Oracle for Developers (PLSQL)

Appendices

Appendix A.

Built-in Packages in Oracle

DBMS_OUTPUT: Enabling and Disabling Output

- DBMS_OUTPUT provides a mechanism for displaying information from the PL/SQL program on to your screen (that is your session's output device).
- The DBMS_OUTPUT package is created when the Oracle database is installed.
- The “dbmsoutp.sql” script contains the source code for the specification of this package.
- This script is called by the “catproc.sql” script, which is normally run immediately after database creation.
- The catproc.sql script creates the public synonym DBMS_OUTPUT for the package.
- Instance-wise access to this package is provided on installation, so no additional steps should be necessary in order to use DBMS_OUTPUT.

DBMS_OUTPUT Program Names

Name: DISABLE

Description : Disables output from the package; the DBMS_OUTPUT buffer will not be flushed to the screen.

Name: ENABLE

Description : Enables output from the package.

Name: GET_LINE

Description : Gets a single line from the buffer.

Name: GET_LINES

Description : Gets specified number of lines from the buffer and passes them into a PL/SQL table.

Name: NEW_LINE

Description : Inserts an end-of-line mark in the buffer.

Name: PUT

Description : Puts information into the buffer.

Name: PUT_LINE

Description : Puts information into the buffer and appends an end-of-line marker after that data.

DBMS_OUTPUT Concepts:

- Each user has a DBMS_OUTPUT buffer of up to 1,000,000 bytes in size. You can write information to this buffer by calling the DBMS_OUTPUT.PUT and DBMS_OUTPUT.PUT_LINE programs.
  - If you are using DBMS_OUTPUT from within SQL*Plus, this information will be automatically displayed when your program terminates.
  - You can (optionally) explicitly retrieve information from the buffer with calls to DBMS_OUTPUT.GET and DBMS_OUTPUT.GET_LINE.
- The DBMS_OUTPUT buffer can be set to a size between 2,000 and 1,000,000 bytes with the DBMS_OUTPUT.ENABLE procedure.
  - If you do not enable the package, no information will be displayed or be retrievable from the buffer.
- The buffer stores three different types of data in their internal representations, namely VARCHAR2, NUMBER, and DATE.
  - These types match the overloading available with the PUT and PUT_LINE procedures.
  - Note that DBMS_OUTPUT does not support Boolean data in either its buffer or its overloading of the PUT procedures.

DBMS_OUTPUT Exceptions:

- DBMS_OUTPUT does not contain any declared exceptions. Instead, Oracle designed the package to rely on two error numbers in the -20 NNN range (usually reserved for Oracle customers). You may, therefore, encounter one of these two exceptions when using the DBMS_OUTPUT package (no names are associated with these exceptions).
  - The -20000 error number indicates that these package-specific exceptions were raised by a call to RAISE_APPLICATION_ERROR, which is in the DBMS_STANDARD package.
- -20000
  - ORU-10027: buffer overflow, limit of <buf_limit> bytes.
  - If you receive the -10027 error, you should see if you can increase the size of your buffer with another call to DBMS_OUTPUT.ENABLE.
- -20000
  - ORU-10028: line length overflow, limit of 255 bytes per line.
  - If you receive the -10028 error, you should restrict the amount of data you are passing to the buffer in a single call to PUT_LINE, or in a batch of calls to PUT followed by NEW_LINE.
- You may also receive the ORA-06502 error:
  - ORA-06502
  - It is a numeric or value error.
  - If you receive the -06502 error, you have tried to pass more than 255 bytes of data to DBMS_OUTPUT.PUT_LINE. You must break up the line into more than one string.

Drawbacks of DBMS_OUTPUT:

- Before learning all about the DBMS_OUPUT package, and rushing to use it, you should be aware of several drawbacks with the implementation of this functionality:
  - The "put" procedures that place information in the buffer are overloaded only for strings, dates, and numbers. You cannot request the display of Booleans or any other types of data. You cannot display combinations of data (a string and a number, for instance), without performing the conversions and concatenations yourself.
  - You will see output from this package only after your program completes its execution. You cannot use DBMS_OUTPUT to examine the results of a program while it is running. And if your program terminates with an unhandled exception, you may not see anything at all!
  - If you try to display strings longer than 255 bytes, DBMS_OUTPUT will raise a VALUE_ERROR exception.
  - DBMS_OUTPUT is not a strong choice as a report generator, because it can handle a maximum of only 1,000,000 bytes of data in a session before it raises an exception.
  - If you use DBMS_OUTPUT in SQL*Plus, you may find that any leading blanks are automatically truncated. Also, attempts to display blank or NULL lines are completely ignored.
- There are workarounds for almost every one of these drawbacks. The solution invariably requires the construction of a package that encapsulates and hides DBMS_OUTPUT.

