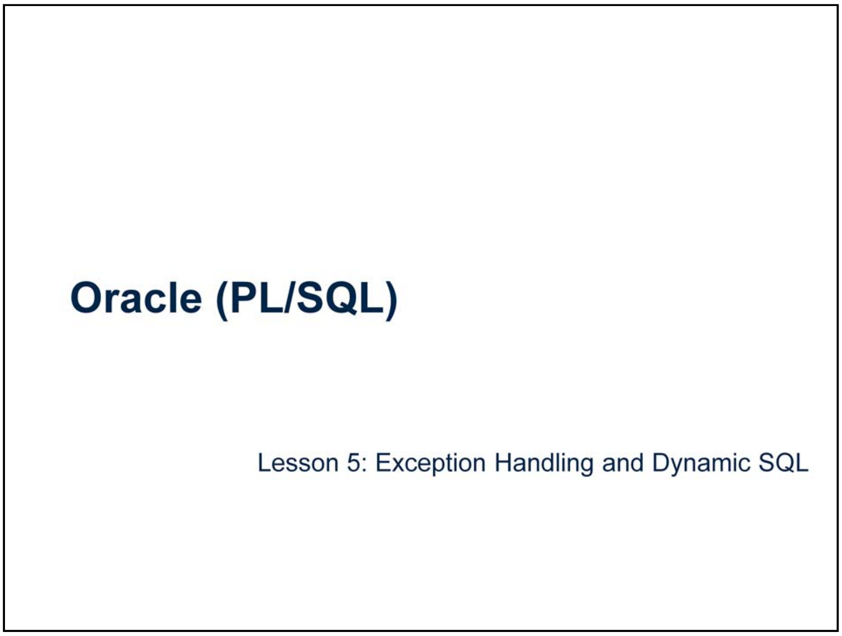
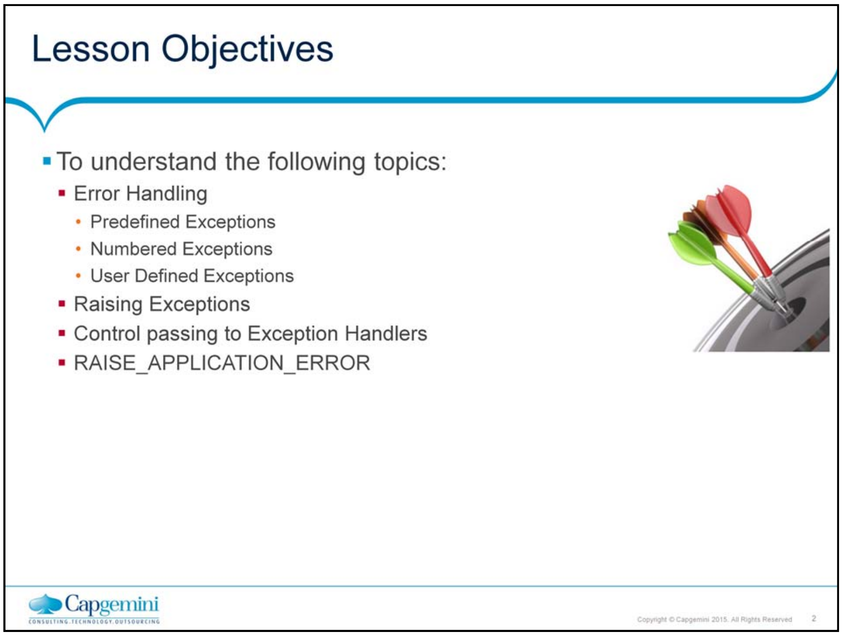


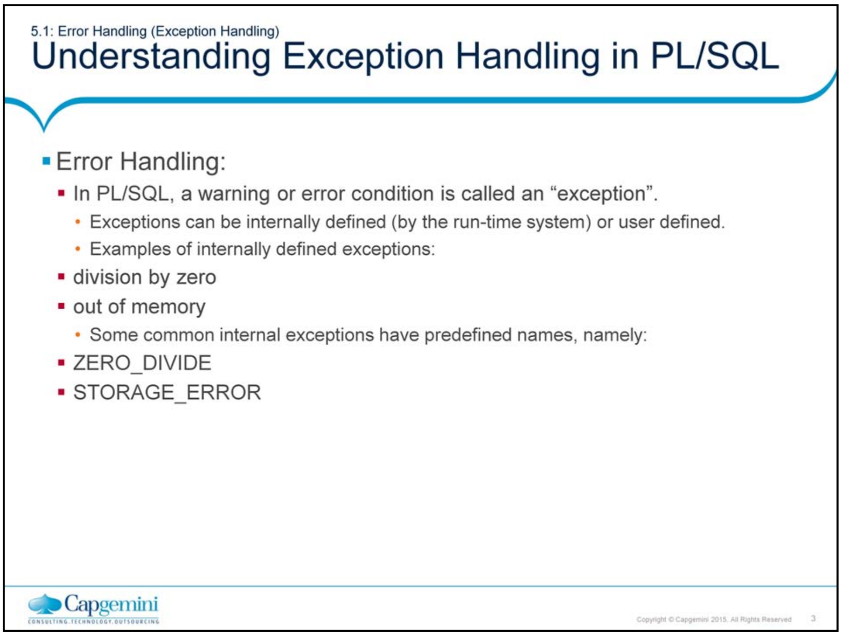
Oracle (PL/SQL)

Lesson 5: Exception Handling and Dynamic SQL



Lesson Objectives

- To understand the following topics:
  - Error Handling
    - Predefined Exceptions
    - Numbered Exceptions
    - User Defined Exceptions
  - Raising Exceptions
  - Control passing to Exception Handlers
  - RAISE_APPLICATION_ERROR



5.1: Error Handling (Exception Handling)

Understanding Exception Handling in PL/SQL

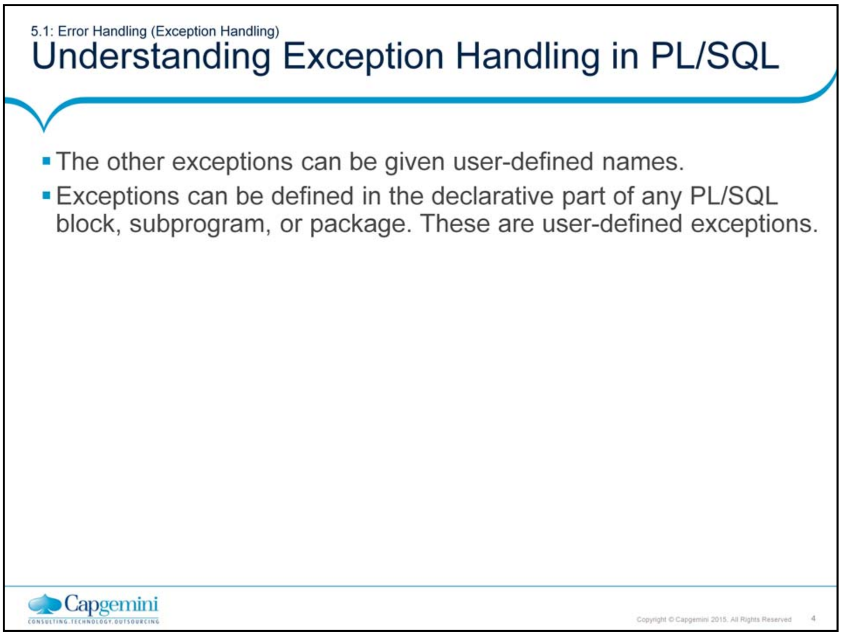
- Error Handling:
  - In PL/SQL, a warning or error condition is called an “exception”.
    - Exceptions can be internally defined (by the run-time system) or user defined.
    - Examples of internally defined exceptions:
  - division by zero
  - out of memory
    - Some common internal exceptions have predefined names, namely:
  - ZERO_DIVIDE
  - STORAGE_ERROR

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.

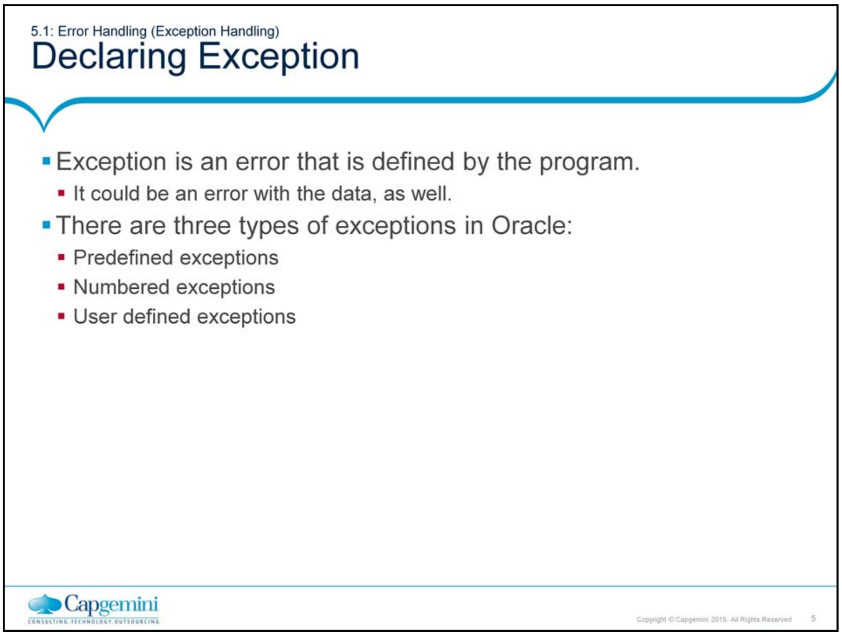


5.1: Error Handling (Exception Handling)

