5

Using Oracle-Supplied Packages in Application Development

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

- **Describe how the `DBMS_OUTPUT` package works**
- **Use `UTL_FILE` to direct output to operating system files**
- **Use the `HTP` package to generate a simple Web page**
- **Describe the main features of `UTL_MAIL`**
- **Call the `DBMS_SCHEDULER` package to schedule PL/SQL code for execution**

ORACLE

Lesson Aim

In this lesson, you learn how to use some of the Oracle-supplied packages and their capabilities. This lesson focuses on the packages that generate text-based and Web-based output, e-mail processing, and the provided scheduling capabilities.

Using Oracle-Supplied Packages

The Oracle-supplied packages:

- **Are provided with the Oracle server**
- **Extend the functionality of the database**
- **Enable access to certain SQL features that are normally restricted for PL/SQL**

For example, the `DBMS_OUTPUT` package was originally designed to debug PL/SQL programs.

ORACLE

Using Oracle-Supplied Packages

Packages are provided with the Oracle server to allow either of the following:

- PL/SQL access to certain SQL features
- The extension of the functionality of the database

You can use the functionality provided by these packages when creating your application, or you may simply want to use these packages as ideas when you create your own stored procedures.

Most of the standard packages are created by running `catproc.sql`. The `DBMS_OUTPUT` package is the one that you will be most familiar with during this course. You should know about this package if you attended the course *Oracle Database 10g: PL/SQL Fundamentals*.

List of Some Oracle-Supplied Packages

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

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

ORACLE

List of Some Oracle-Supplied Packages

The list of PL/SQL packages provided with an Oracle database grows with the release of new versions. It would be impossible to cover the exhaustive set of packages and their functionality in this course. For more information, refer to the *PL/SQL Packages and Types Reference 10g* (previously known as the *PL/SQL Supplied Packages Reference)*. This lesson covers the last five packages in this list.

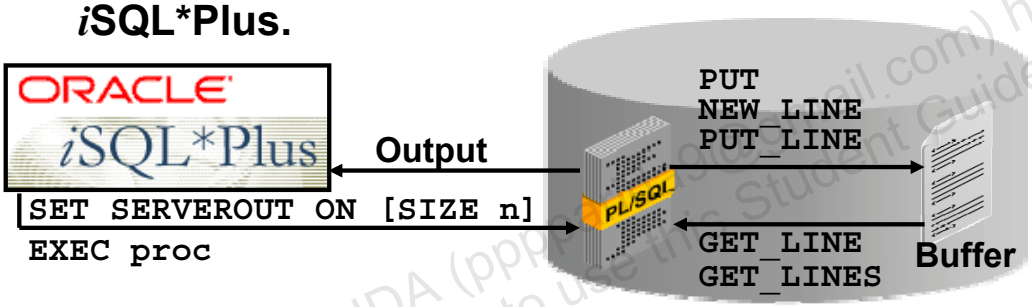
The following is a brief description about all the listed packages:

- `DBMS_ALERT` supports asynchronous notification of database events. Messages or alerts are sent on a `COMMIT` command.
- `DBMS_LOCK` is used to request, convert, and release locks through Oracle Lock Management services.
- `DBMS_SESSION` enables programmatic use of the `ALTER SESSION` SQL statement and other session-level commands.
- `DBMS_OUTPUT` provides debugging and buffering of text data.
- `HTP` package writes HTML-tagged data into database buffers.
- `UTL_FILE` enables reading and writing of operating system text files.
- `UTL_MAIL` enables composing and sending of e-mail messages.
- `DBMS_SCHEDULER` enables scheduling and automated execution of PL/SQL blocks, stored procedures, and external procedures and executables.

How the DBMS_OUTPUT Package Works

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

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



How the DBMS_OUTPUT Package Works

The DBMS_OUTPUT package sends textual messages from any PL/SQL block into a buffer in the database. Procedures provided by the package include the following:

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

The buffer size can be set by using:

- The SIZE n option appended to the SET SERVEROUTPUT ON command, where n is between 2,000 (the default) and 1,000,000 (1 million characters)
- An integer parameter between 2,000 and 1,000,000 in the ENABLE procedure

Practical Uses

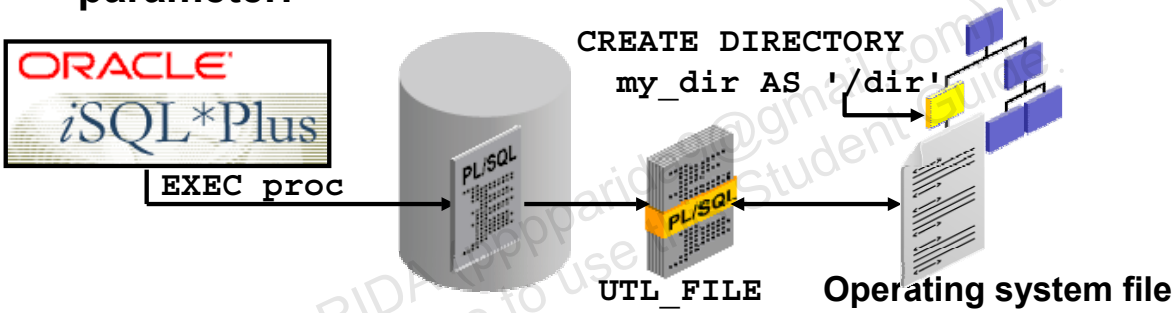
- You can output results to the window for debugging purposes.
- You can trace code execution path for a function or procedure.
- You can send messages between subprograms and triggers.

Note: There is no mechanism to flush output during the execution of a procedure.

Interacting with Operating System Files

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

- **Provides a restricted version of operating system stream file I/O for text files**
- **Can access files in operating system directories defined by a `CREATE DIRECTORY` statement. You can also use the `utl_file_dir` database parameter.**



Interacting with Operating System Files

The Oracle-supplied `UTL_FILE` package is used to access text files in the operating system of the database server. The database provides read and write access to specific operating system directories by using:

- A `CREATE DIRECTORY` statement that associates an alias with an operating system directory. The database directory alias can be granted the `READ` and `WRITE` privileges to control the type of access to files in the operating system. For example:

```
CREATE DIRECTORY my_dir AS '/temp/my_files';
GRANT READ, WRITE ON my_dir TO public.
```

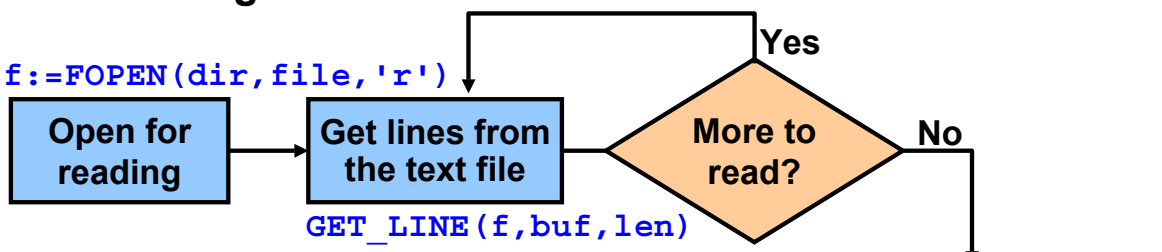
- The paths specified in the `utl_file_dir` database initialization parameter

The preferred approach is to use the directory alias created by the `CREATE DIRECTORY` statement, which does not require the database to be restarted. The operating system directories specified by using either of these techniques should be accessible to and on the same machine as the database server processes. The path (directory) names may be case sensitive for some operating systems.