- Writing to DBMS_OUTPUT buffer

You can write information to the DBMS_OUTPUT buffer with calls to the PUT, NEW_LINE, and PUT_LINE procedures.

The DBMS_OUTPUT.PUT procedure:

The PUT procedure puts information into the buffer, but does not append a newline marker into the buffer.

Use PUT if you want to place information in the buffer (usually with more than one call to PUT), but not also automatically issue a newline marker.

- The specification for PUT is overloaded, so that you can pass data in its native format to the package without having to perform conversions.

```
PROCEDURE DBMS_OUTPUT.PUT (A VARCHAR2);
PROCEDURE DBMS_OUTPUT.PUT (A NUMBER);
PROCEDURE DBMS_OUTPUT.PUT (A DATE);
```

where A is the data being passed.

Example:

- In the following example, three simultaneous calls to PUT, place the employee name, department ID number, and hire date into a single line in the DBMS_OUTPUT buffer:

```
DBMS_OUTPUT.PUT (:employee.lname || ', ' ||
:employee.fname);
DBMS_OUTPUT.PUT (:employee.department_id);
DBMS_OUTPUT.PUT (:employee.hiredate);
```

- If you follow these PUT calls with a NEW_LINE call, that information can then be retrieved with a single call to GET_LINE.

The DBMS_OUTPUT.NEW_LINE procedure:

The NEW_LINE procedure inserts an end-of-line marker in the buffer.

Use NEW_LINE after one or more calls to PUT in order to terminate those entries in the buffer with a newline marker.

Given below is the specification for NEW_LINE:

```
PROCEDURE DBMS_OUTPUT.NEW_LINE
```

The DBMS_OUTPUT.PUT_LINE procedure:

The PUT_LINE procedure puts information into the buffer, and then appends a newline marker into the buffer.

The specification for PUT_LINE is overloaded, so that you can pass data in its native format to the package without having to perform conversions.

```
PROCEDURE DBMS_OUTPUT.PUT_LINE (A
VARCHAR2);
PROCEDURE DBMS_OUTPUT.PUT_LINE (A NUMBER);
PROCEDURE DBMS_OUTPUT.PUT_LINE (A DATE);
```

where A is the data being passed

- The PUT_LINE procedure is the one most commonly used in SQL*Plus to debug PL/SQL programs.
- When you use PUT_LINE in these situations, you do not need to call GET_LINE to extract the information from the buffer. Instead, SQL*Plus will automatically dump out the DBMS_OUTPUT buffer when your PL/SQL block finishes executing. (You will not see any output until the program ends.)

Writing to DBMS_OUTPUT buffer: DBMS_OUTPUT.PUT_LINE (contd.):
Example:

- Suppose that you execute the following three statements in SQL*Plus:

```
SQL> exec DBMS_OUTPUT.PUT ('I am');
SQL> exec DBMS_OUTPUT.PUT (' writing');
SQL> exec DBMS_OUTPUT.PUT ('a');
```

- You will not see anything, because PUT will place the information in the buffer, but will not append the newline marker. Now suppose you issue this next PUT_LINE command, namely:
  - SQL> exec DBMS_OUTPUT.PUT_LINE ('book!');
- Then you will see the following output:
  - I am writing a book!
- All of the information added to the buffer with the calls to PUT, patiently wait to be flushed out with the call to PUT_LINE. This is the behavior you will see when you execute individual calls at the SQL*Plus command prompt to the PUT programs.
- Suppose you place these same commands in a PL/SQL block, namely:

```
BEGIN
  DBMS_OUTPUT.PUT ('I am');
  DBMS_OUTPUT.PUT (' writing ');
  DBMS_OUTPUT.PUT ('a ');
  DBMS_OUTPUT.PUT_LINE ('book');
END;
/
```

- Then the output from this script will be exactly the same as that generated by this single call:
  - SQL> exec DBMS_OUTPUT.PUT_LINE ('I am writing a book!');

Retrieving Data from the DBMS_OUTPUT buffer:

You can retrieve information from the DBMS_OUTPUT buffer with call to the GET_LINE procedure.

The DBMS_OUTPUT.GET_LINE procedure:

The GET_LINE procedure retrieves one line of information from the buffer.

Given below is the specification for the procedure:

```
PROCEDURE DBMS_OUTPUT.GET_LINE    (line
OUT VARCHAR2,     status OUT INTEGER);
```

- If you are using DBMS_OUTPUT from within SQL*Plus, however, you will never need to call either of these procedures. Instead, SQL*Plus will automatically extract the information and display it on the screen for you.
- The GET_LINE procedure retrieves one line of information from the buffer.

- The parameters are summarized as shown below:

Parameter	**Description**
Line	Retrieved line of text
Status	GET request status

contd.