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



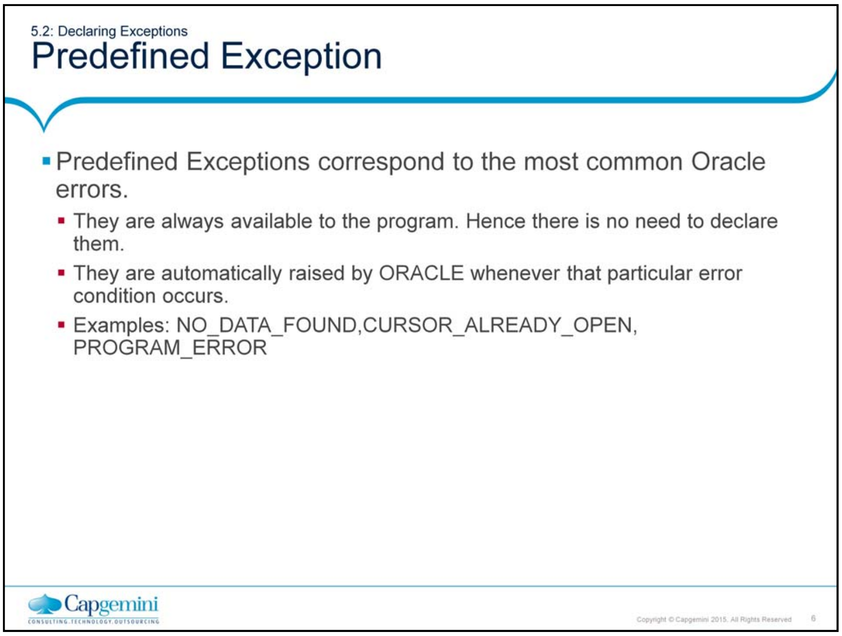
5.1: Error Handling (Exception Handling)

Declaring Exception

- Exception is an error that is defined by the program.
  - It could be an error with the data, as well.
- There are three types of exceptions in Oracle:
  - Predefined exceptions
  - Numbered exceptions
  - User defined exceptions

Declaring Exceptions:

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



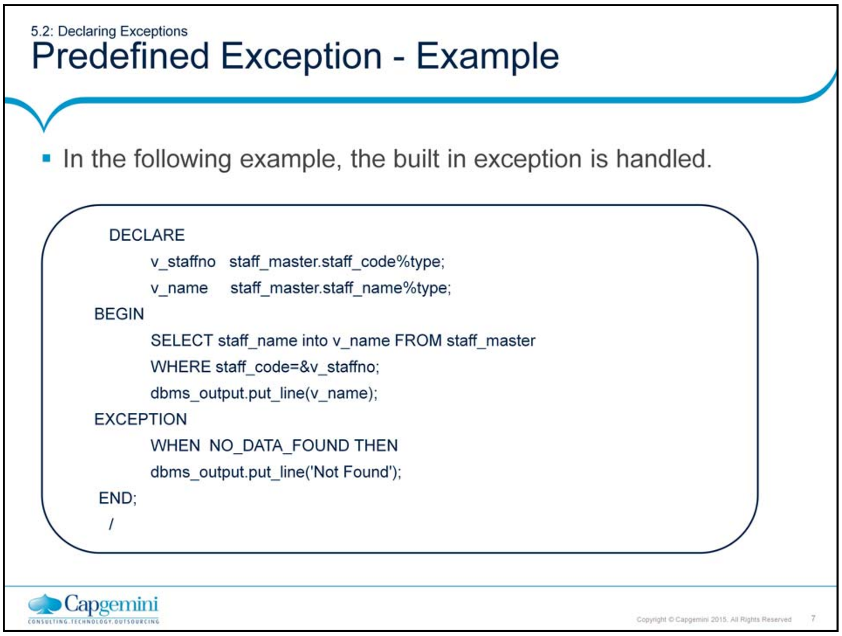
5.2: Declaring Exceptions

Predefined Exception

- Predefined Exceptions correspond to the most common Oracle errors.
  - They are always available to the program. Hence there is no need to declare them.
  - They are automatically raised by ORACLE whenever that particular error condition occurs.
  - Examples: NO_DATA_FOUND,CURSOR_ALREADY_OPEN, PROGRAM_ERROR

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.



5.2: Declaring Exceptions

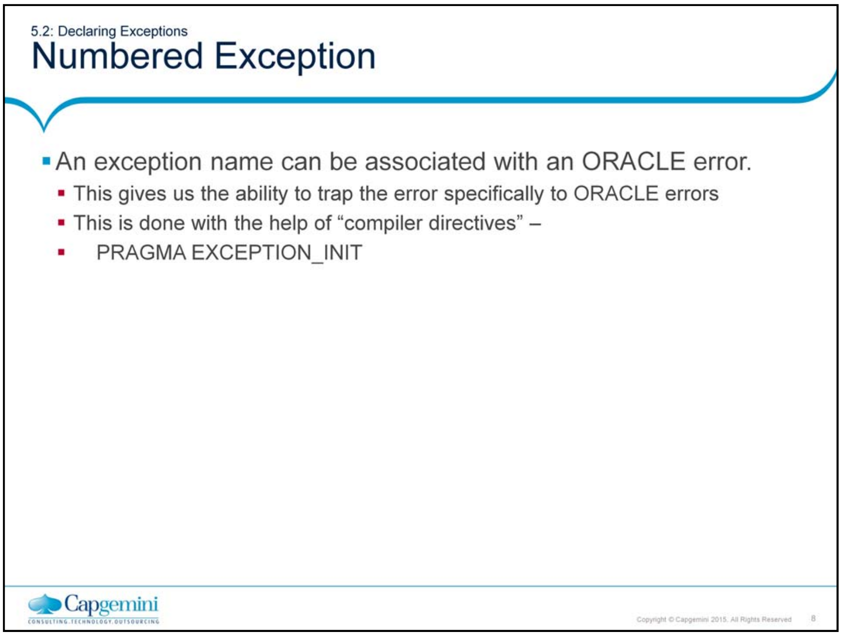
Predefined Exception - Example

- In the following example, the built in exception is handled.

```
  DECLARE
        v_staffno   staff_master.staff_code%type;
        v_name      staff_master.staff_name%type;
BEGIN
        SELECT staff_name into v_name FROM staff_master
        WHERE staff_code=&v_staffno;
        dbms_output.put_line(v_name);
EXCEPTION
        WHEN  NO_DATA_FOUND THEN
        dbms_output.put_line('Not Found');
 END;
  /
```

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.



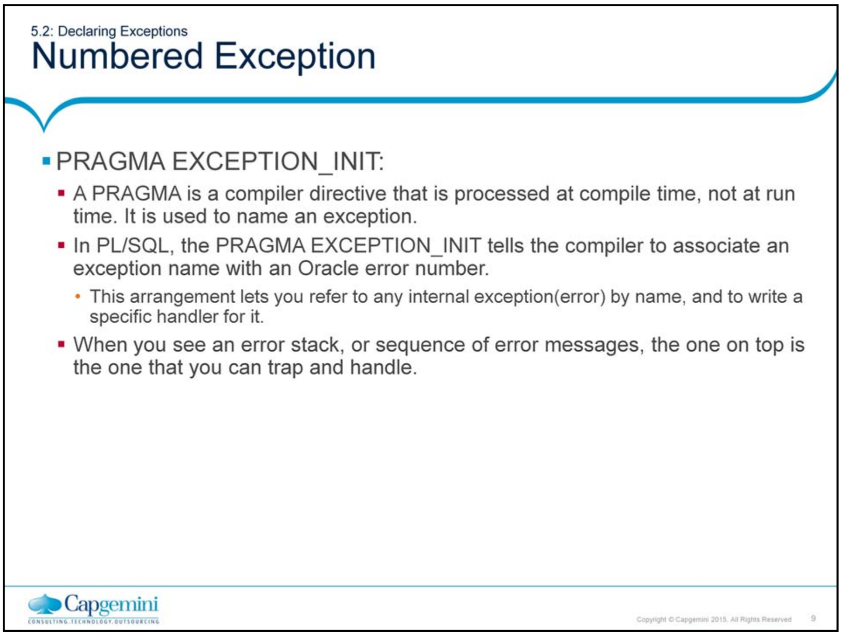
5.2: Declaring Exceptions

Numbered Exception

- An exception name can be associated with an ORACLE error.
  - This gives us the ability to trap the error specifically to ORACLE errors
  - This is done with the help of “compiler directives” –
  - PRAGMA EXCEPTION_INIT

Numbered Exception:

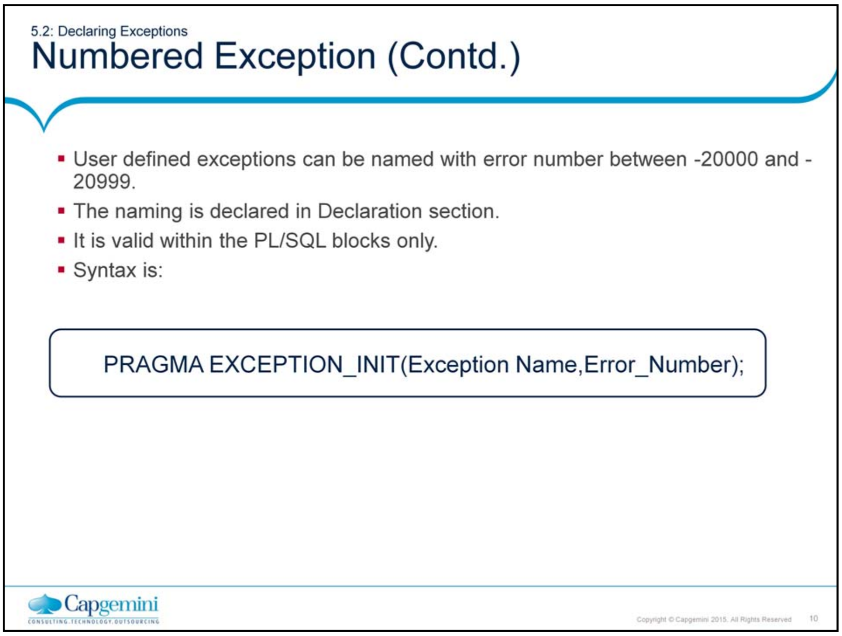
The Numbered Exceptions are Oracle errors bound to a user defined exception name.



5.2: Declaring Exceptions

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.