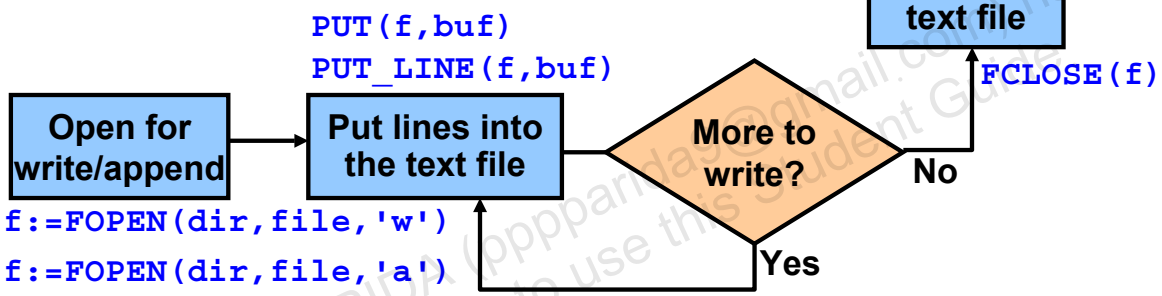
Note: The `DBMS_LOB` package can be used to read binary files on the operating system. `DBMS_LOB` is covered in the lesson titled "Manipulating Large Objects."

File Processing Using the UTL_FILE Package

- **Reading a file:**



- **Writing or appending to a file:**



File Processing Using the UTL_FILE Package

Using the procedures and functions in the UTL_FILE package, open files with the FOPEN function. You then either read from or write or append to the file until processing is done. After completing processing the file, close the file by using the FCLOSE procedure. The following are the subprograms:

- The FOPEN function opens a file in a specified directory for input/output (I/O) and returns a file handle used in subsequent I/O operations.
- The IS_OPEN function returns a Boolean value whenever a file handle refers to an open file. Use IS_OPEN to check whether the file is already open before opening the file.
- The GET_LINE procedure reads a line of text from the file into an output buffer parameter. (The maximum input record size is 1,023 bytes unless you specify a larger size in the overloaded version of FOPEN.)
- The PUT and PUT_LINE procedures write text to the opened file.
- The PUTF procedure provides formatted output with two format specifiers: %s to substitute a value into the output string and \n for a new line character.
- The NEW_LINE procedure terminates a line in an output file.
- The FFLUSH procedure writes all data buffered in memory to a file.
- The FCLOSE procedure closes an opened file.
- The FCLOSE_ALL procedure closes all opened file handles for the session.

Exceptions in the UTL_FILE Package

You may have to handle one of these exceptions when using UTL_FILE subprograms:

- INVALID_PATH
- INVALID_MODE
- INVALID_FILEHANDLE
- INVALID_OPERATION
- READ_ERROR
- WRITE_ERROR
- INTERNAL_ERROR

Other exceptions not in the UTL_FILE package are:

- NO_DATA_FOUND and VALUE_ERROR

ORACLE

Exceptions in the UTL_FILE Package

The UTL_FILE package declares seven exceptions that indicate an error condition in the operating system file processing. The UTL_FILE exceptions are:

- INVALID_PATH if the file location or file name was invalid
- INVALID_MODE if the OPEN_MODE parameter in FOPEN was invalid
- INVALID_FILEHANDLE if the file handle was invalid
- INVALID_OPERATION if the file could not be opened or operated on as requested
- READ_ERROR if an operating system error occurred during the read operation
- WRITE_ERROR if an operating system error occurred during the write operation
- INTERNAL_ERROR if an unspecified error occurred in PL/SQL

Note: These exceptions must always be prefaced with the package name. UTL_FILE procedures can also raise predefined PL/SQL exceptions such as NO_DATA_FOUND or VALUE_ERROR.

The NO_DATA_FOUND exception is raised when reading past the end of a file by using UTL_FILE.GET_LINE or UTL_FILE.GET_LINES.

FOPEN and IS_OPEN Function Parameters

```
FUNCTION FOPEN (location IN VARCHAR2,
                filename IN VARCHAR2,
                open_mode IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

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

Example:

```
CREATE PROCEDURE read_file(dir VARCHAR2, filename
VARCHAR2) IS file UTL_FILE.FILE_TYPE;
...
BEGIN
  ...
  IF NOT UTL_FILE.IS_OPEN(file) THEN
   file := UTL_FILE.FOPEN (dir, filename, 'R');
   ...
  END IF;
END read_file;
```

ORACLE

FOPEN and IS_OPEN Function Parameters

The parameters include the following:

- `location` parameter: Specifies the name of a directory alias defined by a `CREATE DIRECTORY` statement, or an operating system–specific path specified by using the `utl_file_dir` database parameter
- `filename` parameter: Specifies the name of the file, including the extension, without any path information
- `open_mode` string: Specifies how the file is to be opened. Values are:
  - `'r'` for reading text (use `GET_LINE`)
  - `'w'` for writing text (`PUT`, `PUT_LINE`, `NEW_LINE`, `PUTF`, `FFLUSH`)
  - `'a'` for appending text (`PUT`, `PUT_LINE`, `NEW_LINE`, `PUTF`, `FFLUSH`)

The return value from `FOPEN` is a file handle whose type is `UTL_FILE.FILE_TYPE`. The handle must be used on subsequent calls to routines that operate on the opened file.

The `IS_OPEN` function parameter is the file handle. The `IS_OPEN` function tests a file handle to see whether it identifies an opened file. It returns a Boolean value of `TRUE` if the file has been opened; otherwise it returns a value of `FALSE` indicating that the file has not been opened. The slide example shows how to combine the use of the two subprograms.

Note: For the full syntax, refer to the *PL/SQL Packages and Types Reference*.

Using UTL_FILE: Example

```
CREATE OR REPLACE PROCEDURE sal_status(
  dir IN VARCHAR2, filename IN VARCHAR2) IS
  file UTL_FILE.FILE_TYPE;
 CURSOR empc IS
  SELECT last_name, salary, department_id
  FROM employees ORDER BY department_id;
 newdeptno employees.department_id%TYPE;
 olddeptno employees.department_id%TYPE := 0;
BEGIN
  file:= UTL_FILE.FOPEN (dir, filename, 'w');
  UTL_FILE.PUT_LINE(file,
    'REPORT: GENERATED ON ' || SYSDATE);
  UTL_FILE.NEW_LINE (file); ...
```

ORACLE

Using UTL_FILE: Example

In the example, the `sal_status` procedure creates a report of employees for each department, along with their salaries. The data is written to a text file by using the `UTL_FILE` package. In the code example, the variable `file` is declared as `UTL_FILE.FILE_TYPE`, a package type that is a record with a field called `ID` of the `BINARY_INTEGER` data type. For example:

```
TYPE file_type IS RECORD (id BINARY_INTEGER);
```

The field of `FILE_TYPE` record is private to the `UTL_FILE` package and should never be referenced or changed. The `sal_status` procedure accepts two parameters:

- The `dir` parameter for the name of the directory in which to write the text file
- The `filename` parameter to specify the name of the file

For example, to call the procedure, use the following:

```
EXECUTE sal_status('MY_DIR', 'salreport.txt')
```

Note: The directory location used (`MY_DIR`) must be in uppercase characters if it is a directory alias created by a `CREATE DIRECTORY` statement. When reading a file in a loop, the loop should exit when it detects the `NO_DATA_FOUND` exception. The `UTL_FILE` output is sent synchronously. `DBMS_OUTPUT` procedures do not produce output until the procedure is completed.

Using UTL_FILE: Example

```
  FOR emp_rec IN empc LOOP
    IF emp_rec.department_id <> olddeptno THEN
      UTL_FILE.PUT_LINE (file,
       'DEPARTMENT: ' || emp_rec.department_id);
      UTL_FILE.NEW_LINE (file);
    END IF;
    UTL_FILE.PUT_LINE (file,
       '  EMPLOYEE: ' || emp_rec.last_name ||
       ' earns: ' || emp_rec.salary);
    olddeptno := emp_rec.department_id;
    UTL_FILE.NEW_LINE (file);
  END LOOP;
  UTL_FILE.PUT_LINE(file,'*** END OF REPORT ***');
  UTL_FILE.FCLOSE (file);
EXCEPTION
 WHEN UTL_FILE.INVALID_FILEHANDLE THEN
  RAISE_APPLICATION_ERROR(-20001,'Invalid File.');
 WHEN UTL_FILE.WRITE_ERROR THEN
  RAISE_APPLICATION_ERROR (-20002, 'Unable to
write to file');
END sal_status;
/
```

ORACLE

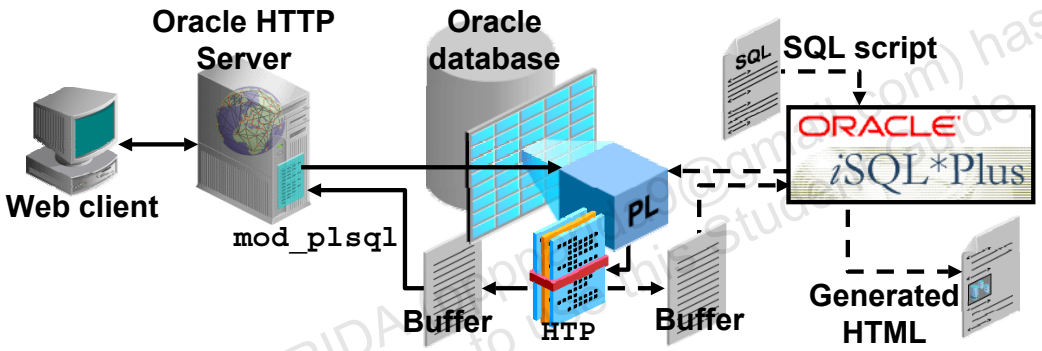
Using UTL_FILE: Example (continued)

The output for this report in the salreport.txt file is as follows:

```
SALARY REPORT: GENERATED ON 08-MAR-01
     DEPARTMENT: 10
          EMPLOYEE: Whalen earns: 4400
     DEPARTMENT: 20
          EMPLOYEE: Hartstein earns: 13000
          EMPLOYEE: Fay earns: 6000
     DEPARTMENT: 30
          EMPLOYEE: Raphaely earns: 11000
          EMPLOYEE: Khoo earns: 3100
          ...
     DEPARTMENT: 100
          EMPLOYEE: Greenberg earns: 12000
         ...
     DEPARTMENT: 110
          EMPLOYEE: Higgins earns: 12000
          EMPLOYEE: Gietz earns: 8300
          EMPLOYEE: Grant earns: 7000
     *** END OF REPORT ***
```

Generating Web Pages with the HTP Package

- The HTP package procedures generate HTML tags.
- The HTP package is used to generate HTML documents dynamically and can be invoked from:
  - A browser using Oracle HTTP Server and PL/SQL Gateway (mod_plsql) services
  - An *i*SQL*Plus script to display HTML output



Generating Web Pages with the HTP Package

The HTP package contains procedures that are used to generate HTML tags. The HTML tags that are generated typically enclose the data provided as parameters to the various procedures. The slide illustrates two ways in which the HTP package can be used:

- Most likely your procedures are invoked by the PL/SQL Gateway services, via the mod_plsql component supplied with Oracle HTTP Server, which is part of the Oracle Application Server product (represented by solid lines in the graphic).
- Alternatively (as represented by dotted lines in the graphic), your procedure can be called from *i*SQL*Plus that can display the generated HTML output, which can be copied and pasted to a file. This technique is used in this course because Oracle Application Server software is not installed as a part of the course environment.

Note: The HTP procedures output information to a session buffer held in the database server. In the Oracle HTTP Server context, when the procedure completes, the mod_plsql component automatically receives the buffer contents, which are then returned to the browser as the HTTP response. In SQL*Plus, you must manually execute:

- A SET SERVEROUTPUT ON command
- The procedure to generate the HTML into the buffer
- The OWA_UTIL.SHOWPAGE procedure to display the buffer contents

Using the HTP Package Procedures

- **Generate one or more HTML tags. For example:**

```
htp.bold('Hello');                -- <B>Hello</B>
htp.print('Hi <B>World</B>');     -- Hi <B>World</B>
```

- **Are used to create a well-formed HTML document:**

```
BEGIN                                  -- Generates:
 htp.htmlOpen;    --------->           <HTML>
 htp.headOpen;    --------->           <HEAD>
 htp.title('Welcome');  -->            <TITLE>Welcome</TITLE>
 htp.headClose;   --------->           </HEAD>
 htp.bodyOpen;    --------->           <BODY>
 htp.print('My home page');           My home page
 htp.bodyClose;   --------->           </BODY>
 htp.htmlClose;   --------->           </HTML>
END;
```


Using the HTP Package Procedures

The HTP package is structured to provide a one-to-one mapping of a procedure to standard HTML tags. For example, to display bold text on a Web page, the text must be enclosed in the HTML tag pair <B> and </B>. The first code box in the slide shows how to generate the word Hello in HTML bold text by using the equivalent HTP package procedure—that is, HTP.BOLD. The HTP.BOLD procedure accepts a text parameter and ensures that it is enclosed in the appropriate HTML tags in the HTML output that is generated.

The HTP.PRINT procedure copies its text parameter to the buffer. The example in the slide shows how the parameter supplied to the HTP.PRINT procedure can contain HTML tags. This technique is recommended only if you need to use HTML tags that cannot be generated by using the set of procedures provided in the HTP package.

The second example in the slide provides a PL/SQL block that generates the basic form of an HTML document. The example serves to illustrate how each of the procedures generates the corresponding HTML line in the enclosed text box on the right.

The benefit of using the HTP package is that you create well-formed HTML documents, eliminating the need to manually type the HTML tags around each piece of data.

Note: For information about all the HTP package procedures, refer to the *PL/SQL Packages and Types Reference*.

Creating an HTML File with *i*SQL*Plus

To create an HTML file with *i*SQL*Plus, perform the following steps:

1. **Create a SQL script with the following commands:**