Retrieving Data from the DBMS_OUTPUT buffer (contd.):

The DBMS_OUTPUT.GET_LINE procedure (contd.)

- The line can have up to 255 bytes in it, which is not very long. If GET_LINE completes successfully, then status is set to 0. Otherwise, GET_LINE returns a status of 1.
- Note that even though the PUT and PUT_LINE procedures allow you to place information into the buffer in their native representations (dates as dates, numbers and numbers, and so forth), GET_LINE always retrieves the information into a character string. The information returned by GET_LINE is everything in the buffer up to the next newline character. This information might be the data from a single PUT_LINE or from multiple calls to PUT.
- **For example:**
  The following call to GET_LINE extracts the next line of information into a local PL/SQL variable:

```
FUNCTION get_next_line RETURN VARCHAR2
IS
  return_value VARCHAR2(255);
  get_status INTEGER;
BEGIN
  DBMS_OUTPUT.GET_LINE (return_value,
get_status);
  IF get_status = 0
  THEN
    RETURN return_value;
  ELSE
    RETURN NULL;
  END IF;
END;
```

The UTL_FILE package

The UTL_FILE package is created when the Oracle database is installed. The utlfile.sql script (found in the built-in packages source code directory) contains the source code for the specification of this package. This script is called by catproc.sql, which is normally run immediately after database creation. The script creates the public synonym UTL_FILE for the package and grants EXECUTE privilege on the package to public.

Table: UTL_FILE programs

Name : FCLOSE

Description : Closes the specified files.

Name : FCLOSE_ALL

Description : Closes all open files.

Name : FFLUSH

Description : Flushes all the data from the UTL_FILE buffer.

Name : FOPEN

Description : Opens the specified file.

Name : GET_LINE

Description : Gets the next line from the file.

Name : IS_OPEN

Description : Returns TRUE if the file is already open.

Name : NEW_LINE

Description : Inserts a newline mark in the file at the end of the current line.

Name : PUT

Description : Puts text into the buffer.

Name : PUT_LINE

Description : Puts a line of text into the file.

Name : PUTF

Description : Puts formatted text into the buffer.

UTL_FILE: READING AND WRITING

- UTL_FILE is a package that has been welcomed warmly by PL/SQL developers. It allows PL/SQL programs to both “read from” and “write to” any operating system files that are accessible from the server on which your database instance is running.
  - You can now read ini files and interact with the operating system a little more easily than has been possible in the past.
  - You can directly load data from files into database tables while applying the full power and flexibility of PL/SQL programming.
  - You can directly generate reports from within PL/SQL without worrying about the maximum buffer restrictions of DBMS_OUTPUT.

File security:

- UTL_FILE lets you read and write files accessible from the server on which your database is running. So theoretically you can use UTL_FILE to write right over your tablespace data files, control files, and so on. That is of course not a very good idea.
- Server security requires the ability to place restrictions on where you can read and write your files.
- UTL_FILE implements this security by limiting access to files that reside in one of the directories specified in the INIT.ORA file for the database instance on which UTL_FILE is running.
- When you call FOPEN to open a file, you must specify both the location and the name of the file, in separate arguments. This file location is then checked against the list of accessible directories.
- Given below is the format of the parameter for file access in the INIT.ORA file:
  - utl_file_dir = <directory>
- Include a parameter for utl_file_dir for each directory you want to make accessible for UTL_FILE operations. For example: The following entries enable four different directories in UNIX:
  - utl_file_dir = /tmp
  - utl_file_dir = /ora_apps/hr/time_reporting
  - utl_file_dir = /ora_apps/hr/time_reporting/log
  - utl_file_dir = /users/test_area
- To bypass server security and allow read/write access to all directories, you can use the special syntax given below:
  - utl_file_dir = *

Specifying file locations:

- The location of the file is an operating system-specific string that specifies the directory or area in which to open the file. The location you provide must have been listed as an accessible directory in the INIT.ORA file for the database instance.
- The INIT.ORA location is a valid directory or area specification, as shown in the following examples:
  - In Windows NT:
  - 'k:\common\debug'
  - In UNIX:
  - '/usr/od2000/admin'
- Few examples are given below:
  - In Windows NT: file_id := UTL_FILE.FOPEN ('k:\common\debug', 'trace.lis', 'R');
  - In UNIX: file_id := UTL_FILE.FOPEN ('/usr/od2000/admin', 'trace.lis', 'W');
- Your location must be an explicit, complete path to the file. You cannot use operating system-specific parameters such as environment variables in UNIX to specify file locations.

