

Semester 2 Mid-Term Review

Semester 2 Mid Term Review

This slide set covers the following topics:

- **Creating Packages**
- **Managing Package Concepts**
- **Advanced Package Concepts**
- **Persistent State of Package Variables**
- **Using Oracle-Supplied Packages**
- **Dynamic SQL**
- **Introduction to Triggers**
- **Creating DML Triggers**

Creating Packages



What Are PL/SQL Packages?

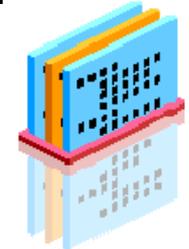
PL/SQL packages are containers that enable you to bundle related PL/SQL variables, cursors, exceptions, and subprograms.

For example, a Human Resources package could contain hiring and firing procedures, commission and bonus functions, and tax exemption variables.

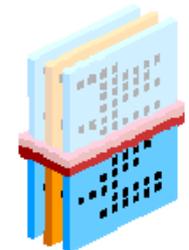
Components of a PL/SQL Package

A package consists of two parts stored separately in the database:

Package specification



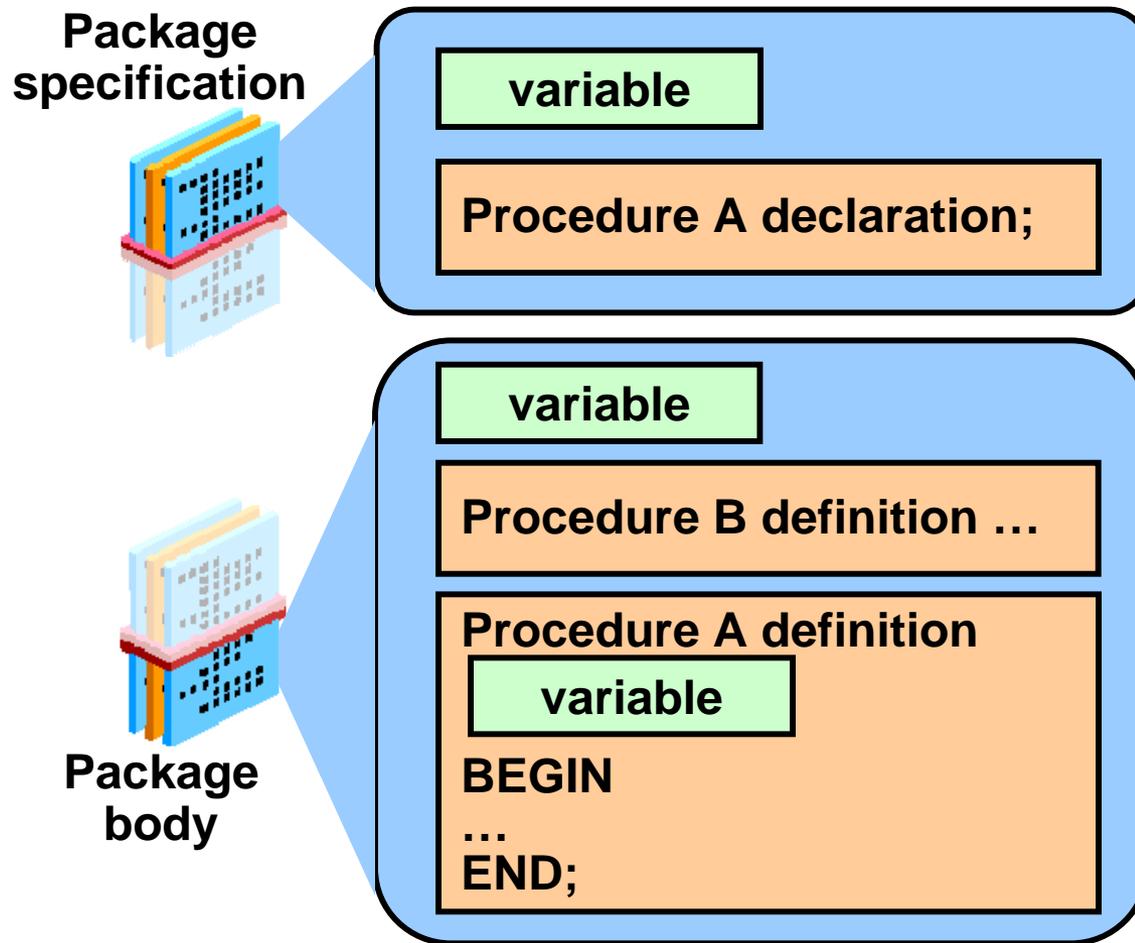
- **Package specification:** This declares the constructs (procedures, functions, variables, and so on) that can be invoked and used by applications.
- **Package body:** This contains the executable code of the subprograms that were declared in the package specification. It can also contain its own variable declarations.