```
SET SERVEROUTPUT ON
ACCEPT procname PROMPT "Procedure: "
EXECUTE &procname
EXECUTE owa_util.showpage
UNDEFINE proc
```

2. **Load and execute the script in *i*SQL*Plus, supplying values for substitution variables.**
3. **Select, copy, and paste the HTML text that is generated in the browser to an HTML file.**
4. **Open the HTML file in a browser.**

ORACLE

Creating an HTML File with *i*SQL*Plus

The slide example shows the steps for generating HTML by using any procedure and saving the output into an HTML file. You should perform the following steps:

1. Turn on server output with the `SET SERVEROUTPUT ON` command. Without this, you receive exception messages when running procedures that have calls to the `HTP` package.
2. Execute the procedure that contains calls to the `HTP` package.
   Note: This does *not* produce output, unless the procedure has calls to the `DBMS_OUTPUT` package.
3. Execute the `OWA_UTIL.SHOWPAGE` procedure to display the text. This call actually displays the HTML content that is generated from the buffer.

The `ACCEPT` command prompts for the name of the procedure to execute. The call to `OWA_UTIL.SHOWPAGE` displays the HTML tags in the browser window. You can then copy and paste the generated HTML tags from the browser window into an HTML file, typically with a `.htm` or `.html` extension.

Note: If you are using SQL*Plus, then you can use the `SPOOL` command to direct the HTML output directly to an HTML file. The `SPOOL` command is not supported in *i*SQL*Plus; therefore, the copy-and-paste technique is the only option.

Using UTL_MAIL

The UTL_MAIL package:

- **Is a utility for managing e-mail that includes such commonly used e-mail features as attachments, CC, BCC, and return receipt**
- **Requires the SMTP_OUT_SERVER database initialization parameter to be set**
- **Provides the following procedures:**
  - **SEND for messages without attachments**
  - **SEND_ATTACH_RAW for messages with binary attachments**
  - **SEND_ATTACH_VARCHAR2 for messages with text attachments**

Using UTL_MAIL

The UTL_MAIL package is a utility for managing e-mail that includes commonly used e-mail features such as attachments, CC, BCC, and return receipt.

The UTL_MAIL package is not installed by default because of the SMTP_OUT_SERVER configuration requirement and the security exposure this involves. When installing UTL_MAIL, you should take steps to prevent the port defined by SMTP_OUT_SERVER being swamped by data transmissions. To install UTL_MAIL, log in as a DBA user in *i*SQL*Plus and execute the following scripts:

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

You should define the SMTP_OUT_SERVER parameter in the init.ora file database initialization file:

```
SMTP_OUT_SERVER=mystmpserver.mydomain.com
```

The SMTP_OUT_SERVER parameter specifies the SMTP host and port to which UTL_MAIL delivers outbound e-mail. Multiple servers can be specified, separated by commas. If the first server in the list is unavailable, then UTL_MAIL tries the second server, and so on. If SMTP_OUT_SERVER is not defined, then this invokes a default setting derived from DB_DOMAIN, which is a database initialization parameter specifying the logical location of the database within the network structure. For example:

```
db_domain=mydomain.com
```

Installing and Using UTL_MAIL

- As SYSDBA, using *i*SQL*Plus:
  - Set the SMTP_OUT_SERVER (requires DBMS restart).

```
ALTER SYSTEM SET SMTP_OUT_SERVER='smtp.server.com'
SCOPE=SPFILE
```

  - Install the UTL_MAIL package.

```
@?/rdbms/admin/utlmail.sql
@?/rdbms/admin/prvtmail.plb
```

- As a developer, invoke a UTL_MAIL procedure:

```
BEGIN
 UTL_MAIL.SEND('otn@oracle.com','user@oracle.com',
   message => 'For latest downloads visit OTN',
   subject => 'OTN Newsletter');
END;
```

ORACLE

Installing and Using UTL_MAIL

The slide shows how to configure the SMTP_OUT_SERVER parameter to the name of the SMTP host in your network, and how to install the UTL_MAIL package that is not installed by default. Changing the SMTP_OUT_SERVER parameter requires restarting the database instance. These tasks are performed by a user with SYSDBA capabilities.

The last example in the slide shows the simplest way to send a text message by using the UTL_MAIL.SEND procedure with at least a subject and a message. The first two required parameters are the following :

- The sender e-mail address (in this case, otn@oracle.com)
- The recipients e-mail address (for example, user@oracle.com). The value can be a comma-separated list of addresses.

The UTL_MAIL.SEND procedure provides several other parameters, such as cc, bcc, and priority with default values, if not specified. In the example, the message parameter specifies the text for the e-mail, and the subject parameter contains the text for the subject line. To send an HTML message with HTML tags, add the mime_type parameter (for example, mime_type=>'text/html').

Note: For details about all the UTL_MAIL procedure parameters, refer to the *PL/SQL Packages and Types Reference*.

Sending E-Mail with a Binary Attachment

Use the `UTL_MAIL.SEND_ATTACH_RAW` procedure:

```
CREATE OR REPLACE PROCEDURE send_mail_logo IS
BEGIN
  UTL_MAIL.SEND_ATTACH_RAW(
    sender => 'me@oracle.com',
    recipients => 'you@somewhere.net',
    message =>
      '<HTML><BODY>See attachment</BODY></HTML>',
    subject => 'Oracle Logo',
    mime_type => 'text/html'
    attachment => get_image('oracle.gif'),
    att_inline => true,
    att_mime_type => 'image/gif',
    att_filename => 'oralogo.gif');
END;
/
```

ORACLE

Sending E-Mail with a Binary Attachment

The slide shows a procedure calling the `UTL_MAIL.SEND_ATTACH_RAW` procedure to send a textual or an HTML message with a binary attachment. In addition to the `sender`, `recipients`, `message`, `subject`, and `mime_type` parameters that provide values for the main part of the e-mail message, the `SEND_ATTACH_RAW` procedure has the following highlighted parameters:

- The `attachment` parameter (required) accepts a `RAW` data type, with a maximum size of 32,767 binary characters.
- The `att_inline` parameter (optional) is Boolean (default `TRUE`) to indicate that the attachment is viewable with the message body.
- The `att_mime_type` parameter (optional) specifies the format of the attachment. If not provided, it is set to `application/octet`.
- The `att_filename` parameter (optional) assigns any file name to the attachment. It is `NULL` by default, in which case, the name is assigned a default name.

The `get_image` function in the example uses a `BFILE` to read the image data. Using a `BFILE` requires creating a logical directory name in the database by using the `CREATE DIRECTORY` statement. The details about working with a `BFILE` are covered in the lesson titled “Manipulating Large Objects.” The code for `get_image` is shown on the following page.

Sending E-Mail with a Binary Attachment (continued)

The `get_image` function uses the `DBMS_LOB` package to read a binary file from the operating system:

```
CREATE OR REPLACE FUNCTION get_image(
     filename VARCHAR2, dir VARCHAR2 := 'TEMP')
RETURN RAW IS
  image RAW(32767);
  file  BFILE := BFILENAME(dir, filename);
BEGIN
  DBMS_LOB.FILEOPEN(file, DBMS_LOB.FILE_READONLY);
  image := DBMS_LOB.SUBSTR(file);
  DBMS_LOB.CLOSE(file);
  RETURN image;
END;
/
```

To create the directory called `TEMP`, execute the following statement in *i*SQL*Plus:

```
CREATE DIRECTORY temp AS 'e:\temp';
```

Note: You need the `CREATE ANY DIRECTORY` system privilege to execute this statement.

Sending E-Mail with a Text Attachment

Use the `UTL_MAIL.SEND_ATTACH_VARCHAR2` procedure:

```
CREATE OR REPLACE PROCEDURE send_mail_file IS
BEGIN
  UTL_MAIL.SEND_ATTACH_VARCHAR2(
    sender => 'me@oracle.com',
    recipients => 'you@somewhere.net',
    message =>
      '<HTML><BODY>See attachment</BODY></HTML>',
    subject => 'Oracle Notes',
    mime_type => 'text/html'
    attachment => get_file('notes.txt'),
    att_inline => false,
    att_mime_type => 'text/plain',
    att_filename => 'notes.txt');
END;
/
```

ORACLE

Sending E-Mail with a Text Attachment

The slide shows a procedure that calls the `UTL_MAIL.SEND_ATTACH_VARCHAR2` procedure to send a textual or an HTML message with a text attachment. In addition to the `sender`, `recipients`, `message`, `subject`, and `mime_type` parameters that provide values for the main part of the e-mail message, the `SEND_ATTACH_VARCHAR2` procedure has the following parameters highlighted:

- The `attachment` parameter (required) accepts a `VARCHAR2` data type with a maximum size of 32,767 binary characters.
- The `att_inline` parameter (optional) is a Boolean (default `TRUE`) to indicate that the attachment is viewable with the message body.
- The `att_mime_type` parameter (optional) specifies the format of the attachment. If not provided, it is set to `application/octet`.
- The `att_filename` parameter (optional) assigns any file name to the attachment. It is `NULL` by default, in which case, the name is assigned a default name.

The `get_file` function in the example uses a `BFILE` to read a text file from the operating system directories for the value of the `attachment` parameter, which could simply be populated from a `VARCHAR2` variable. The code for `get_file` is shown on the following page.

Sending E-Mail with a Text Attachment (continued)

The `get_file` function uses the `DBMS_LOB` package to read a binary file from the operating system, and uses the `UTL_RAW` package to convert the `RAW` binary data into readable text data in the form of a `VARCHAR2` data type:

```
CREATE OR REPLACE FUNCTION get_file(
  filename VARCHAR2, dir VARCHAR2 := 'TEMP')
RETURN VARCHAR2 IS
  contents VARCHAR2(32767);
  file  BFILE := BFILENAME(dir, filename);
BEGIN
  DBMS_LOB.FILEOPEN(file, DBMS_LOB.FILE_READONLY);
  contents := UTL_RAW.CAST_TO_VARCHAR2(
                  DBMS_LOB.SUBSTR(file));
  DBMS_LOB.CLOSE(file);
  RETURN contents;
END;
/
```

Note: Alternatively, you could read the contents of the text file into a `VARCHAR2` variable by using the `UTL_FILE` package functionality.

The preceding example requires the `TEMP` directory to be created similar to the following statement in *i*SQL*Plus:

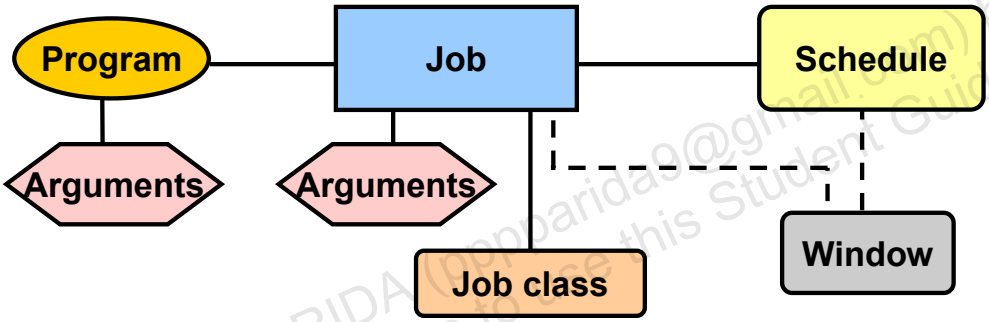
```
CREATE DIRECTORY temp AS 'e:\temp';
```

Note: The `CREATE ANY DIRECTORY` system privilege is required to execute this statement.

DBMS_SCHEDULER Package

The database Scheduler comprises several components to enable jobs to be run. Use the DBMS_SCHEDULER package to create each job with:

- **A unique job name**
- **A program ("what" should be executed)**
- **A schedule ("when" it should run)**



DBMS_SCHEDULER Package

Oracle Database 10*g* provides a collection of subprograms in the DBMS_SCHEDULER package to simplify management and to provide a rich set of functionality for complex scheduling tasks. Collectively, these subprograms are called the Scheduler and can be called from any PL/SQL program. The Scheduler enables database administrators and application developers to control when and where various tasks take place. By ensuring that many routine database tasks occur without manual intervention, you can lower operating costs, implement more reliable routines, and minimize human error.

The diagram shows the following architectural components of the Scheduler:

- A **job** is the combination of a program and a schedule. Arguments required by the program can be provided with the program or the job. All job names have the format [schema.]name. When you create a job, you specify the job name, a program, a schedule, and (optionally) job characteristics that can be provided through a **job class**.
- A **program** determines what should be run. Every automated job involves a particular executable, whether it is a PL/SQL block, a stored procedure, a native binary executable, or a shell script. A program provides metadata about a particular executable and may require a list of arguments.
- A **schedule** specifies when and how many times a job is executed.

`DBMS_SCHEDULER` Package (continued)

- A **job class** defines a category of jobs that share common resource usage requirements and other characteristics. At any given time, each job can belong to only a single job class. A job class has the following attributes:
  - A database **service** name. The jobs in the job class will have an affinity to the particular service specified—that is, the jobs will run on the instances that cater to the specified service.
  - A **resource consumer group**, which classifies a set of user sessions that have common resource-processing requirements. At any given time, a user session or job class can belong to a single resource consumer group. The resource consumer group that the job class associates with determines the resources that are allocated to the job class.
- A **window** is represented by an interval of time with a well-defined beginning and end, and is used to activate different resource plans at different times.

The slide focuses on the job component as the primary entity. However, a program, a schedule, a window, and a job class are components that can be created as individual entities that can be associated with a job to be executed by the Scheduler. When a job is created, it may contain all the information needed in-line—that is, in the call that creates the job. Alternatively, creating a job may reference a program or schedule component that was previously defined. Examples of this are discussed in the next few pages.

For more information about the Scheduler, see the Online Course titled *Oracle Database 10g: Configure and Manage Jobs with the Scheduler*.

Creating a Job

A job can be created in several ways by using a combination of in-line parameters, named `Programs`, and named `Schedules`. You can create a job with the `CREATE_JOB` procedure by:

- **Using in-line information with the "what" and the schedule specified as parameters**
- **Using a named (saved) program and specifying the schedule in-line**
- **Specifying what should be done in-line and using a named Schedule**
- **Using named Program and Schedule components**

Note: Creating a job requires the `CREATE JOB` system privilege.

ORACLE

Creating a Job

The component that causes something to be executed at a specified time is called a **job**. Use the `DBMS_SCHEDULER.CREATE_JOB` procedure of the `DBMS_SCHEDULER` package to create a job, which is in a disabled state by default. A job becomes active and scheduled when it is explicitly enabled. To create a job, you:

- Provide a name in the format `[schema.]name`
- Need the `CREATE JOB` privilege

Note: A user with the `CREATE ANY JOB` privilege can create a job in any schema except the `SYS` schema. Associating a job with a particular class requires the `EXECUTE` privilege for that class.

In simple terms, a job can be created by specifying all the job details—the program to be executed (what) and its schedule (when)—in the arguments of the `CREATE_JOB` procedure. Alternatively, you can use predefined Program and Schedule components. If you have a named Program and Schedule, then these can be specified or combined with in-line arguments for maximum flexibility in the way a job is created.

A simple logical check is performed on the schedule information (that is, checking the date parameters when a job is created). The database checks whether the end date is after the start date. If the start date refers to a time in the past, then the start date is changed to the current date.

Creating a Job with In-Line Parameters

Specify the type of code, code, start time, and frequency of the job to be run in the arguments of the `CREATE_JOB` procedure.

Here is an example that schedules a PL/SQL block every hour:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB(
   job_name => 'JOB_NAME',
   job_type => 'PLSQL_BLOCK',
   job_action => 'BEGIN ...; END;',
   start_date => SYSTIMESTAMP,
   repeat_interval=>'FREQUENCY=HOURLY;INTERVAL=1',
   enabled => TRUE);
END;
/
```