UTL_FILE exceptions:

- The package specification of UTL_FILE defines seven exceptions. The cause behind a UTL_FILE exception can often be difficult to understand.
- Given below are the explanations Oracle provides for each of the exceptions:
  - **INVALID_PATH**
    The file location or the filename is invalid. Perhaps the directory is not listed as a utl_file_dir parameter in the INIT.ORA file (or doesn't exist as all), or you are trying to read a file and it does not exist.
  - **INVALID_MODE**
    The value you provided for the open_mode parameter in UTL_FILE.FOPEN was invalid. It must be “A”, “R”, or “W”.
  - **INVALID_FILEHANDLE**
    The file handle you passed to a UTL_FILE program was invalid. You must call UTL_FILE.FOPEN to obtain a valid file handle.
  - **INVALID_OPERATION**
    UTL_FILE could not open or operate on the file as requested. For example, if you try to write to a read-only file, you will raise this exception.
  - **READ_ERROR**
    The operating system returned an error when you tried to read from the file. (This does not occur very often.)
  - **WRITE_ERROR**
    The operating system returned an error when you tried to write to the file. (This does not occur very often.)
  - **INTERNAL_ERROR**
    Uh-oh. Something went wrong and the PL/SQL runtime engine couldn't assign blame to any of the previous exceptions. Better call Oracle Support!
- Programs in UTL_FILE may also raise the following standard system exceptions:
  - **NO_DATA_FOUND**
    It is raised when you read past the end of the file with UTL_FILE.GET_LINE.
  - **VALUE_ERROR**
    It is raised when you try to read or write lines in the file which are too long.
  - **INVALID_MAXLINESIZE**
    Oracle 8.0 and above: It is raised when you try to open a file with a maximum linesize outside of the valid range (between 1 through 32767).

You can use the FOPEN and IS_OPEN functions when you open files via UTL_FILE.

Note:

> Using the UTL-FILE package, you can only open a maximum of ten files for each Oracle session.

UTL_FILE provides only one program to retrieve data from a file, namely the GET_LINE procedure.

UTL_FILE.FOPEN function:

> The FOPEN function opens the specified file and returns a file handle that you can then use to manipulate the file.
>
> The header for the function is:

```
FUNCTION UTL_FILE.FOPEN (FUNCTION UTL_FILE.FOPEN
(location IN VARCHAR2, location IN VARCHAR2, filename IN
VARCHAR2, filename IN VARCHAR2, open_mode IN VARCHAR2)
open_mode IN VARCHAR2, RETURN file_type; max_linesize IN
BINARY_INTEGER) RETURN file_type;
```

UTL_FILE.FOPEN Function:

- Parameters for the function shown in the slide are summarized in the following table.

Parameter : location

Description : Location of the file

Parameter : filename

Description : Name of the file

Parameter : openmode

Description : Mode in which the file has to be opened

Parameter : max_linesize

Description : The maximum number of characters per line, including the newline character, for this file. Minimum is 1, maximum is 32767.

contd.

UTL_FILE.FOPEN Function (contd.):

- You can open the file in one of the following three modes:
  - **R mode**
    Open the file read-only. If you use this mode, use UTL_FILE's GET_LINE procedure to read from the file.
  - **W mode**
    Open the file to read and write in replace mode. When you open in replace mode, all existing lines in the file are removed. If you use this mode, then you can use any of the following UTL_FILE programs to modify the file: PUT, PUT_LINE, NEW_LINE, PUTF, and FFLUSH.
  - **A mode**
    Open the file to read and write in append mode. When you open in append mode, all existing lines in the file are kept intact. New lines will be appended after the last line in the file. If you use this mode, then you can use any of the following UTL_FILE programs to modify the file: PUT, PUT_LINE, NEW_LINE, PUTF, and fFFLUSH.
- Example
  - The following example shows how to declare a file handle, and then open a configuration file for that handle in read-only mode:

```
DECLARE
        config_file UTL_FILE.FILE_TYPE;
BEGIN
        config_file := UTL_FILE.FOPEN ('/maint/admin',
'config.txt', 'R');
```

UTL_FILE.IS_OPEN function:

The IS_OPEN function returns TRUE if the specified handle points to a file that is already open. Otherwise, it returns FALSE.

The header for the function is:

where file is the file to be checked.

```
FUNCTION UTL_FILE.IS_OPEN (file IN
UTL_FILE.FILE_TYPE) RETURN BOOLEAN;
```

Reading from Files

UTL_FILE.GET_LINE procedure:

- The GET_LINE procedure reads a line of data from the specified file, if it is open, into the provided line buffer. Given below is the header for the procedure:

```
PROCEDURE UTL_FILE.GET_LINE
  (file IN UTL_FILE.FILE_TYPE,
   buffer OUT VARCHAR2);
```

- Parameters are summarized in the following table:

Parameter : File

Description : The file handle returned by a call to FOPEN.

Parameter : Buffer

Description : The buffer into which the line of data is read.

contd.