Package body

The specification **must** be created before the body.

Components of a PL/SQL Package (continued)



Syntax for Creating the Package Specification

All public constructs are declared within the package specification.

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

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



Creating the Package Specification

The package specification should contain procedure and function headings terminated by a semicolon.

```
CREATE OR REPLACE PACKAGE check_emp_pkg
IS
  g_max_length_of_service CONSTANT NUMBER := 100;
  PROCEDURE chk_hiredate
    (p_date      IN      employees.hire_date%TYPE);
  PROCEDURE chk_dept_mgr
    (p_empid    IN      employees.employee_id%TYPE,
     p_mgr      IN      employees.manager_id%TYPE);
END check_emp_pkg;
```

- G_MAX_LENGTH_OF_SERVICE is a constant declared and initialized in the specification.
- CHK_HIREDATE and CHK_DEPT_MGR are two public procedures declared in the specification. Their detailed code is written in the package body.

Syntax for Creating the Package Body

Create a package body to define and implement all public subprograms and supporting private constructs.

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

- The OR REPLACE option drops and re-creates the package body.
- Identifiers defined in the package body are private and not visible outside the package body.



Creating the Package Body

When creating a package body, do the following:

- Specify the `OR REPLACE` option to overwrite an existing package body.
- Define the subprograms in an appropriate order. The basic principle is that you must declare a variable or subprogram before it can be referenced by other components in the same package body. It is common to see all private variables and subprograms defined first and the public subprograms defined last in the package body.
- The package body must complete the implementation for all procedures or functions declared in the package specification.



Creating the Package Body (continued)

```
CREATE OR REPLACE PACKAGE BODY check_emp_pkg IS
  PROCEDURE chk_hiredate
    (p_date      IN      employees.hire_date%TYPE)
  IS BEGIN
    IF MONTHS_BETWEEN(SYSDATE, p_date) >
      g_max_length_of_service * 12 THEN
      RAISE_APPLICATION_ERROR(-20200, 'Invalid Hiredate');
    END IF;
  END chk_hiredate;
  PROCEDURE chk_dept_mgr
    (p_empid     IN      employees.employee_id%TYPE,
     p_mgr       IN      employees.manager_id%TYPE)
  IS BEGIN ...
  END chk_dept_mgr;
END check_emp_pkg;
```



Invoking Subprograms

You invoke packaged procedures and functions in the same way as non-packaged subprograms, except that you must dot-prefix the subprogram name with the package name. For example:

```
BEGIN
    check_emp_pkg.chk_hiredate('17-Jul-95');
END;
```

You can DESCRIBE a package in the same way as you can DESCRIBE a table or view:

```
DESCRIBE check_emp_pkg
```

You cannot DESCRIBE individual packaged subprograms, only the whole package.