ORACLE

Creating a Job with In-Line Parameters

You can create a job to run a PL/SQL block, stored procedure, or external program by using the `DBMS_SCHEDULER.CREATE_JOB` procedure. The `CREATE_JOB` procedure can be used directly without requiring you to create Program or Schedule components.

The example in the slide shows how you can specify all the job details in-line. The parameters of the `CREATE_JOB` procedure define "what" is to be executed, the schedule, and other job attributes. The following parameters define what is to be executed:

- The `job_type` parameter can be one of three values:
  - `PLSQL_BLOCK` for any PL/SQL block or SQL statement. This type of job cannot accept arguments.
  - `STORED_PROCEDURE` for any stored stand-alone or packaged procedure. The procedures can accept arguments that are supplied with the job.
  - `EXECUTABLE` for an executable command-line operating system application
- The schedule is specified by using the following parameters:
  - The `start_date` accepts a time stamp, and the `repeat_interval` is string-specified as a calendar or PL/SQL expression. An `end_date` can be specified.

Note: String expressions that are specified for `repeat_interval` are discussed later. The example specifies that the job should run every hour.

Creating a Job Using a Program

- Use `CREATE_PROGRAM` to create a program:

```
BEGIN
  DBMS_SCHEDULER.CREATE_PROGRAM(
   program_name => 'PROG_NAME',
   program_type => 'PLSQL_BLOCK',
   program_action => 'BEGIN ...; END;');
END;
```

- Use overloaded `CREATE_JOB` procedure with its `program_name` parameter:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   program_name => 'PROG_NAME',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'FREQ=DAILY',
   enabled => TRUE);
END;
```

ORACLE

Creating a Job Using a Program

The `DBMS_SCHEDULER.CREATE_PROGRAM` procedure defines a program that must be assigned a unique name. Creating the program separately for a job enables you to:

- Define the action once and then reuse this action within multiple jobs
- Change the schedule for a job without having to re-create the PL/SQL block
- Change the program executed without changing all the jobs

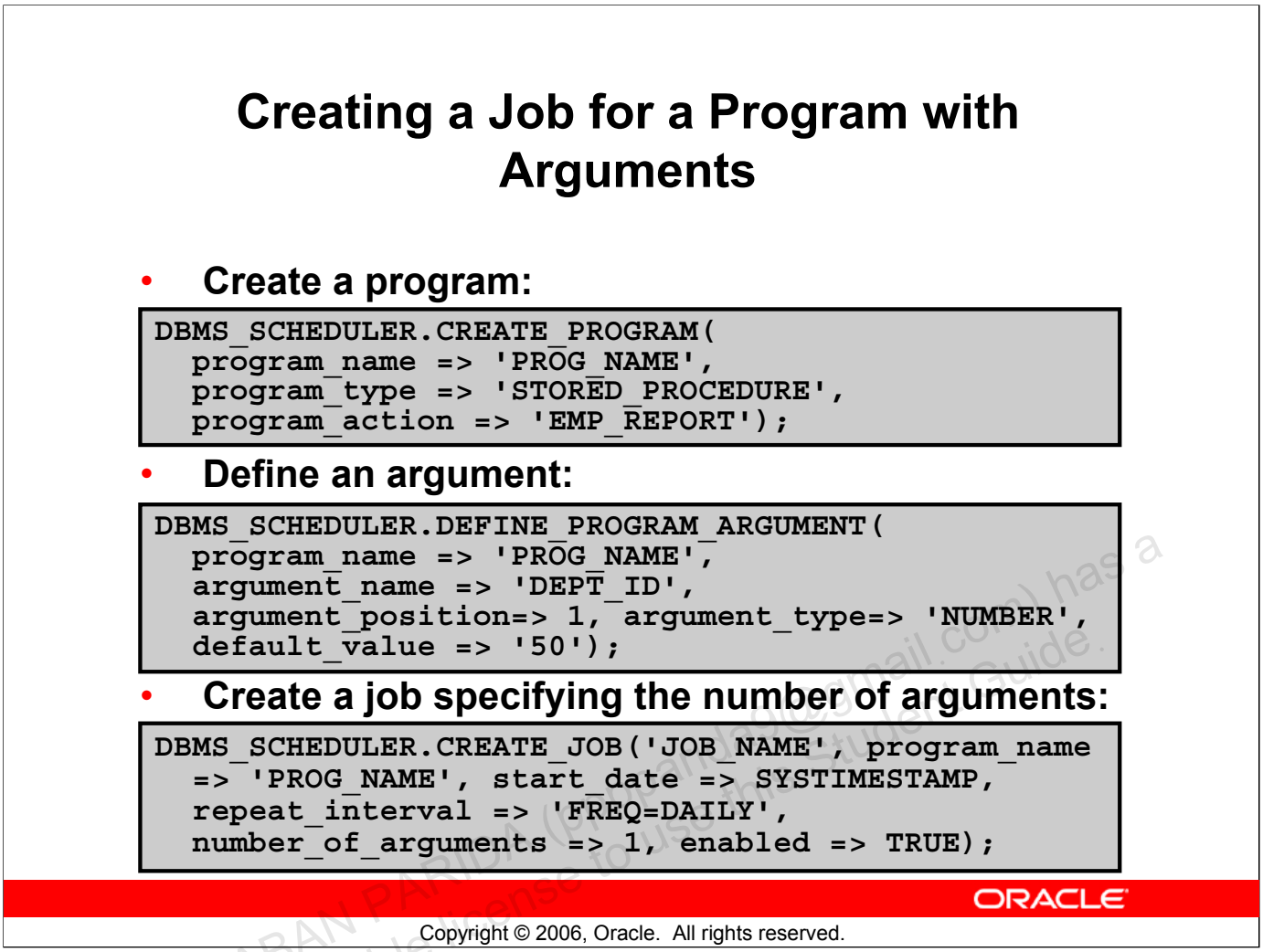
The program action string specifies a procedure, executable name, or PL/SQL block depending on the value of the `program_type` parameter, which can be:

- `PLSQL_BLOCK` to execute an anonymous block or SQL statement
- `STORED_PROCEDURE` to execute a stored procedure, such as PL/SQL, Java, or C
- `EXECUTABLE` to execute operating system command-line programs

The example shown in the slide demonstrates calling an anonymous PL/SQL block. You can also call an external procedure within a program, as in the following example:

```
DBMS_SCHEDULER.CREATE_PROGRAM(program_name =>
'GET_DATE',
    program_action => '/usr/local/bin/date',
    program_type => 'EXECUTABLE');
```

To create a job with a program, specify the program name in the `program_name` argument in the call to the `DBMS_SCHEDULER.CREATE_JOB` procedure, as shown in the slide.



Creating a Job for a Program with Arguments

Programs, such as PL/SQL or external procedures, may require input arguments. Using the `DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT` procedure, you can define an argument for an existing program. The `DEFINE_PROGRAM_ARGUMENT` procedure parameters include the following:

- The `program_name` specifies an existing program that is to be altered.
- The `argument_name` specifies a unique argument name for the program.
- The `argument_position` specifies the position in which the argument is passed when the program is called.
- The `argument_type` specifies the data type of the argument value that is passed to the called program.
- The `default_value` specifies a default value that is supplied to the program if the job that schedules the program does not provide a value.

The slide shows how to create a job executing a program with one argument. The program argument default value is 50. To change the program argument value for a job, use:

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE(
  job_name => 'JOB_NAME',
  argument_name => 'DEPT_ID', argument_value => '80');
```

Creating a Job Using a Schedule

- Use `CREATE_SCHEDULE` to create a schedule:

```
BEGIN
  DBMS_SCHEDULER.CREATE_SCHEDULE('SCHED_NAME',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'FREQ=DAILY',
   end_date => SYSTIMESTAMP +15);
END;
```

- Use `CREATE_JOB` by referencing the schedule in the `schedule_name` parameter:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   schedule_name => 'SCHED_NAME',
   job_type => 'PLSQL_BLOCK',
   job_action => 'BEGIN ...; END;',
   enabled => TRUE);
END;
```

ORACLE

Creating a Job Using a Schedule

You can create a common schedule that can be applied to different jobs without having to specify the schedule details each time. The following are the benefits of creating a schedule:

- It is reusable and can be assigned to different jobs.
- Changing the schedule affects all jobs using the schedule. The job schedules are changed once, not multiple times.

A schedule is precise to only the nearest second. Although the `TIMESTAMP` data type is more accurate, the Scheduler rounds off anything with a higher precision to the nearest second.

The start and end times for a schedule are specified by using the `TIMESTAMP` data type. The `end_date` for a saved schedule is the date after which the schedule is no longer valid. The schedule in the example is valid for 15 days after using it with a specified job.

The `repeat_interval` for a saved schedule must be created by using a calendaring expression. A `NULL` value for `repeat_interval` specifies that the job runs only once.

Note: You cannot use PL/SQL expressions to express the repeat interval for a saved schedule.

Setting the Repeat Interval for a Job

- **Using a calendaring expression:**

```
repeat_interval=> 'FREQ=HOURLY; INTERVAL=4'
repeat_interval=> 'FREQ=DAILY'
repeat_interval=> 'FREQ=MINUTELY;INTERVAL=15'
repeat_interval=> 'FREQ=YEARLY;
                   BYMONTH=MAR,JUN,SEP,DEC;
                   BYMONTHDAY=15'
```

- **Using a PL/SQL expression:**

```
repeat_interval=> 'SYSDATE + 36/24'
repeat_interval=> 'SYSDATE + 1'
repeat_interval=> 'SYSDATE + 15/(24*60)'
```


ORACLE

Setting the Repeat Interval for a Job

When scheduling repeat intervals for a job, you can specify either a PL/SQL expression (if it is within a job argument) or a calendaring expression.

The examples in the slide include the following:

- `FREQ=HOURLY;INTERVAL=4` indicates a repeat interval of every four hours.
- `FREQ=DAILY` indicates a repeat interval of every day, at the same time as the start date of the schedule.
- `FREQ=MINUTELY;INTERVAL=15` indicates a repeat interval of every 15 minutes.
- `FREQ=YEARLY;BYMONTH=MAR,JUN,SEP,DEC;BYMONTHDAY=15` indicates a repeat interval of every year on March 15, June 15, September 15, and December 15.

With a calendaring expression, the next start time for a job is calculated using the repeat interval and the start date of the job.

Note: If no repeat interval is specified (that is, if a `NULL` value is provided in the argument), the job runs only once on the specified start date.

Creating a Job Using a Named Program and Schedule

- Create a named program called `PROG_NAME` by using the `CREATE_PROGRAM` procedure.
- Create a named schedule called `SCHED_NAME` by using the `CREATE_SCHEDULE` procedure.
- Create a job referencing the named program and schedule:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   program_name => 'PROG_NAME',
   schedule_name => 'SCHED_NAME',
   enabled => TRUE);
END;
/
```


ORACLE

Creating a Job Using a Named Program and Schedule

The example in the slide shows the final form for using the `DBMS_SCHEDULER.CREATE_JOB` procedure. In this example, the named program (`PROG_NAME`) and schedule (`SCHED_NAME`) are specified in their respective parameters in the call to the `DBMS_SCHEDULER.CREATE_JOB` procedure.

With this example, you can see how easy it is to create jobs by using a predefined program and schedule.

Some jobs and schedules can be too complex to cover in this course. For example, you can create windows for recurring time plans and associate a resource plan with a window. A resource plan defines attributes about the resources required during the period defined by execution window.

For more information, refer to the Online Course titled *Oracle Database 10g: Configure and Manage Jobs with the Scheduler*.

Managing Jobs

- **Run a job:**

```
DBMS_SCHEDULER.RUN_JOB('SCHEMA.JOB_NAME');
```

- **Stop a job:**

```
DBMS_SCHEDULER.STOP_JOB('SCHEMA.JOB_NAME');
```

- **Drop a job even if it is currently running:**

```
DBMS_SCHEDULER.DROP_JOB('JOB_NAME', TRUE);
```


ORACLE

Managing Jobs

After a job has been created, you can:

- Run the job by calling the `RUN_JOB` procedure specifying the name of the job. The job is immediately executed in your current session.
- Stop the job by using the `STOP_JOB` procedure. If the job is running currently, it is stopped immediately. The `STOP_JOB` procedure has two arguments:
  - **`job_name`:** Is the name of the job to be stopped
  - **`force`:** Attempts to gracefully terminate a job. If this fails and `force` is set to `TRUE`, then the job slave is terminated. (Default value is `FALSE`.) To use `force`, you must have the `MANAGE SCHEDULER` system privilege.
- Drop the job with the `DROP_JOB` procedure. This procedure has two arguments:
  - **`job_name`:** Is the name of the job to be dropped
  - **`force`:** Indicates whether the job should be stopped and dropped if it is currently running (Default value is `FALSE`.)

  If the `DROP_JOB` procedure is called and the job specified is currently running, then the command fails unless the `force` option is set to `TRUE`. If the `force` option is set to `TRUE`, then any instance of the job that is running is stopped and the job is dropped.

Note: To run, stop, or drop a job that belongs to another user, you need `ALTER` privileges on that job or the `CREATE ANY JOB` system privilege.