UTL_FILE.GET_LINE procedure (contd.):

- The variable specified for the buffer parameter must be large enough to hold all the data up to the next carriage return or end-of-file condition in the file. If not, PL/SQL will raise the VALUE_ERROR exception. The line terminator character is not included in the string passed into the buffer.
- For example:
  Since GET_LINE reads data only into a string variable, you will have to perform your own conversions to local variables of the appropriate datatype if your file holds numbers or dates. Of course, you can call this procedure and directly read data into string and numeric variables, as well. In this case, PL/SQL will be performing a runtime, implicit conversion for you. In many situations, this is fine.
- It is generally recommended that you avoid implicit conversions and instead perform your own conversion. This approach more clearly documents the steps and dependencies.
- Here is an example:

```
DECLARE
  fileID UTL_FILE.FILE_TYPE;
  strbuffer VARCHAR2(100);
  mynum NUMBER;
BEGIN
  fileID := UTL_FILE.FOPEN ('/tmp', 'numlist.txt', 'R');
  UTL_FILE.GET_LINE (fileID, strbuffer);
  mynum := TO_NUMBER (strbuffer);
END;
/
```

- When GET_LINE attempts to read past the end of the file, the NO_DATA_FOUND exception is raised. This is the same exception that is raised when you:
  - execute an implicit (SELECT INTO) cursor that returns no rows, or
  - reference an undefined row of a PL/SQL (nested in PL/SQL8) table.
- If you are performing more than one of these operations in the same PL/SQL block, remember that this same exception can be caused by very different parts of your program.

Writing to Files: UTL_FILE.PUT procedure

- The PUT procedure puts data out to the specified open file.
- Given below is the header for this procedure:

```
PROCEDURE UTL_FILE.PUT
   (file IN UTL_FILE.FILE_TYPE,
   buffer IN VARCHAR2);      ---------------OUT  replace with
IN
```

- Parameters are summarized in the following table.

Parameter	Description
File	The file handle returned by a call to FOPEN.
Buffer	The buffer containing the text to be written to the file; maximum size allowed is 32K for Oracle 8.0.3 and above; for earlier versions, it is 1023 bytes.

- The PUT procedure adds the data to the current line in the opened file, but does not append a line terminator. You must use the NEW_LINE procedure to terminate the current line or use PUT_LINE to write out a complete line with a line termination character.

Exceptions:

- PUT may raise any of the following exceptions:
  - UTL_FILE.INVALID_FILEHANDLE
  - UTL_FILE.INVALID_OPERATION
  - UTL_FILE.WRITE_ERROR

UTL_FILE offers a number of different procedures you can use to write to a file:

UTL_FILE.PUT procedure

Puts a piece of data (string, number, or date) into a file in the current line.

UTL_FILE.NEW_LINE procedure

Puts a newline or line termination character into the file at the current position.

UTL_FILE.PUT_LINE procedure

Puts a string into a file, followed by a platform-specific line termination character.

UTL_FILE.PUTF procedure

Puts up to five strings out to the file in a format based on a template string, similar to the printf function in C.

UTL_FILE.PUT_LINE procedure:

- This procedure writes data to a file, and then immediately appends a newline character after the text. Given below is the header for PUT_LINE:

Parameters are summarized in the following table.

```
PROCEDURE UTL_FILE.PUT_LINE
   (file IN UTL_FILE.FILE_TYPE,
   buffer IN VARCHAR2);
```

Parameter

Description

File

The file handle returned by a call to FOPEN.

Buffer

Text to be written to the file; maximum size allowed is 32K for Oracle 8.0. 3 and above; for earlier versions, it is 1023 bytes.

- Before you can call UTL_FILE.PUT_LINE, you must have already opened the file.

contd.

UTL_FILE.FCLOSE procedure:

- Use FCLOSE to close an open file. The header for this procedure is:

  where file is the file handle.

- Note that the argument to UTL_FILE.FCLOSE is an IN OUT parameter, because the procedure sets the id field of the record to NULL after the file is closed.
- If there is buffered data that has not yet been written to the file when you try to close it, UTL_FILE will raise the WRITE_ERROR exception.

```
PROCEDURE UTL_FILE.FCLOSE (file IN OUT
FILE_TYPE);
```

UTL_FILE.FCLOSE_ALL procedure:

- FCLOSE_ALL closes all the opened files. Given below is the header for this procedure:

```
PROCEDURE UTL_FILE.FCLOSE_ALL;
```

- This procedure will come in handy when you have opened a variety of files and want to make sure that none of them are left open when your program terminates.
- In programs in which files have been opened, you should also call FCLOSE_ALL in exception handlers in programs. If there is an abnormal termination of the program, files will then still be closed.

```
EXCEPTION
  WHEN OTHERS

THEN
    UTL_FILE.FCLOSE_ALL;
    ... other clean up activities ...
END;
```

- NOTE: When you close your files with the FCLOSE_ALL procedure, none of your file handles will be marked as closed (the id field, in other words, will still be non-NULL). The result is that any calls to IS_OPEN for those file handles will still return TRUE. You will not, however, be able to perform any read or write operations on those files (unless you reopen them).