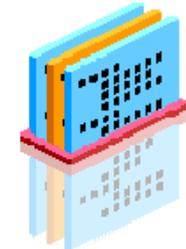
Managing Package Concepts

Components of a PL/SQL Package

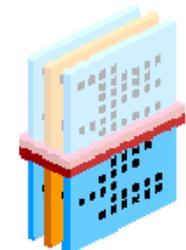
Public subprograms are declared in the package specification. You can invoke public subprograms from any calling environment, provided the user has been granted `EXECUTE` privilege on the package.

Private subprograms are declared only in the package body and can be invoked only by other (public or private) subprograms within the same package body. They cannot be invoked from outside the package.

**Package
specification**



(Public)



**Package
body
(Private)**



Scope of Package Components

The *scope* of a package component states where the component can be seen and invoked.

Because public subprograms can be seen (by `DESCRIBE package_name`) and invoked from inside or outside the package, you can say that the scope of a public subprogram includes the whole of the package, and the calling environment.

Because private subprograms cannot be seen or invoked from outside the package, you can say that the scope of a private subprogram is limited to the package in which it is defined.



Global and Local Package Variables

Variables in packages are also public or private.

If a variable is declared in the specification, it is public. If it is declared in the package body, it is private, and can be referenced only within the package.

However, you use different words when describing the scope of package variables. You call a public variable a global variable, and you call a private variable a local variable.

Explicit cursors and user-named exceptions are also variables, and are therefore also global or local.

 **Tell Me / Show Me**

Example of package specification: salary_pkg:

You have a business rule that no employee's salary can be increased by more than 20 percent at one time.

```
CREATE OR REPLACE PACKAGE salary_pkg
IS
    g_max_sal_raise CONSTANT NUMBER := 0.20;
    PROCEDURE update_sal
        (p_employee_id IN employees.employee_id%TYPE,
         p_new_salary IN employees.salary%TYPE);
END salary_pkg;
```

- g_max_sal_raise is a global constant initialized to 0.20.
- update_sal is a public procedure that updates an employee's salary.

 **Tell Me / Show Me**

Example of package body: salary_pkg:

```
CREATE OR REPLACE PACKAGE BODY salary_pkg IS
  FUNCTION validate_raise ...
END validate_raise;
PROCEDURE update_sal      ... IS
  v_old_salary employees.salary%TYPE; ...
END update_sal;
END salary_pkg;
```

- `validate_raise` is a private function and cannot be referenced from outside the package.
- `v_old_salary` is a local variable and cannot be referenced outside the `update_sal` procedure.

Invoking Package Subprograms

After the package is stored in the database, you can invoke subprograms stored within the same package or stored externally to the package.

Within the same package	Specify the subprogram name: <pre>subprogram</pre> <p>You can fully qualify a subprogram within the same package, but this is optional</p> <pre>package_name.subprogram</pre>
External to the package	Fully qualify the subprogram with its package name: <pre>package_name.subprogram</pre>

Invoking Public Package Constructs

- Invoke a public procedure:

```
BEGIN
  salary_pkg.update_sal(100,15000);
END;
```

- Reference a global variable:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(salary_pkg.g_max_sal_raise);
END;
```



Removing Packages

- To remove the **entire** package, specification and body, use the following syntax:

```
DROP PACKAGE package_name;
```

- To remove **only** the package body, use the following syntax:

```
DROP PACKAGE BODY package_name;
```

Viewing Packages in the Data Dictionary

The source code for PL/SQL packages is viewable through the `USER_SOURCE` and `ALL_SOURCE` dictionary views.

- To view the package specification, use:

```
SELECT text
FROM   USER_SOURCE
WHERE  name = 'SALARY_PKG' AND type = 'PACKAGE'
ORDER BY line;
```

- To view the package body, use:

```
SELECT text
FROM   USER_SOURCE
WHERE  name = 'SALARY_PKG' AND type = 'PACKAGE BODY'
ORDER BY line;
```



Guidelines for Writing Packages

- Construct packages for general use.
- Define the package specification before the body.
- The package specification should contain only those constructs that you want to be public.
- Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions.
- Only recompile the body, if possible, because changes to the package specification require recompilation of all programs that call the package.
- The package specification should contain as few constructs as possible.

Advantages of Using Packages

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

Advantages of Using Packages (continued)

- Added functionality: Persistency of variables and cursors
- Better performance:
 - The entire package is loaded into memory when the package is first referenced.
 - There is only one copy in memory for all users.
 - The dependency hierarchy is simplified.
- Overloading: Multiple subprograms of the same name

Advanced Package Concepts



Overloading Subprograms

The overloading feature in PL/SQL enables you to develop two or more packaged subprograms with the same name. Overloading is useful when you want a subprogram to accept similar sets of parameters that have different data types.

The overloading feature in PL/SQL:

- Enables you to build flexible ways for invoking the same subprograms with different data
- Makes tasks easier for the application developer, who has to remember only one subprogram name.
- Overloading can be done with subprograms in packages, but not with standalone subprograms.
- The key rule is that you can use the same name for different subprograms, as long as their formal parameters differ in number, order, or category of data type.



Overloading Subprograms

Consider using overloading when the purposes of two or more subprograms are similar, but the type or number of parameters used varies.

Overloading can provide alternative ways for finding different data with varying search criteria. For example, you might want to find employees by their employee ID, and also provide a way to find employees by their job ID, or by their hire date. The purpose is the same, but the parameters or search criteria differ.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE find_emp
    (p_employee_id IN NUMBER, p_last_name OUT VARCHAR2);
  PROCEDURE find_emp
    (p_job_id IN VARCHAR2, p_last_name OUT VARCHAR2);
  PROCEDURE find_emp
    (p_hiredate IN DATE, p_last_name OUT VARCHAR2);
END dept_pkg;
```



Overloading Restrictions

You cannot overload:

- Two subprograms if their formal parameters differ only in data type and the different data types are in the same category (NUMBER and INTEGER belong to the same category; VARCHAR2 and CHAR belong to the same category)
- Two functions that differ only in return type, even if the types are in different categories

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



Overloading: Example 2

```
CREATE PACKAGE sample_pack IS
  PROCEDURE sample_proc (p_char_param IN CHAR)          IS ...;
  PROCEDURE sample_proc (p_varchar_param IN VARCHAR2) IS ...;
END sample_pack;
```

Now you invoke a procedure using positional notation:

```
BEGIN
  sample_pack.sample_proc('Smith');
END;
```

This fails because 'Smith' can be either CHAR or VARCHAR2. But the following invocation succeeds:

```
BEGIN
  sample_pack.sample_proc(p_char_param => 'Smith');
END;
```



Overloading: Example 3

```
CREATE OR REPLACE PACKAGE dept_pkg IS
  PROCEDURE add_department(p_deptno NUMBER,
    p_name VARCHAR2 := 'unknown', p_loc NUMBER := 1700);
  PROCEDURE add_department(
    p_name VARCHAR2 := 'unknown', p_loc NUMBER := 1700);
END dept_pkg;
```

In this example, the `dept_pkg` package specification contains an overloaded procedure called `add_department`. The first declaration takes three parameters that are used to provide data for a new department record inserted into the department table. The second declaration takes only two parameters, because this version internally generates the department ID through an Oracle sequence.