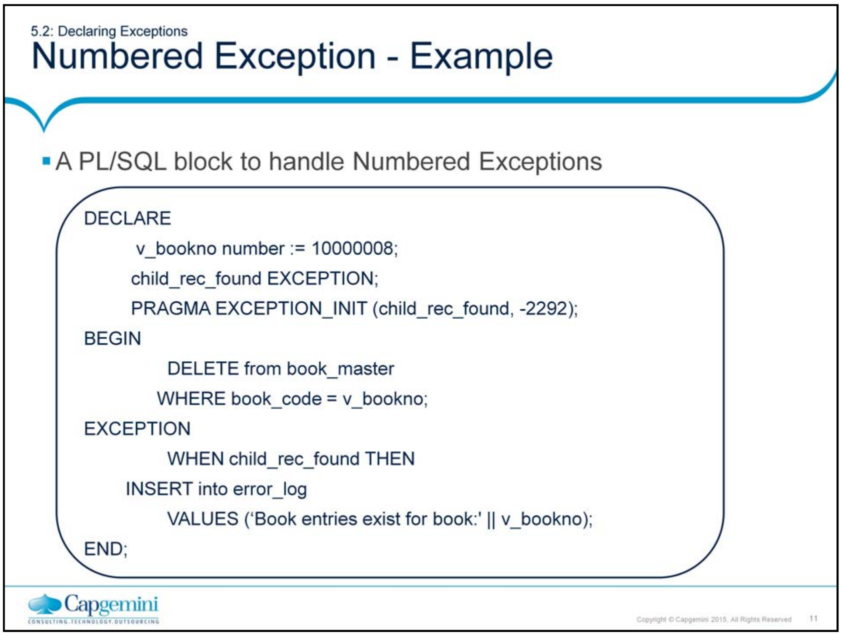


5.2: Declaring Exceptions

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);
```



5.2: Declaring Exceptions

Numbered Exception - Example

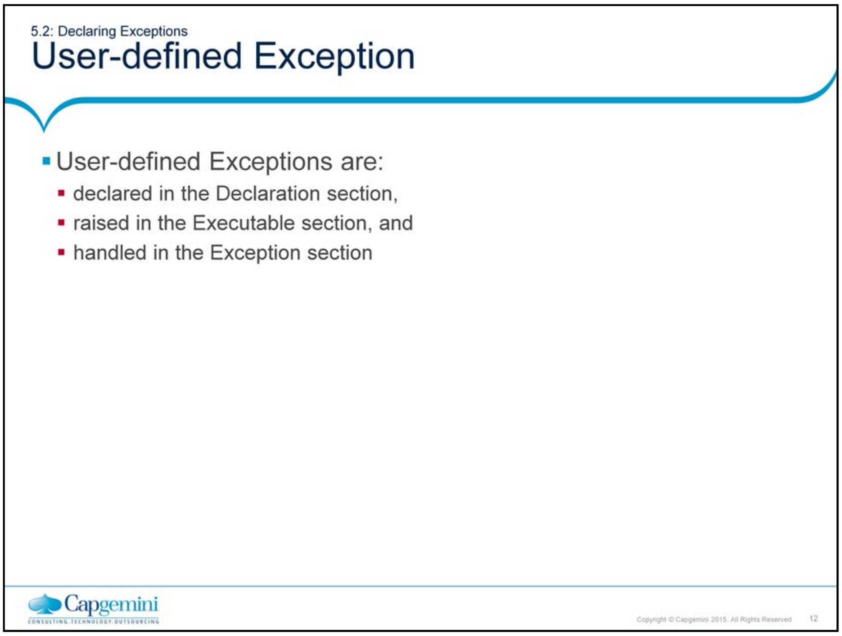
- A PL/SQL block to handle Numbered Exceptions

```
DECLARE
         v_bookno number := 10000008;
        child_rec_found EXCEPTION;
        PRAGMA EXCEPTION_INIT (child_rec_found, -2292);
BEGIN
                DELETE from book_master
              WHERE book_code = v_bookno;
EXCEPTION
                WHEN child_rec_found THEN
        INSERT into error_log
                VALUES ('Book entries exist for book:' || v_bookno);
END;
```

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.



5.2: Declaring Exceptions

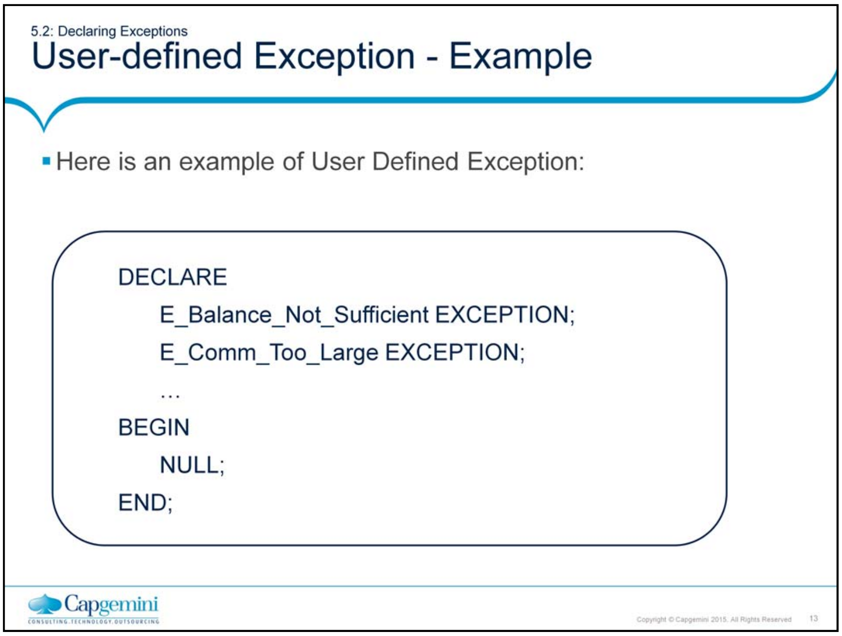
User-defined Exception

- User-defined Exceptions are:
  - declared in the Declaration section,
  - raised in the Executable section, and
  - handled in the Exception section

Copyright © Capgemini 2015. All Rights Reserved 12

User-Defined Exceptions:

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

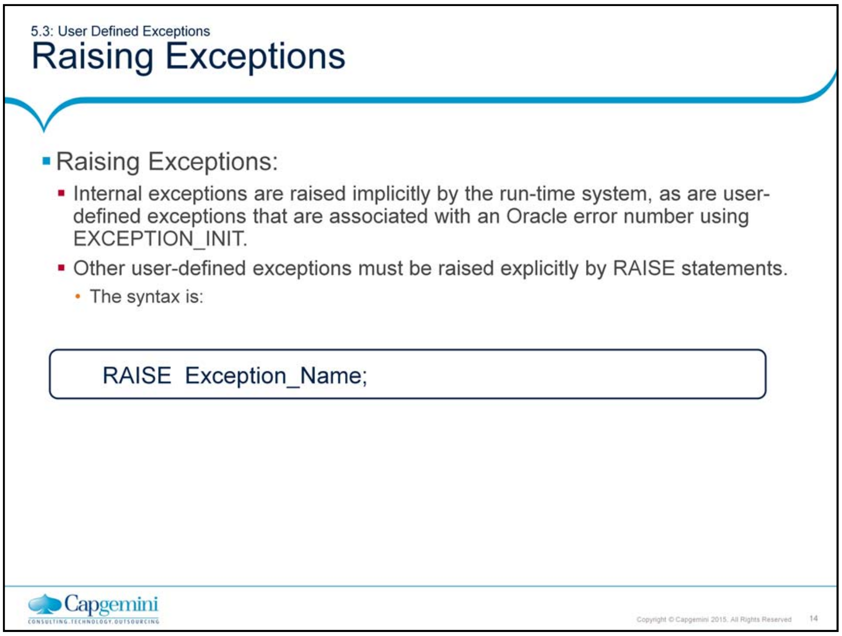


5.2: Declaring Exceptions

User-defined Exception - Example

- Here is an example of User Defined Exception:

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



5.3: User Defined Exceptions

Raising Exceptions

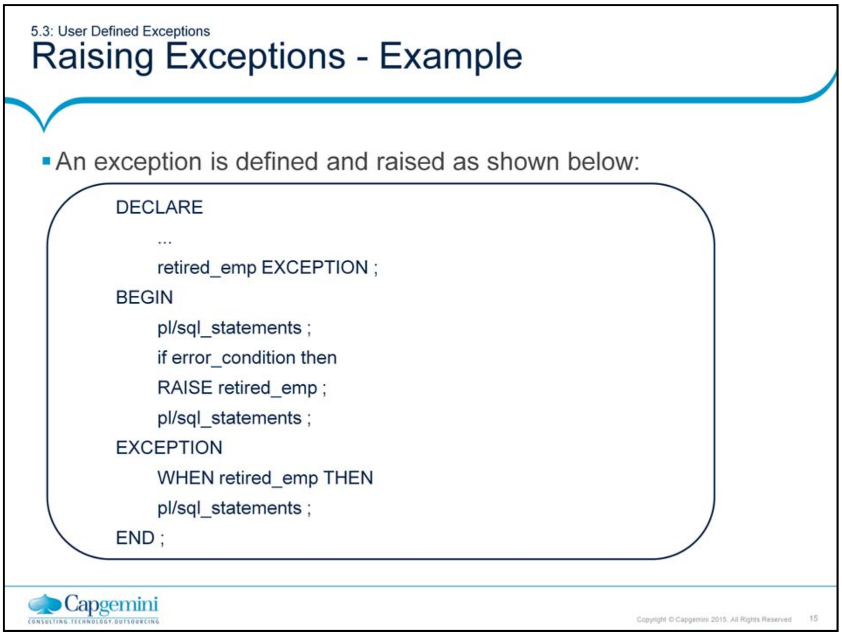
- Raising Exceptions:
  - Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that are associated with an Oracle error number using EXCEPTION_INIT.
  - Other user-defined exceptions must be raised explicitly by RAISE statements.
    - The syntax is:

```
RAISE  Exception_Name;
```

Raising Exceptions:

When the error associated with an exception occurs, the exception is raised.

This is done through the RAISE command.

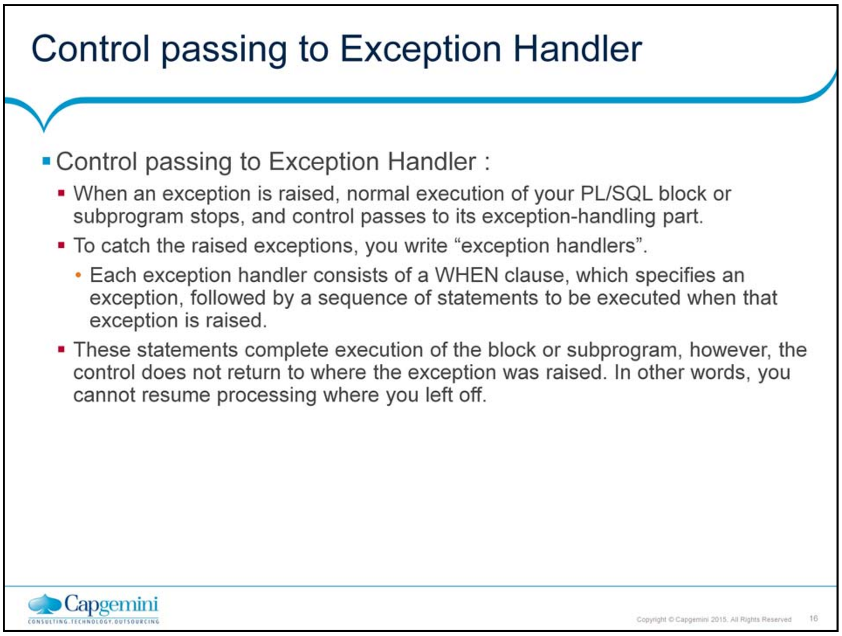


5.3: User Defined Exceptions

Raising Exceptions - Example

- An exception is defined and raised as shown below:

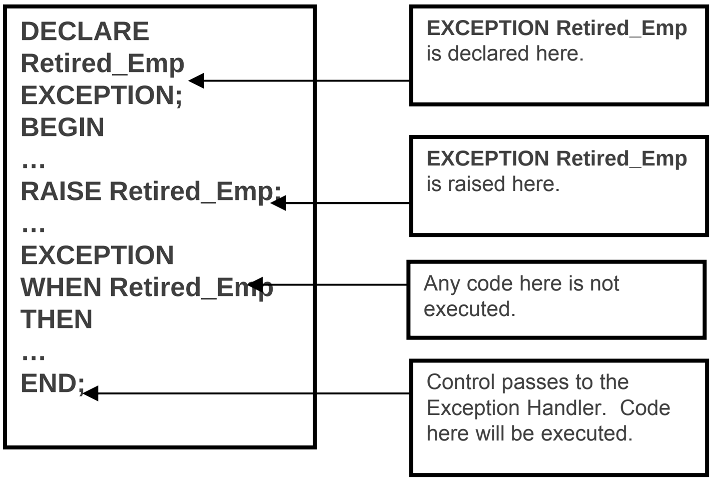
```
DECLARE
    ...
    retired_emp EXCEPTION ;
BEGIN
    pl/sql_statements ;
    if error_condition then
    RAISE retired_emp ;
    pl/sql_statements ;
EXCEPTION
    WHEN retired_emp THEN
    pl/sql_statements ;
END ;
```



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.

Control passing to Exception Handler:



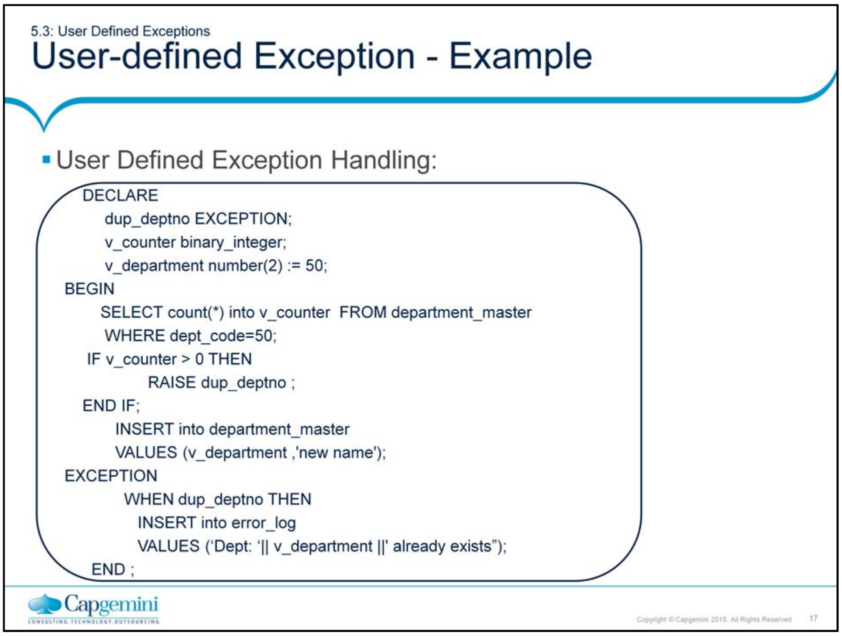
```
DECLARE
Retired_Emp
EXCEPTION;
BEGIN
…
RAISE Retired_Emp;
…
EXCEPTION
WHEN Retired_Emp
THEN
…
END;
```

EXCEPTION Retired_Emp is declared here.

EXCEPTION Retired_Emp is raised here.

Any code here is not executed.

Control passes to the Exception Handler. Code here will be executed.



5.3: User Defined Exceptions

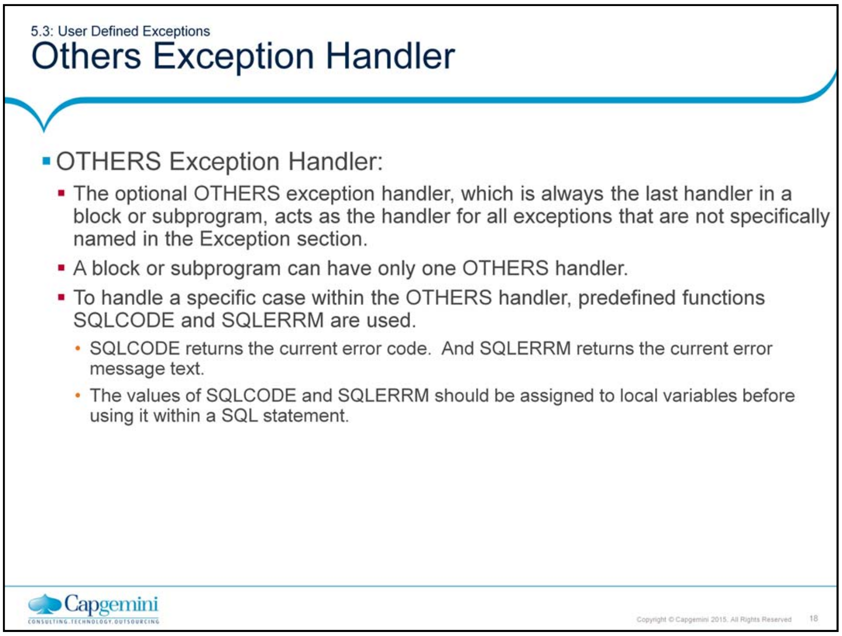
User-defined Exception - Example

- User Defined Exception Handling:

```
    DECLARE
        dup_deptno EXCEPTION;
        v_counter binary_integer;
        v_department number(2) := 50;
  BEGIN
        SELECT count(*) into v_counter  FROM department_master
        WHERE dept_code=50;
     IF v_counter > 0 THEN
              RAISE dup_deptno ;
    END IF;
          INSERT into department_master
          VALUES (v_department ,'new name');
  EXCEPTION
           WHEN dup_deptno THEN
             INSERT into error_log
             VALUES (‘Dept: ‘|| v_department ||' already exists”);
     END ;
```

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

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.

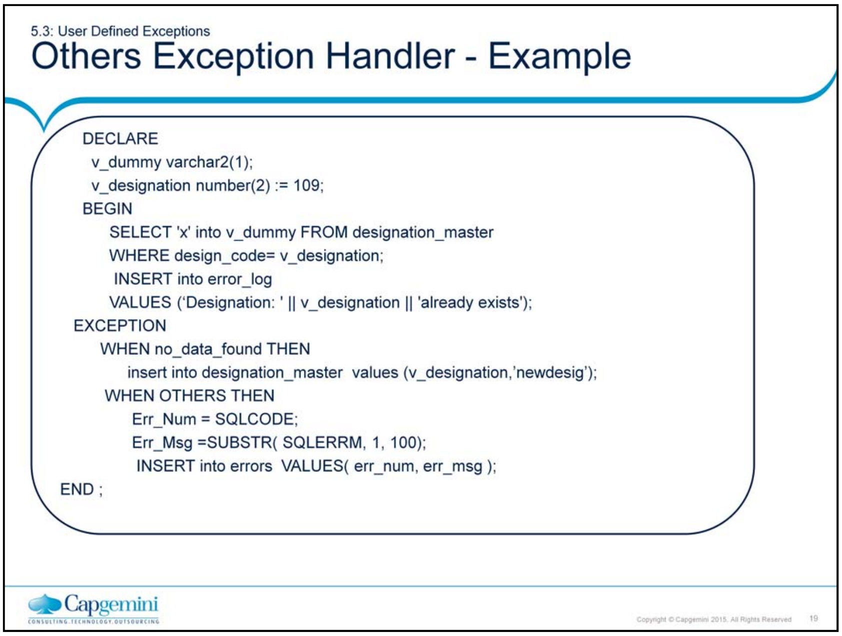


5.3: User Defined Exceptions

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



5.3: User Defined Exceptions

Others Exception Handler - Example

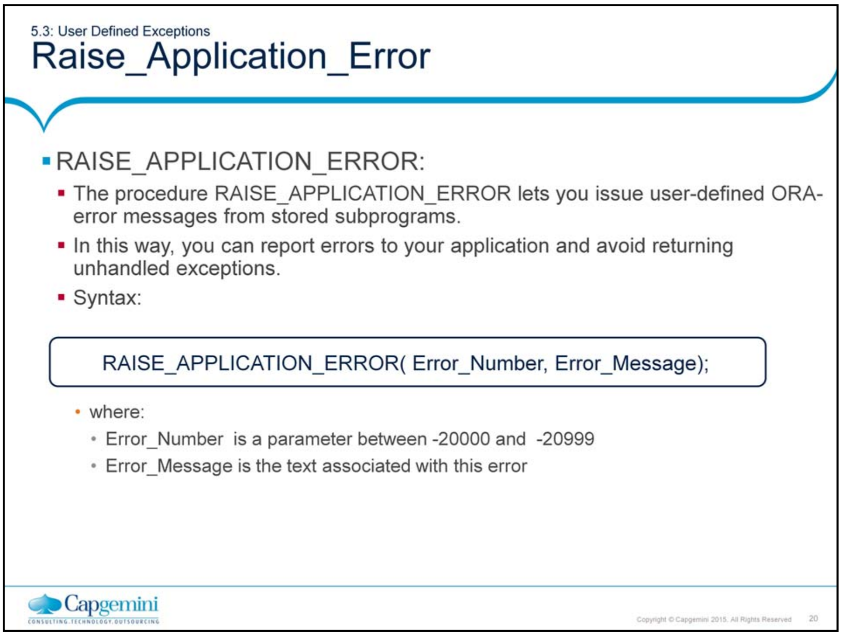
```
  DECLARE
   v_dummy varchar2(1);
   v_designation number(2) := 109;
  BEGIN
      SELECT 'x' into v_dummy FROM designation_master
      WHERE design_code= v_designation;
       INSERT into error_log
      VALUES (‘Designation: ' || v_designation || 'already exists');
 EXCEPTION
     WHEN no_data_found THEN
         insert into designation_master  values (v_designation,’newdesig’);
      WHEN OTHERS THEN
          Err_Num = SQLCODE;
          Err_Msg =SUBSTR( SQLERRM, 1, 100);
           INSERT into errors  VALUES( err_num, err_msg );
END ;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 19

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.