Exceptions

- FCLOSE_ALL may raise the exception UTL_FILE.WRITE_ERROR.

Handling LOB (Large Objects):

- Large Objects (LOBs) are a set of datatypes that are designed to hold large amounts of data.

Two types of LOBs are supported:

Those stored in the database either in-line in the table or in a separate segment or tablespace, such as BLOB, CLOB, and NCLOB. LOBs in the database are stored inside database tablespaces in a way that optimizes space and provides efficient access. The following SQL datatypes are supported for declaring internal LOBs: BLOB, CLOB, and NCLOB

Those stored as operating system files, such as BFILEs

- LOBs are designed to support Unstructured kind of data.

Unstructured Data:

- Unstructured Data cannot be decomposed into Standard Components:
  - Unstructured data cannot be decomposed into standard components. Data about an Employee can be “structured” into a Name (probably a character string), an identification (likely a number), a Salary, and so on. But if you are given a Photo, you find that the data really consists of a long stream of 0s and 1s. These 0s and 1s are used to switch pixels on or off so that you will see the Photo on a display. However, they cannot be broken down into any finer structure in terms of database storage.
- Unstructured Data is Large:
  - Also interesting is the fact that unstructured data such as text, graphic images, still video clips, full motion video, and sound waveforms tend to be large - a typical employee record may be a few hundred bytes, but even small amounts of multimedia data can be thousands of times larger.
- Unstructured Data in System Files needs Accessing from the Database:
  - Finally, some multimedia data may reside on operating system files, and it is desirable to access them from the database.

contd.

- Given below is a summary of all the four types of LOBs :
  - BLOB (Binary LOB)
    BLOB stores unstructured binary data up to 4GB in length.
    For example: Video or picture information
  - CLOB (Character LOB)
    CLOB stores single-byte character data up to 4GB in length.
    For example: Store document
  - NCLOB (National CLOB)
    NCLOB stores a CLOB column that supports multi-byte characters from National Character set defined by Oracle 8 database.
  - BFILE (Binary File)
    BFILE stores read-only binary data as an external file outside the database.
    Internal objects store a locator in the Large Object column of a table.
    Locator is a pointer that specifies the actual location of LOB stored out-of-line.
    The LOB locator for BFILE is the pointer to the location of the binary file stored in the operating system.
    Oracle supports data integrity and concurrency for all the LOBs except for BFILEs.

Handling LOB (Large Objects) (contd.):
Unstructured Data (contd.):

- LOB Datatype helps support Internet Applications:
  - With the growth of the internet and content-rich applications, it has become imperative that the database supports a datatype that fulfills the following:
    - datatype should store unstructured data
    - datatype should be optimized for large amounts of such data
    - datatype should provide an uniform way of accessing large unstructured data within the database or outside

<u>BFILE considerations</u>:

- There are some special considerations you should be aware of when you work with BFILEs.
  - **The DIRECTORY object**
    - A BFILE locator consists of a directory alias and a filename. The directory alias is an Oracle8 database object that allows references to operating system directories without hard-coding directory pathnames. This statement creates a directory:
      - → CREATE DIRECTORY IMAGES AS 'c:\images';
    - To refer to the c:\images directory within SQL, you can use the IMAGES alias, rather than hard-coding the actual directory pathname.
    - To create a directory, you need the CREATE DIRECTORY or CREATE ANY DIRECTORY privilege. To reference a directory, you must be granted the READ privilege, as in:
      - → GRANT READ ON DIRECTORY IMAGES TO SCOTT;
  - **Populating a BFILE locator**
    - The Oracle8 built-in function BFILENAME can be used to populate a BFILE locator. BFILENAME is passed a directory alias and filename and returns a locator to the file. In the following block, the BFILE variable corporate_logo is assigned a locator for the file named ourlogo.bmp located in the IMAGES directory:

- Once a BFILE column or variable is associated with a physical file, read operations on the BFILE can be performed using the DBMS_LOB package.
- Remember that access to physical files via BFILEs is read-only, and that the BFILE value is a pointer. The contents of the file remain outside of the database, but on the same server on which the database resides.

```
DECLARE
   corporate_logo     BFILE;
BEGIN
   corporate_logo := BFILENAME ( 'IMAGES',
'ourlogo.bmp' );
END;
```

The following statements populate the my_book_files table; each row is associated with a file in the BOOK_TEXT directory:

```
INSERT INTO my_book_files ( file_descr, book_file )
     VALUES ( 'Chapter 1', BFILENAME('BOOK_TEXT',
'chapter01.txt') );
UPDATE  my_book_files
   SET  book_file = BFILENAME( 'BOOK_TEXT',
'chapter02rev.txt' )
 WHERE  file_descr = 'Chapter 2';
```

Internal LOB considerations: Few more points:

Given below are a few more special considerations for Internal LOBs.