 Overloading: Example 3 (continued)

```
CREATE OR REPLACE PACKAGE BODY dept_pkg IS
  PROCEDURE add_department (p_deptno NUMBER,
    p_name VARCHAR2:='unknown', p_loc NUMBER:=1700)
  IS
  BEGIN
    INSERT INTO departments(department_id,
      department_name, location_id)
      VALUES (p_deptno, p_name, p_loc);
  END add_department;

  PROCEDURE add_department (
    p_name VARCHAR2:='unknown', p_loc NUMBER:=1700)
  IS
  BEGIN
    INSERT INTO departments (department_id,
      department_name, location_id)
      VALUES (departments_seq.NEXTVAL, p_name, p_loc);
  END add_department;
END dept_pkg;
```

 **Overloading: Example 3 (continued)**

If you call `add_department` with an explicitly provided department ID, then PL/SQL uses the first version of the procedure. Consider the following example:

```
BEGIN
  dept_pkg.add_department(980,'Education',2500);
END;

SELECT * FROM departments
WHERE department_id = 980;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
980	Education	-	2500



Overloading: Example 3 (continued)

If you call `add_department` with no department ID, then PL/SQL uses the second version:

```
BEGIN
  dept_pkg.add_department ('Training', 2500);
END;

SELECT * FROM departments
WHERE department_name = 'Training';
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
290	Training	-	2500

Overloading and the STANDARD Package

- A package named STANDARD defines the PL/SQL environment and built-in functions.
- Most built-in functions are overloaded. You have already seen that TO_CHAR is overloaded. Another example is the UPPER function:

```
FUNCTION UPPER (ch VARCHAR2) RETURN VARCHAR2;  
FUNCTION UPPER (ch CLOB) RETURN CLOB;
```

- You do not prefix STANDARD package subprograms with the package name.

What if You Create Your Own Function With the Same Name as a STANDARD Package Function?

For example, you create your own `UPPER` function. Then you invoke `UPPER(argument)`. Which one is executed?

Answer: Even though your function is in your own schema, the built-in `STANDARD` function is executed. To call your own function, you need to prefix it with your schema name:

```
...  
BEGIN  
    v_return_value := your-schema-name.UPPER(argument);  
END;
```

Using Forward Declarations

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

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

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

- `calc_rating` is referenced (in `award_bonus`) before it has been declared.



Using Forward Declarations

You can solve the illegal reference problem by reversing the order of the two procedures.

However, coding standards often require that subprograms be kept in alphabetical sequence to make them easy to find. In this case, you might encounter problems, as shown in the slide example.

Note: The compilation error for `calc_rating` occurs only if `calc_rating` is a private packaged subprogram. If `calc_rating` is declared in the package specification, then it is already declared as if it was a forward declaration, and its reference can be resolved by the PL/SQL compiler.



Using Forward Declarations

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

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
  PROCEDURE calc_rating (...);-- forward declaration
  -- Subprograms defined in alphabetical order

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

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



Using Forward Declarations (continued)

- Forward declarations help to:
 - Define subprograms in logical or alphabetical order
 - Define mutually recursive subprograms. Mutually recursive programs are programs that call each other directly or indirectly.
 - Group and logically organize subprograms in a package body
- When creating a forward declaration:
 - The formal parameters must appear in both the forward declaration and the subprogram body
 - The subprogram body can appear anywhere after the forward declaration, but both must appear in the same package body.



Package Initialization Block

Suppose you want to automatically execute some code every time you make the first call to a package in your session? For example, you want to automatically load a tax rate into a package variable.

If the tax rate is a constant, you can initialize the package variable as part of its declaration:

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
  g_tax  NUMBER := 0.20;
  ...
END taxes_pkg;
```

But what if the tax rate is stored in a database table ?



Package Initialization Block

Optionally, you can include an un-named block at the end of the package body. This block automatically executes once and is used to initialize public and private package variables.

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
  g_tax  NUMBER;
  ... -- declare all public procedures/functions
END taxes_pkg;

CREATE OR REPLACE PACKAGE BODY taxes_pkg IS
  ... -- declare all private variables
  ... -- define public/private procedures/functions
BEGIN
  SELECT  rate_value INTO g_tax
  FROM    tax_rates
  WHERE   rate_name = 'TAX';
END taxes_pkg;
```

Restrictions on Using Package Functions in SQL Statements

- Package functions can be used in SQL statements.
- Functions called from:
 - A query or DML statement must not end the current transaction, create or roll back to a savepoint, or alter the system or session.
 - A query or a parallelized DML statement cannot execute a DML statement or modify the database.
 - A DML statement cannot read or modify the table being changed by that DML statement.