Data Dictionary Views

- [DBA | ALL | USER]_SCHEDULER_JOBS
- [DBA | ALL | USER]_SCHEDULER_RUNNING_JOBS
- [DBA | ALL]_SCHEDULER_JOB_CLASSES
- [DBA | ALL | USER]_SCHEDULER_JOB_LOG
- [DBA | ALL | USER]_SCHEDULER_JOB_RUN_DETAILS
- [DBA | ALL | USER]_SCHEDULER_PROGRAMS

ORACLE

Data Dictionary Views

The DBA_SCHEDULER_JOB_LOG view shows all completed job instances, both successful and failed.

To view the state of your jobs, use the following query:

```
SELECT job_name, program_name, job_type, state
FROM USER_SCHEDULER_JOBS;
```

To determine which instance a job is running on, use the following query:

```
SELECT owner, job_name, running_instance,
resource_consumer_group
FROM DBA_SCHEDULER_RUNNING_JOBS;
```

To determine information about how a job ran, use the following query:

```
SELECT job_name, instance_id, req_start_date,
actual_start_date, status
FROM ALL_SCHEDULER_JOB_RUN_DETAILS;
```

To determine the status of your jobs, use the following query:

```
SELECT job_name, status, error#, run_duration, cpu_used
FROM USER_SCHEDULER_JOB_RUN_DETAILS;
```

Summary

In this lesson, you should have learned how to:

- **Use various preinstalled packages that are provided by the Oracle server**
- **Use the following packages:**
  - **`DBMS_OUTPUT` to buffer and display text**
  - **`UTL_FILE` to write operating system text files**
  - **`HTP` to generate HTML documents**
  - **`UTL_MAIL` to send messages with attachments**
  - **`DBMS_SCHEDULER` to automate processing**
- **Create packages individually or by using the `catproc.sql` script**

ORACLE

Summary

This lesson covers a small subset of packages provided with the Oracle database. You have extensively used `DBMS_OUTPUT` for debugging purposes and displaying procedurally generated information on the screen in *i*SQL*Plus.

In this lesson, you should have learned how to use the power features provided by the database to create text files in the operating system by using `UTL_FILE`, generate HTML reports with the `HTP` package, send e-mail with or without binary or text attachments by using the `UTL_MAIL` package, and schedule PL/SQL and external code for execution with the `DBMS_SCHEDULER` package.

Note: For more information about all PL/SQL packages and types, refer to the *PL/SQL Packages and Types Reference*.

Practice 5: Overview

This practice covers the following topics:

- **Using `UTL_FILE` to generate a text report**
- **Using `HTP` to generate a Web page report**
- **Using `DBMS_SCHEDULER` to automate report processing**

ORACLE

Practice 5: Overview

In this practice, you use `UTL_FILE` to generate a text file report of employees in each department. You create a procedure to generate an HTML version of the employee report, and write a SQL script file to spool the results to a text file. You use the `DBMS_SCHEDULER` to run the first report that creates a text file every five minutes.

Practice 5

1. Create a procedure called `EMPLOYEE_REPORT` that generates an employee report in a file in the operating system, using the `UTL_FILE` package. The report should generate a list of employees who have exceeded the average salary of their departments.
   a. Your program should accept two parameters. The first parameter is the output directory. The second parameter is the name of the text file that is written.
   Note: Use the directory location value `UTL_FILE`. Add an exception-handling section to handle errors that may be encountered when using the `UTL_FILE` package.
   b. Invoke the program, using the second parameter with a name such as `sal_rpt`*xx*`.txt`, where *xx* represents your user number (for example, 01, 15, and so on). The following is a sample output from the report file:

```
Employees who earn more than average salary:
REPORT GENERATED ON 26-FEB-04
Hartstein                      20      $13,000.00
Raphaely                       30      $11,000.00
Marvis                         40      $6,500.00
     ...
     *** END OF REPORT ***
```

   Note: The data displays the employee's last name, department ID, and salary. Ask your instructor to provide instructions on how to obtain the report file from the server using the `Putty PSFTP` utility.

2. Create a new procedure called `WEB_EMPLOYEE_REPORT` that generates the same data as the `EMPLOYEE_REPORT`.
   a. First, execute `SET SERVEROUTPUT ON`, and then execute `htp.print('hello')` followed by executing `OWA_UTIL.SHOWPAGE`. The exception messages generated can be ignored.
   b. Write the `WEB_EMPLOYEE_REPORT` procedure by using the `HTP` package to generate an HTML report of employees with a salary greater than the average for their departments. If you know HTML, create an HTML table; otherwise, create simple lines of data.
   Hint: Copy the cursor definition and the `FOR` loop from the `EMPLOYEE_REPORT` procedure for the basic structure for your Web report.
   c. Execute the procedure using *i*SQL*Plus to generate the HTML data into a server buffer, and execute the `OWA_UTIL.SHOWPAGE` procedure to display contents of the buffer. Remember that `SERVEROUTPUT` should be `ON` before you execute the code.
   d. Create an HTML file called `web_employee_report.htm` containing the output result text that you select and copy from the opening `<HTML>` tag to the closing `</HTML>` tag. Paste the copied text into the file and save it to disk. Double-click the file to display the results in your default browser.

Practice 5 (continued)

3. Your boss wants to run the employee report frequently. You create a procedure that uses the DBMS_SCHEDULER package to schedule the EMPLOYEE_REPORT procedure for execution. You should use parameters to specify a frequency, and an optional argument to specify the number of minutes after which the scheduled job should be terminated.
   a. Create a procedure called SCHEDULE_REPORT that provides the following two parameters:
   - interval: To specify a string indicating the frequency of the scheduled job
   - minutes: To specify the total life in minutes (default of 10) for the scheduled job, after which it is terminated. The code divides the duration by the quantity (24 × 60) when it is added to the current date and time to specify the termination time.

   When the procedure creates a job, with the name of EMPSAL_REPORT by calling DBMS_SCHEDULER.CREATE_JOB, the job should be enabled and scheduled for the PL/SQL block to start immediately. You must schedule an anonymous block to invoke the EMPLOYEE_REPORT procedure so that the file name can be updated with a new time, each time the report is executed. The EMPLOYEE_REPORT is given the directory name supplied by your instructor for task 1, and the file name parameter is specified in the following format: sal_rptxx_hh24-mi-ss.txt, where xx is your assigned user number and hh24-mi-ss represents the hours, minutes, and seconds.

   Use the following local PL/SQL variable to construct a PL/SQL block:

```
plsql_block VARCHAR2(200) :=
'BEGIN'||
' EMPLOYEE_REPORT(''UTL_FILE'',''||
'''sal_rpt'''||'xx'||'_'''||to_char(sysdate,''HH24-MI-
SS'')||''.txt'');'||
'END;';
```

   This code is provided to help you because it is a nontrivial PL/SQL string to construct. In the PL/SQL block, **xx** is your student number.

   b. Test the SCHEDULE_REPORT procedure by executing it with a parameter specifying a frequency of every two minutes and a termination time 10 minutes after it starts.
   Note: You must connect to the database server by using PSFTP to check whether your files are created.

   c. During and after the process, you can query the job_name and enabled columns from the USER_SCHEDULER_JOBS table to check whether the job still exists.
   Note: This query should return no rows after 10 minutes have elapsed.