- **Retaining the LOB locator**
  - The following statement populates the my_book_text table, which contains CLOB column chapter_text:
  - Programs within the DBMS_LOB package require a LOB locator to be passed as input. If you want to insert the preceding row and then call a DBMS_LOB program using the row's CLOB value, you must retain the LOB locator created by your INSERT statement.
  - You can do this as shown in the following block, which inserts a row, selects the inserted LOB locator, and then calls the DBMS_LOB.GETLENGTH program to get the size of the CLOB chapter_text column. Note that the GETLENGTH program expects a LOB locator.

```
INSERT INTO my_book_text ( chapter_descr, chapter_text )
VALUES ( 'Chapter 1', 'It was a dark and stormy night.' );
```

```
DECLARE
  chapter_loc          CLOB;
  chapter_length       INTEGER;
BEGIN
  INSERT INTO my_book_text ( chapter_descr,
chapter_text )
       VALUES ( 'Chapter 1', 'It was a dark and stormy
night.' );
  SELECT  chapter_text
    INTO  chapter_loc
    FROM  my_book_text
   WHERE  chapter_descr = 'Chapter 1';
  chapter_length := DBMS_LOB.GETLENGTH(
chapter_loc );
  DBMS_OUTPUT.PUT_LINE( 'Length of Chapter 1: ' ||
chapter_length );
END;
/
```

This is the output of the script:

```
Length of Chapter 1: 31
```

contd.

Internal LOB considerations: Few more points (contd.):

- **The RETURNING clause**
  - You can avoid the second trip to the database (i.e. the SELECT of the LOB locator after the INSERT) by using a RETURNING clause in the INSERT statement.
  - By using this feature, perform the INSERT operation and the LOB locator value for the new row in a single operation.

```
DECLARE
  chapter_loc       CLOB;
  chapter_length     INTEGER;
BEGIN

  INSERT INTO my_book_text ( chapter_descr, chapter_text
)
      VALUES ( 'Chapter 1', 'It was a dark and stormy night.' )
    RETURNING chapter_text INTO chapter_loc;

  chapter_length := DBMS_LOB.GETLENGTH( chapter_loc
);

  DBMS_OUTPUT.PUT_LINE( 'Length of Chapter 1: ' ||
chapter_length );

END;
/
```

This is the output of the script:

```
Length of Chapter 1: 31
```

- The RETURNING clause can be used in both INSERT and UPDATE statements.

contd.

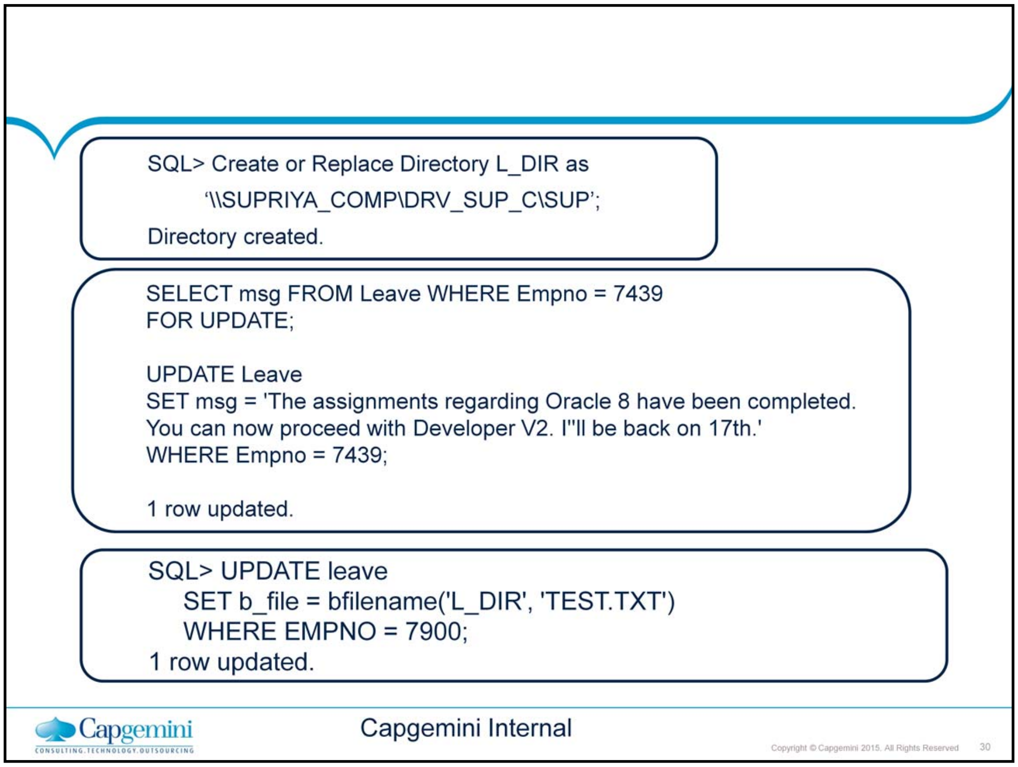
<u>Internal LOB considerations: Few more points (contd.):</u>