5.3: User Defined Exceptions

Raise_Application_Error

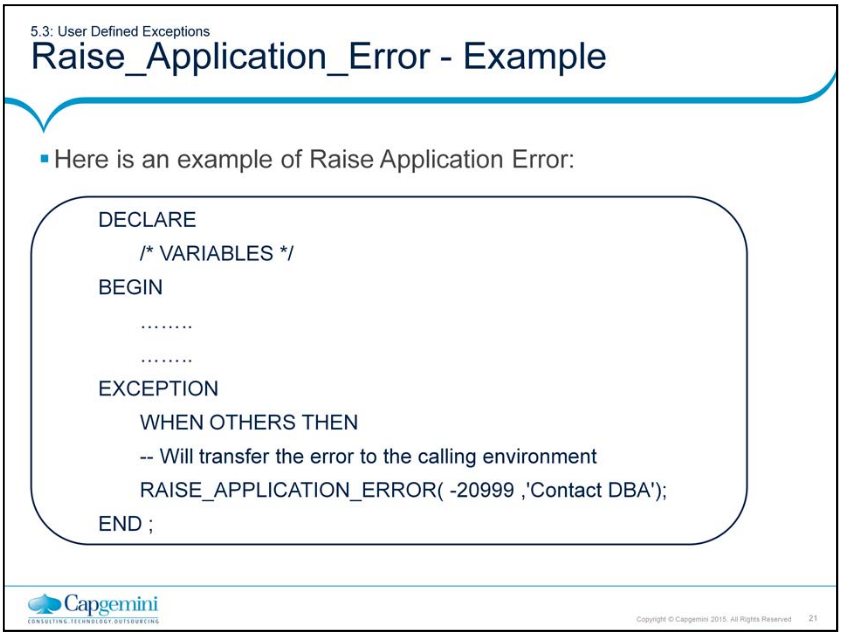
- RAISE_APPLICATION_ERROR:
  - The procedure RAISE_APPLICATION_ERROR lets you issue user-defined ORA-error messages from stored subprograms.
  - In this way, you can report errors to your application and avoid returning unhandled exceptions.
  - Syntax:

```
RAISE_APPLICATION_ERROR( Error_Number, Error_Message);
```

- where:
  - Error_Number is a parameter between -20000 and -20999
  - Error_Message is the text associated with this error

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.



5.3: User Defined Exceptions

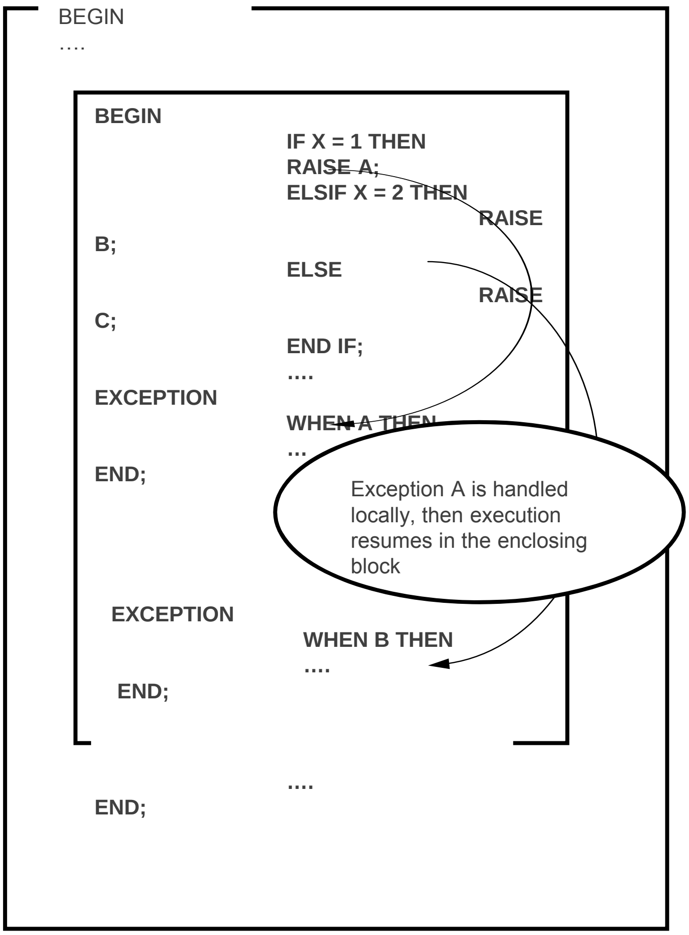
Raise_Application_Error - Example

- 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 ;
```

Propagation of Exceptions:

- When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, then the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search.



```
BEGIN
….
    BEGIN
                        IF X = 1 THEN
                        RAISE A;
                        ELSIF X = 2 THEN
                                                RAISE
    B;
                        ELSE
                                                RAISE
    C;
                        END IF;
                        ….
    EXCEPTION
                        WHEN A THEN
                        …
    END;

      EXCEPTION
                          WHEN B THEN
                          ….
       END;

                        ….
    END;
```

Exception A is handled locally, then execution resumes in the enclosing block

Masking Location of an Error:

- Since the same Exception section is examined for the entire block, it can be difficult to determine, which SQL statement caused the error.
- For example: Consider the following block:

```
SELECT
SELECT
SELECT
EXCEPTION
WHEN  NO_DATA_FOUND THEN
--You Don't Know  which caused the
NO_DATA_FOUND
END ;
```

➢ There are two methods to solve this problem:

```
DECLARE
V_Counter  NUMBER:= 1;
BEGIN
SELECT …….
V_Counter := 2;
SELECT  …..
V_Counter :=3;
SELECT  …
WHEN NO_DATA_FOUND THEN
 -- Check values of V_Counter to  find out which
SELECT statement
 -- caused the exception NO_DATA_FOUND
END ;
```

➢ The second method is to put the statement in its own block:

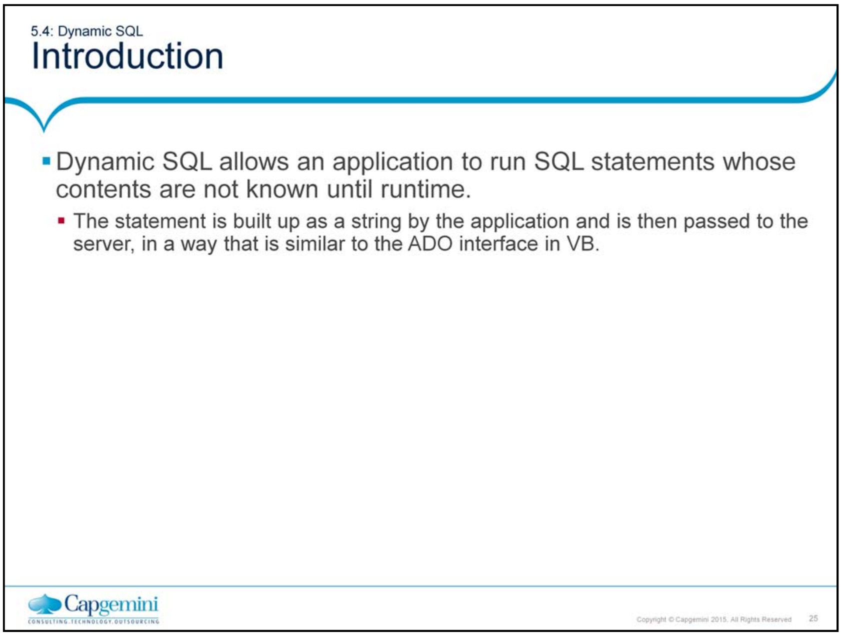
```
BEGIN
-- PL/SQL Statements
BEGIN
SELECT ….
EXCEPTION
WHEN NO_DATA_FOUND THEN
        --
END;
BEGIN
SELECT ….
EXCEPTION
WHEN NO_DATA_FOUND THEN
        --
END;
BEGIN
SELECT ….
EXCEPTION
WHEN NO_DATA_FOUND THEN
        --
END;
END ;
```

contd.

Masking Location of an Error (contd.):

```
BEGIN
--------
./* PL/SQL statements */
BEGIN
SELECT ……..
WHEN  NO_DATA_FOUND  THEN
-- Process  the error for NO_DATA_FOUND
END;