- **NULL versus “empty” LOB locators**
  - Oracle8 provides the built-in functions EMPTY_BLOB and EMPTY_CLOB to set BLOB, CLOB, and NCLOB columns to “empty”.
  - For example:
  - The LOB data is set to NULL. However, the associated LOB locator is assigned a valid locator value, which points to the NULL data. This LOB locator can then be passed to DBMS_LOB programs.

```
INSERT INTO my_book_text ( chapter_descr, chapter_text )
   VALUES ( 'Table of Contents', EMPTY_CLOB() );
```

This is the output of the script:
Length of Table of Contents: 0

- Note that EMPTY_CLOB can be used to populate both CLOB and NCLOB columns. EMPTY_BLOB and EMPTY_CLOB can be called with or without empty parentheses.
- Note: Do not populate BLOB, CLOB, or NCLOB columns with NULL values. Instead, use the EMPTY_BLOB or EMPTY_CLOB functions, which will populate the columns with a valid LOB locator and set the associated data to NULL.



```
SQL> Create or Replace Directory L_DIR as
        '\\SUPRIYA_COMP\DRV_SUP_C\SUP';
Directory created.
```

```
SELECT msg FROM Leave WHERE Empno = 7439
FOR UPDATE;

UPDATE Leave
SET msg = 'The assignments regarding Oracle 8 have been completed.
You can now proceed with Developer V2. I''ll be back on 17th.'
WHERE Empno = 7439;

1 row updated.
```

```
SQL> UPDATE leave
    SET b_file = bfilename('L_DIR', 'TEST.TXT')
    WHERE EMPNO = 7900;
1 row updated.
```

Capgemini Internal

Copyright © Capgemini 2015. All Rights Reserved 30

Accessing External LOBs - Examples**Example 1:** Create a directory object as shown below:

Example 2: In the following statement, we associate the file TEST.TXT for Empno 7900.

Read operations on the BFILE can be performed by using PL/SQL DBMS_LOB package and OCI.

These files are read-only through BFILES.

These files cannot be updated or deleted through BFILES

While updating LOBs:

Explicitly lock the rows.

Use the FOR UPDATE clause in a SELECT statement.

- To update a column that uses the BFILE datatype, you do not have to lock the rows.
- Using a DBMS_LOB Package:
  - Provides procedures to access LOBs.
  - Allows reading and modifying BLOBs, CLOBs, and NCLOBs, and provides read-only operations on BFILEs.
- All DBMS_LOB routines work based on LOB locators.

DBMS_LOB Package Routines:

The routines that can modify **BLOB**, **CLOB**, and **NCLOB** values are:

- APPEND() - appends the contents of the source LOB to the destination LOB
- COPY() - copies all or part of the source LOB to the destination LOB
- ERASE() - erases all or part of a LOB
- LOADFROMFILE() - loads BFILE data into an internal LOB
- TRIM() - trims the LOB value to the specified shorter length
- WRITE() - writes data to the LOB from a specified offset

The routines that read or examine **LOB** values are:

- GETLENGTH() - gets the length of the LOB value
- INSTR() - returns the matching position of the nth occurrence of the pattern in the LOB
- READ() - reads data from the LOB starting at the specified offset
- SUBSTR() - returns part of the LOB value starting at the specified offset

The read-only routines specific to **BFILEs** are:

- FILECLOSE() - closes the file
- FILECLOSEALL()- closes all previously opened files
- FILEEXISTS() - checks if the file exists on the server
- FILEGETNAME() - gets the directory alias and file name
- FILEISOPEN() - checks if the file was opened using the input BFILE
- locators
- FILEOPEN() - opens a file

DBMS_LOB Exceptions:
A DBMS_LOB function or procedure can raise any of the named exceptions shown in the table.

Exception

Code in error msg

Meaning

INVALID_ARGVAL

21560

“argument %s is null, invalid, or out of range”

ACCESS_ERROR

22925

Attempt to read/write beyond maximum LOB size on <n>.

NO_DATA_FOUND

1403

EndofLOB indicator for looping read operations.

VALUE_ERROR

6502

Invalid value in parameter.