/* Some more PL/SQL statements
This will execute irrespective of when
NO_DATA_FOUND */
  END;
```



5.4: Dynamic SQL

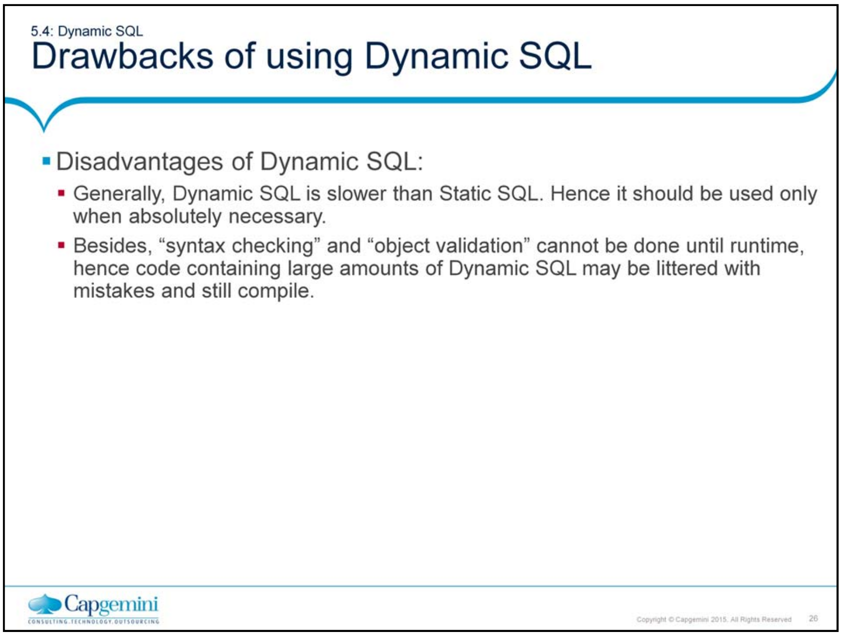
Introduction

- Dynamic SQL allows an application to run SQL statements whose contents are not known until runtime.
  - The statement is built up as a string by the application and is then passed to the server, in a way that is similar to the ADO interface in VB.

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

Dynamic SQL:

- Most database applications do a specific job.
- For example:
  - ➢ A simple program might prompt the user for an employee number, then update rows in the EMP and DEPT tables.
  - ➢ In this case, you know the makeup of the UPDATE statement at pre-compile time. That is, you know which tables might be changed, the constraints defined for each table and column, the columns that might be updated, and the datatype of each column.
- However, some applications must accept (or build), and process a variety of SQL statements at run time.
- For example:
  - ➢ A general-purpose report writer must build different SELECT statements for the various reports it generates.
  - ➢ In this case, the statement's makeup is unknown until run time. Such statements can, and probably will, change from execution to execution. They are aptly called “Dynamic SQL statements”.
- “Dynamic SQL statements” are stored in character strings built by your program at run time. Such strings must contain the text of a valid SQL statement or PL/SQL block. They can also contain placeholders for bind arguments.
  - ➢ A placeholder is an undeclared identifier, so its name, to which you must prefix a colon, does not matter.



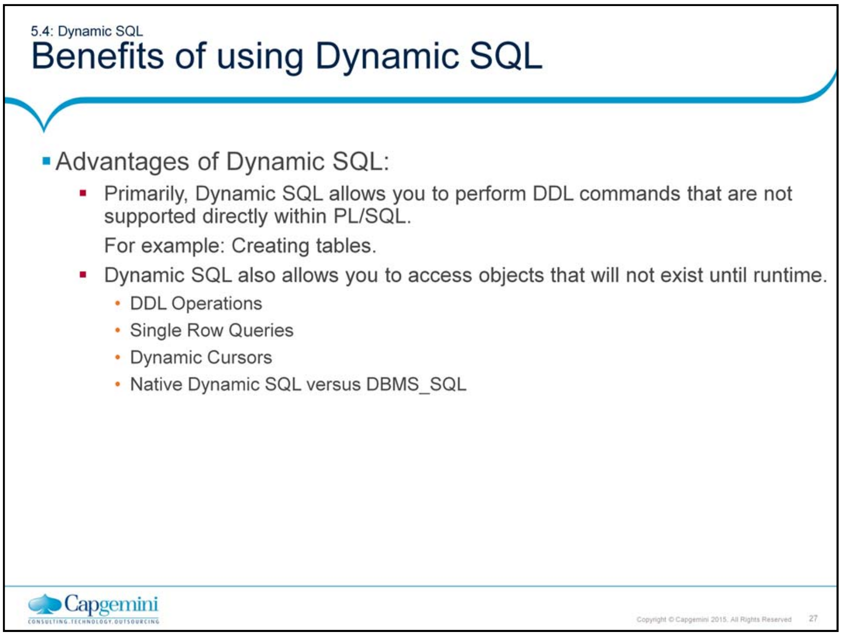
5.4: Dynamic SQL

Drawbacks of using Dynamic SQL

- Disadvantages of Dynamic SQL:
  - Generally, Dynamic SQL is slower than Static SQL. Hence it should be used only when absolutely necessary.
  - Besides, “syntax checking” and “object validation” cannot be done until runtime, hence code containing large amounts of Dynamic SQL may be littered with mistakes and still compile.

Disadvantages of Dynamic SQL:

Some dynamic queries require complex coding, the use of special data structures, and more runtime processing. While you might not notice the added processing time, you might find the coding difficult unless you fully understand dynamic SQL concepts and methods.


5.4: Dynamic SQL

Benefits of using Dynamic SQL

- Advantages of Dynamic SQL:
  - Primarily, Dynamic SQL allows you to perform DDL commands that are not supported directly within PL/SQL.

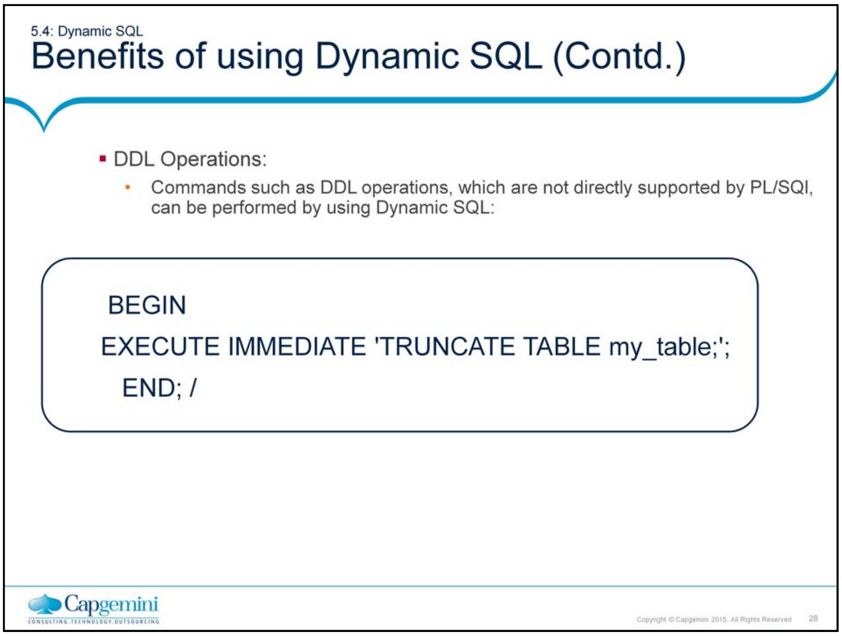
    For example: Creating tables.
  - Dynamic SQL also allows you to access objects that will not exist until runtime.
    - DDL Operations
    - Single Row Queries
    - Dynamic Cursors
    - Native Dynamic SQL versus DBMS_SQL

Advantages of Dynamic SQL:

- Host programs that accept and process dynamically defined SQL statements are more versatile than plain embedded SQL programs.
- Dynamic SQL statements can be built interactively with input from users having little or no knowledge of SQL.
- **For example:** Your program might simply prompt users for a search condition to be used in the WHERE clause of a SELECT, UPDATE, or DELETE statement. A more complex program might allow users to choose from menus listing SQL operations, table and view names, column names, and so on. Thus, Dynamic SQL lets you write highly flexible applications.

When to use Dynamic SQL:

- In practice, Static SQL will meet nearly all your programming needs. Use Dynamic SQL only if you need its open-ended flexibility. Its use is suggested when one of the following items is unknown at pre-compile time:
  - Text of the SQL statement (commands, clauses, and so on)
  - The number of host variables
  - The datatypes of host variables
  - References to database objects such as columns, indexes, sequences, tables, usernames, and views

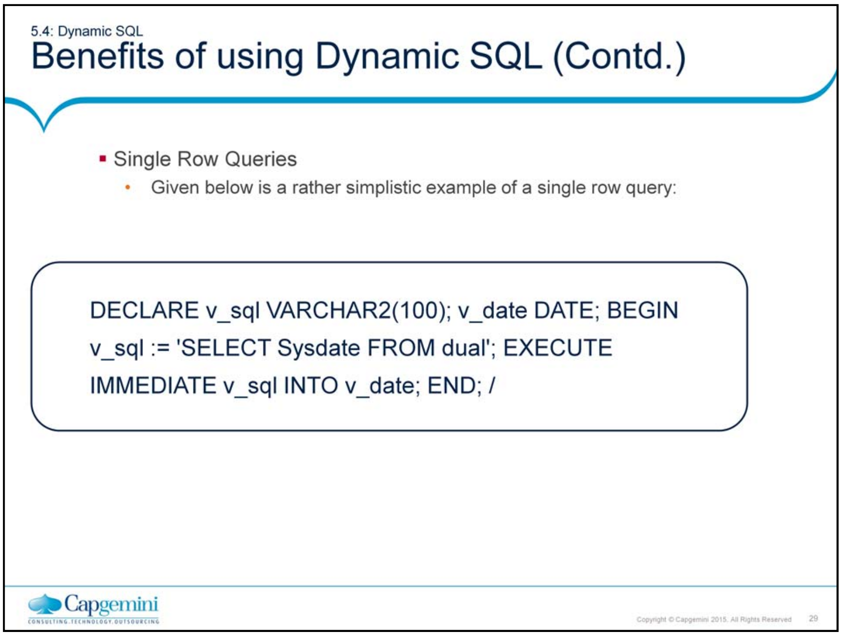


5.4: Dynamic SQL

Benefits of using Dynamic SQL (Contd.)

- DDL Operations:
  - Commands such as DDL operations, which are not directly supported by PL/SQl, can be performed by using Dynamic SQL:

```
 BEGIN
EXECUTE IMMEDIATE 'TRUNCATE TABLE my_table;';
  END; /
```



5.4: Dynamic SQL

Benefits of using Dynamic SQL (Contd.)

- Single Row Queries
  - Given below is a rather simplistic example of a single row query:

```
DECLARE v_sql VARCHAR2(100); v_date DATE; BEGIN
v_sql := 'SELECT Sysdate FROM dual'; EXECUTE
IMMEDIATE v_sql INTO v_date; END; /
```

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

How Dynamic SQL statements are processed?

- Typically, an application program prompts the user for the text of a SQL statement, and the values of host variables used in the statement. Then Oracle parses the SQL statement. That is, Oracle examines the SQL statement to make sure it follows syntax rules and refers to valid database objects. Parsing also involves checking database access rights, reserving needed resources, and finding the optimal access path.
- Next, Oracle binds the host variables to the SQL statement. That is, Oracle gets the addresses of the host variables so that it can read or write their values.
- Then Oracle executes the SQL statement. That is, Oracle does what the SQL statement requested, such as deleting rows from a table.
- The SQL statement can be executed repeatedly by using new values for the host variables.
- There are many constructs available in Dynamic SQL, which allow a developer to use them from time to time.
- **For Example:** EXECUTE IMMEDIATE

EXECUTE IMMEDIATE

Syntax:

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable[, define_variable]... |
record}]
[USING [IN | OUT | IN OUT] bind_argument
   [, [IN | OUT | IN OUT] bind_argument]...];
```

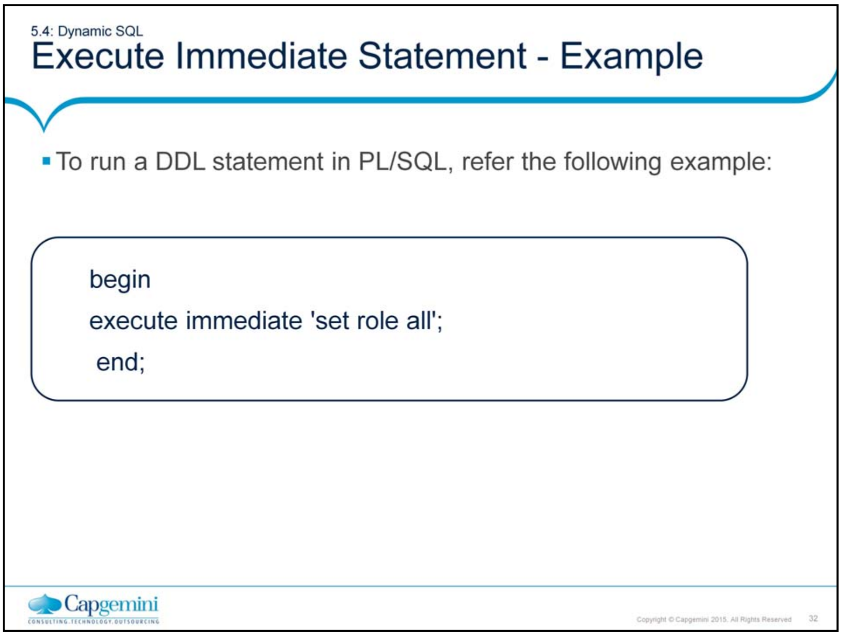
where

- ➢ dynamic_string is a string expression that represents a SQL statement or PL/SQL block
- ➢ define_variable is a variable that stores a SELECTed column value
- ➢ record is a user-defined or %ROWTYPE record that stores a SELECTed row
- ➢ bind_argument is an expression whose value is passed to the dynamic SQL statement or PL/SQL block.

contd.

How Dynamic SQL statements are processed? (contd.)

- The dynamic_string can contain any valid SQL statement (except multi row select query), or a valid PL/SQL block without the terminator. The string can also contain placeholder for bind variables. However there is a restriction, you cannot use placeholder for defining schema objects (table name etc.)
- The INTO clause used for single row queries, specifies the columns or the record type variable into which the column values are fetched. For each value returned by the query, there must be a type compatible variable defined in the INTO clause.
- You must put all the bind arguments in the USING clause. The default parameter mode is IN. At run time, any bind arguments in the USING clause replace corresponding placeholders in the SQL statement or PL/SQL block. So, every placeholder must be associated with a bind argument in the USING clause. Only numeric, character and string literals are allowed in the USING clause. Boolean variables like TRUE, FALSE and NULLs are not permitted in the USING clause.



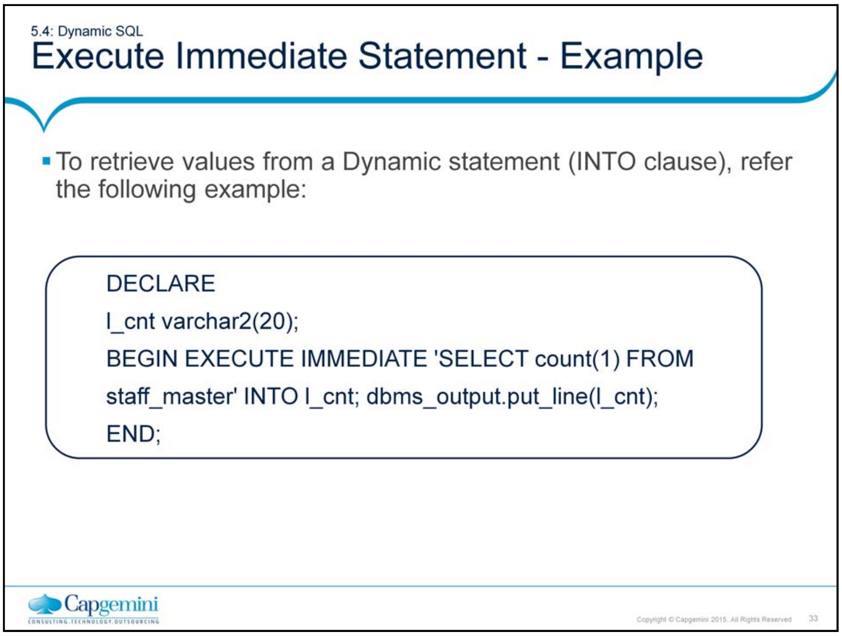
5.4: Dynamic SQL

Execute Immediate Statement - Example

- To run a DDL statement in PL/SQL, refer the following example:

```
begin
execute immediate 'set role all';
 end;
```

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING


5.4: Dynamic SQL

Execute Immediate Statement - Example

- To retrieve values from a Dynamic statement (INTO clause), refer the following example:

```
DECLARE
l_cnt varchar2(20);
BEGIN EXECUTE IMMEDIATE 'SELECT count(1) FROM
staff_master' INTO l_cnt; dbms_output.put_line(l_cnt);
END;
```

Copyright © Capgemini 2015. All Rights Reserved 33

More Examples:

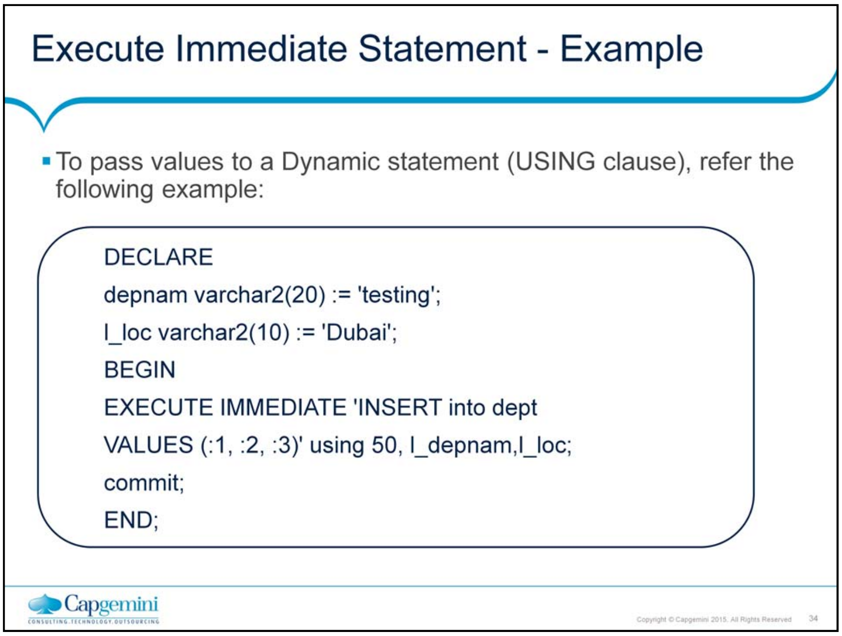
- To dynamically call a routine, refer following example:
  - ➢ The bind variables used for parameters of the routine have to be specified along with the parameter type.
  - ➢ IN type is the default, others have to be explicitly specified.

```
DECLARE
 l_routin varchar2(100) := 'gen2161.get_rowcnt';
l_tblnam varchar2(20) := 'emp'; l_cnt number;
 l_status varchar2(200);
BEGIN EXECUTE IMMEDIATE 'BEGIN ' || l_routin ||
'(:2, :3, :4); END;' using in l_tblnam, out l_cnt, in out
l_status;
IF l_status != 'OK' THEN dbms_output.put_line('error');
END IF;
 END;
```

To return value into a PL/SQL record type, refer the following example.

- ➢ The same option can be used for %rowtype variables, as well.

```
DECLARE
     type empdtlrec is record (empno number(4),
ename varchar2(20), deptno number(2)); empdtl
empdtlrec; BEGIN
    EXECUTE IMMEDIATE 'SELECT staff_code,
staff_name, deptcode ' || 'FROM staff_master
WHERE staff_code = 100001' into empdtl;
   END;
```



Execute Immediate Statement - Example

- To pass values to a Dynamic statement (USING clause), refer the following example:

```
DECLARE
depnam varchar2(20) := 'testing';
l_loc varchar2(10) := 'Dubai';
BEGIN
EXECUTE IMMEDIATE 'INSERT into dept
VALUES (:1, :2, :3)' using 50, l_depnam,l_loc;
commit;
END;
```

<u>More Examples</u>:

- To pass and retrieve values, refer the following example:
  - ➢ The INTO clause should precede the USING clause.

```
DECLARE
  l_dept pls_integer := 20;
  l_nam varchar2(20);
  l_loc varchar2(20);
  BEGIN
  execute immediate 'select dname, loc from
dept where deptno = :1' into l_nam, l_loc using
l_dept ;
 END;
```

Some more examples of EXECUTE IMMEDIATE:

- **Example 1:**

```
DECLARE
sql_stmt varchar2(100);
plsql_block varchar2(200);
my_deptno number(2) := 50;
my_dname varchar2(15) := 'PERSONNEL';
my_loc varchar2(15) := 'DALLAS';
emp_rec emp%ROWTYPE;
BEGIN
.
.
.
```

```
.
.
sql_stmt := 'INSERT INTO dept VALUES (:1, :2,
:3)';
EXECUTE IMMEDIATE sql_stmt
USING my_deptno, my_dname, my_loc;
```

- **Example 1a:**

  Example to use INSERT. Here 3 place holder columns :1, :2, :3 are used.

```
.
.
.
EXECUTE IMMEDIATE
'DELETE FROM dept  WHERE deptno = :n'
USING my_deptno;
```

- **Example 1b:**

  Example to use DELETE. Here we are using the SQL statement directly with EXECUTE IMMEDIATE.

```
.
.
.
sql_stmt := 'SELECT * FROM emp WHERE
empno = :id';
EXECUTE IMMEDIATE sql_stmt INTO
emp_rec USING 7788;
```

- **Example 1c:**

  Example to use INTO clause. In this case entire record is stored in emp_rec for a specific employee code.

contd.

Some more examples of EXECUTE IMMEDIATE (contd.):

- **Example 1d:**

  Example to use PL/SQL. Here we are calling a stored procedure.

```
.
.
plsql_block := 'BEGIN
emp_stuff.raise_salary(:id,
amt);END;';
EXECUTE IMMEDIATE plsql_block USING
7788, 500;
```

- **Example 1e:**

  Example to use DDL.

```
.
.
EXECUTE IMMEDIATE
            'CREATE TABLE bonus (id
NUMBER, amt NUMBER)';
```

- **Example 1f:**

  Example to use Alter Session

```
.
.
sql_stmt := 'ALTER SESSION SET
NLS_DATE_FORMAT='MM/DD/YYYY'';
EXECUTE IMMEDIATE sql_stmt;
END;
```

- **Example 2:**

  The following procedure accepts two parameters, namely table name and an optional WHERE clause. If the WHERE clause is NULL, then it deletes all the rows from the table, otherwise it deletes the rows which satisfy the criteria.

```
CREATE PROCEDURE Delete_Rows
(table_name IN VARCHAR2,
  condition IN varchar2 DEFAULT NULL)
AS
            where_clause varchar2(100) := '
WHERE ' || condition;
BEGIN
  IF condition IS NULL THEN
            where_clause := NULL;
  END IF;
  EXECUTE IMMEDIATE 'DELETE FROM ' ||
            table_name || where_clause;
EXCEPTION
  WHEN OTHERS THEN
            RAISE_APPLICATION_ERROR(-
20001,
            'Error in performing operation');
END;
```

contd.

<u>Some more examples of EXECUTE IMMEDIATE usage (contd.)</u>:

- **Example 3:**

  Suppose you want to pass NULLs to a Dynamic SQL statement.

  Then, you might write the following EXECUTE IMMEDIATE statement:

```
EXECUTE IMMEDIATE 'UPDATE emp SET
comm = :x' USING NULL;
```

  However, this statement fails with a bad expression error because the literal NULL is not allowed in the USING clause. To work around this restriction, simply replace the keyword NULL with an uninitialized variable, as follows :

```
DECLARE
  a_null CHAR(1); -- set to NULL automatically
at run time
BEGIN
  EXECUTE IMMEDIATE 'UPDATE emp SET
comm = :x' USING a_null;
END;
```

<u>OPEN-FOR, FETCH, and CLOSE Statements</u>:

- The combination of these three statements allows you to perform multi-row select query. The concept is similar to cursors used earlier.
- There are three steps involved:
  - Open a cursor
  - Fetch rows from the cursor
  - Close the cursor
- Syntax:

```
OPEN {cursor_variable | :host_cursor_variable}
FOR dynamic_string
[USING bind_argument[,bind_argument]...];
```

where:

  - Cursor_variable is a cursor of weak type (ref cursor)
  - host_cursor_variable is a cursor variable declared in a PL/SQL host environment such as an Pro*c or SQL*J program
  - dynamic_string is a string expression that represents a multi-row query
- Syntax:

```
FETCH {cursor_variable | :host_cursor_variable}
  INTO {define_variable[, define_variable]... |
record};
```

where:

  - Cursor_variable and :host_cursor_variable has the same meaning as in OPEN.
- Syntax:

```
CLOSE {cursor_variable |
:host_cursor_variable};
```

where:

  - Cursor_variable and :host_cursor_variable has the same meaning as in OPEN.

contd.

OPEN-FOR, FETCH, and CLOSE Statements (contd.)

- **Example 1:**

```
DECLARE
                TYPE EmpCurTyp IS REF
CURSOR; --define weak REF CURSOR type
                emp_cv   EmpCurTyp;  -- declare
cursor variable
                my_ename VARCHAR2(15);
                my_sal   NUMBER := 1000;
BEGIN
                OPEN emp_cv FOR
                                                -- open cursor
variable
           'SELECT ename, sal FROM emp
WHERE sal > :s'
            USING my_sal;
                LOOP
                                                FETCH
emp_cv INTO my_ename, my_sal;  -- fetch
next row
                EXIT WHEN
emp_cv%NOTFOUND;  -- exit loop when last
row
  -- is fetched
--process row
                                                ..
                END LOOP;
                CLOSE emp_cv; -- close cursor
variable
END;
```

- **Example 2:** The following example allows you to fetch rows from the result set of a dynamic multi-row query into a record:

```
DECLARE
TYPE EmpCurTyp IS REF CURSOR;
emp_cv EmpCurTyp;
emp_rec emp%ROWTYPE;
sql_stmt VARCHAR2(100);
my_job VARCHAR2(15) := 'CLERK';
BEGIN
sql_stmt := 'SELECT * FROM emp WHERE
job = :j';
OPEN emp_cv FOR sql_stmt USING
my_job;
LOOP
FETCH emp_cv INTO emp_rec;
EXIT WHEN emp_cv%NOTFOUND;
        --process record
                                                ..
                                                ..
END LOOP;
CLOSE emp_cv;
END;
```

Specifying MODE for parameters:

- You need not specify a parameter mode for input bind arguments (those used, for example, in the WHERE clause) because the mode defaults to IN.
- You can specify the OUT mode for output bind arguments used in the RETURNING clause of an INSERT, UPDATE, or DELETE statement.
- **For example:**

```
DECLARE
sql_stmt VARCHAR2(100);
old_loc VARCHAR2(15);
BEGIN
sql_stmt := 'DELETE FROM dept WHERE
deptno = 20 RETURNING loc INTO :x';
EXECUTE IMMEDIATE sql_stmt USING OUT
old_loc;
...
END;
```

- Likewise, when appropriate, you must specify the OUT or IN OUT mode for bind arguments passed as parameters.
- **For example:** Suppose you want to call the following stand-alone procedure:

```
CREATE PROCEDURE Create_Dept (
                                    deptno INOUT
number,
                                    dname  IN
varchar2,
                                    loc    IN varchar2)
AS
BEGIN
  deptno := deptno_seq.NEXTVAL;
  INSERT INTO dept VALUES (deptno, dname, loc);
END;
```

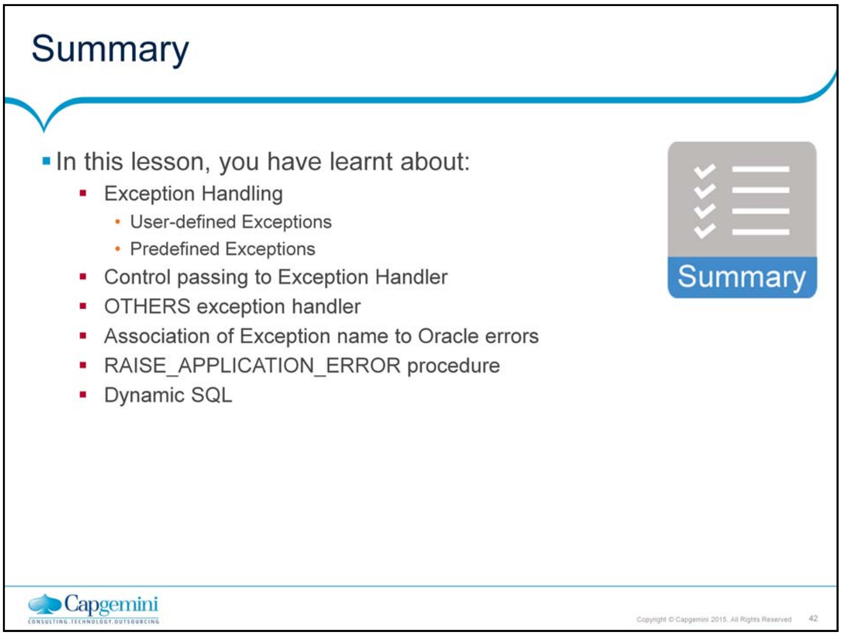
- To call the procedure from a dynamic PL/SQL block, you must specify the IN OUT mode for the bind argument associated with formal parameter deptno, as follows:

```
DECLARE
  plsql_block varchar2(200);
  new_deptno  number(2);
  new_dname   varchar2(15) := 'ADVERTISING';
  new_loc     varchar2(15) := 'NEW YORK';
BEGIN
  plsql_block := 'BEGIN create_dept(:a, :b, :c);
END;';
  EXECUTE IMMEDIATE plsql_block
    USING IN OUT new_deptno, new_dname,
new_loc;
  IF new_deptno > 90 THEN
...
END;
```

Using Duplicate Placeholders:

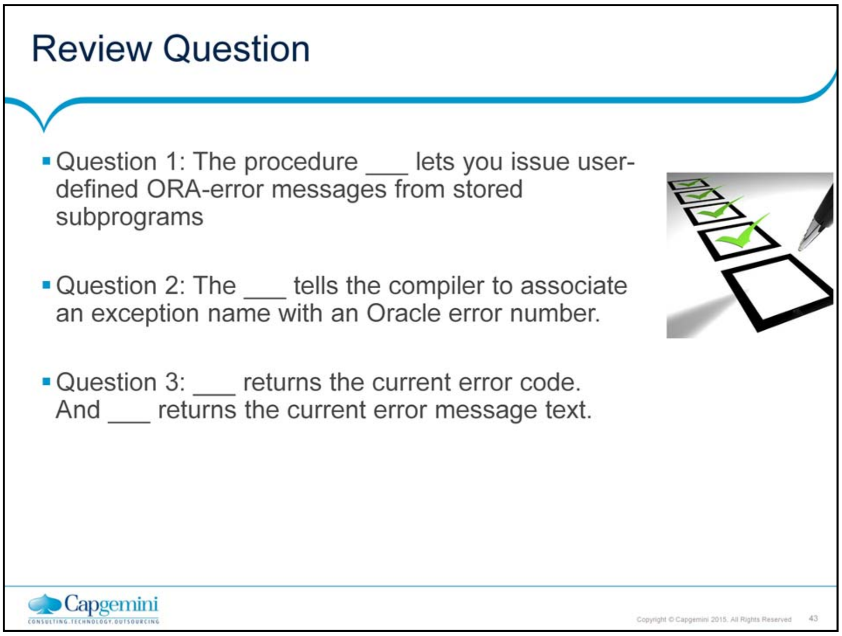
- Placeholders in a Dynamic SQL statement are associated with bind arguments in the USING clause by position, and not by name. So, if the same placeholder appears two or more times in the SQL statement, each appearance must correspond to a bind argument in the USING clause.
- However, only the unique placeholders in a Dynamic PL/SQL block are associated with bind arguments in the USING clause by position. So, if the same placeholder appears two or more times in a PL/SQL block, all appearances correspond to one bind argument in the USING clause. In the example shown below, the first unique placeholder (x) is associated with the first bind argument (a). Likewise, the second unique placeholder (y) is associated with the second bind argument (b).

```
DECLARE
a number := 4;
b number := 7;
BEGIN
plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x);
END;'
EXECUTE IMMEDIATE plsql_block USING a,
b;
...
END;
```



Summary

- In this lesson, you have learnt about:
  - Exception Handling
    - User-defined Exceptions
    - Predefined Exceptions
  - Control passing to Exception Handler
  - OTHERS exception handler
  - Association of Exception name to Oracle errors
  - RAISE_APPLICATION_ERROR procedure
  - Dynamic SQL



Review Question

- Question 1: The procedure ___ lets you issue user-defined ORA-error messages from stored subprograms
- Question 2: The ___ tells the compiler to associate an exception name with an Oracle error number.
- Question 3: ___ returns the current error code. And ___ returns the current error message text.