Note: A function calling subprograms that break the preceding restrictions is not allowed.



Package Function in SQL: Example 1

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
    FUNCTION tax (p_value IN NUMBER) RETURN NUMBER;
END taxes_pkg;

CREATE OR REPLACE PACKAGE BODY taxes_pkg IS
    FUNCTION tax (p_value IN NUMBER) RETURN NUMBER IS
        v_rate NUMBER := 0.08;
    BEGIN
        RETURN (p_value * v_rate);
    END tax;
END taxes_pkg;
```

```
SELECT taxes_pkg.tax(salary), salary, last_name
FROM employees;
```

This call succeeds because the function does not contain any of the restrictions.



Package Function in SQL: Example 2

```
CREATE OR REPLACE PACKAGE sal_pkg IS
  FUNCTION sal (p_emp_id IN NUMBER) RETURN NUMBER;
END sal_pkg;

CREATE OR REPLACE PACKAGE BODY sal_pkg IS
  FUNCTION sal (p_emp_id IN NUMBER) RETURN NUMBER IS
    v_sal employees.salary%TYPE;
  BEGIN
    UPDATE employees SET salary = salary * 2
      WHERE employee_id = p_emp_id;
    SELECT salary INTO v_sal FROM employees
      WHERE employee_id = p_emp_id;
    RETURN (v_sal);
  END sal;
END sal_pkg;
```

```
SELECT sal_pkg.sal(employee_id), salary, last_name
FROM employees;
```

This fails because the function contains an UPDATE .



Persistent State of Package Variables



Package State

The collection of package variables and their current values define the package state. The package state is:

- Initialized when the package is first loaded
- Persistent (by default) for the life of the session
 - Stored in the session's private memory area
 - Unique to each session
 - Subject to change when package subprograms are called or public variables are modified.

Other sessions each have their own package state, and do not see your changes.

 **Package State (continued)**

The following is a simple package that initializes a single global variable and contains a procedure to update it:

```
CREATE OR REPLACE PACKAGE pers_pkg IS
    g_var    NUMBER := 10;
    PROCEDURE upd_g_var (p_var IN NUMBER);
END pers_pkg;

CREATE OR REPLACE PACKAGE BODY pers_pkg IS
    PROCEDURE upd_g_var (p_var IN NUMBER) IS
        BEGIN
            g_var := p_var;
        END upd_g_var;
END pers_pkg;

GRANT EXECUTE ON pers_pkg TO SCOTT, JONES;
```

SCOTT and JONES call the procedure to update the variable.

Package State (continued)

The following sequence of events occurs:

State for:
Scott Jones

Time	Event	Scott	Jones
9:00	Scott> .. svar := pers_pkg.g_var;	10	-
9:30	Jones> .. jvar := pers_pks.g_var; Jones> .. pers_pkg.upd_g_var(20); Scott> .. svar := pers_pkg.g_var;	10	10 20
9:35	Scott> .. pers_pkg.upd_g_var(50); Jones> .. jvar := pers_pks.g_var;	50	20
10:00	Scott disconnects and reconnects in a new session		
10:05	Scott> .. svar := pers_pkg.g_var;	10	



Explanation of the Events on the Previous Slide

- At 9:00: Scott connects and reads the variable, seeing the initialized value 10.
- At 9:30: Jones connects and also reads the variable, also seeing the initialized value 10. At this point, there are two separate and independent copies of the value, one in each session's private memory area. Jones now updates his own session's value to 20 using the procedure. Scott then re-reads the variable but does not see Jones's change.
- At 9:35: Scott updates his own session's value to 50. Again, Jones cannot see the change.
- At 10:00: Scott disconnects and reconnects, creating a new session.
- At 10:05: Scott reads the variable and sees the initialized value 10.
- These changes are not visible in other sessions even if both sessions are connected under the same user name.

Persistent State of a Package Cursor

A cursor declared in the package specification is a type of global variable, and follows the same persistency rules as any other variable. A cursor's state is not defined by a single numeric or other value. A cursor's state consists of the following attributes:

- Whether the cursor is open or closed
- (If open) how many rows have been fetched from the cursor (%ROWCOUNT) and whether the most recent fetch was successful (%NOTFOUND).

Package specification:

```
CREATE OR REPLACE PACKAGE curs_pkg IS
  CURSOR emp_curs IS SELECT employee_id FROM employees;
  PROCEDURE open_curs;
  FUNCTION fetch_n_rows(n NUMBER := 1) RETURN BOOLEAN;
  PROCEDURE close_curs;
END curs_pkg;
```



Persistent State of a Package Cursor: Package Body

```
CREATE OR REPLACE PACKAGE BODY curs_pkg IS
  PROCEDURE open_curs IS
  BEGIN
    IF NOT emp_curs%ISOPEN THEN OPEN emp_curs;
    END IF;
  END open_curs;

  FUNCTION fetch_n_rows(n NUMBER := 1) RETURN BOOLEAN IS
    emp_id employees.employee_id%TYPE;
  BEGIN
    FOR count IN 1 .. n LOOP
      FETCH emp_curs INTO emp_id;
      EXIT WHEN emp_curs%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Id: ' || (emp_id));
    END LOOP;
    RETURN emp_curs%FOUND;
  END fetch_n_rows;

  PROCEDURE close_curs IS BEGIN
    IF emp_curs%ISOPEN THEN CLOSE emp_curs;
    END IF;
  END close_curs;
END curs_pkg;
```

 Invoking CURS_PKG

```
DECLARE
  v_more_rows_exist BOOLEAN := TRUE;
BEGIN
  curs_pkg.open_curs;                                --1
  LOOP
    v_more_rows_exist := curs_pkg.fetch_n_rows(3);  --2
    EXIT WHEN NOT v_more_rows_exist;
  END LOOP;
  curs_pkg.close_curs;                               --3
END;
```

1. Opens the cursor.
2. Fetches (in a loop) and displays three rows at a time until all rows are fetched.
3. Closes the cursor.

Even though the `fetch_n_rows` function does not contain an `OPEN`, the `FETCH` does not fail because the cursor is still open from the previous call to the procedure.



Invoking CURS_PKG

```
DECLARE
  v_more_rows_exist BOOLEAN := TRUE;
BEGIN
  curs_pkg.open_curs;
  LOOP
    v_more_rows_exist := curs_pkg.fetch_n_rows(3);
    EXIT WHEN NOT v_more_rows_exist;
  END LOOP;
  curs_pkg.close_curs;
END;
```

- The first looped call to `fetch_n_rows` displays the first three rows. The second time round the loop, the next three rows are fetched and displayed. And so on.
- This technique is often used in applications that need to `FETCH` a large number of rows from a cursor, but can only display them to the user one screen at a time.

Using Oracle-Supplied Packages

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 might simply want to use these packages as ideas when you create your own stored procedures.

 **List of Some Oracle-Supplied Packages**

DBMS_LOB	Enables manipulation of Oracle Large Object column datatypes: CLOB, BLOB, and BFILE
DBMS_LOCK	Used to request, convert, and release locks in the database through Oracle Lock Management services
DBMS_OUTPUT	Provides debugging and buffering of messages
HTP	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 of PL/SQL blocks, stored procedures, and external procedures or executables



The DBMS_OUTPUT Package

The `DBMS_OUTPUT` package sends text messages from any PL/SQL block into a private memory area called a buffer, from which the messages can be displayed. It is provided to enable testing and debugging of PL/SQL programs.

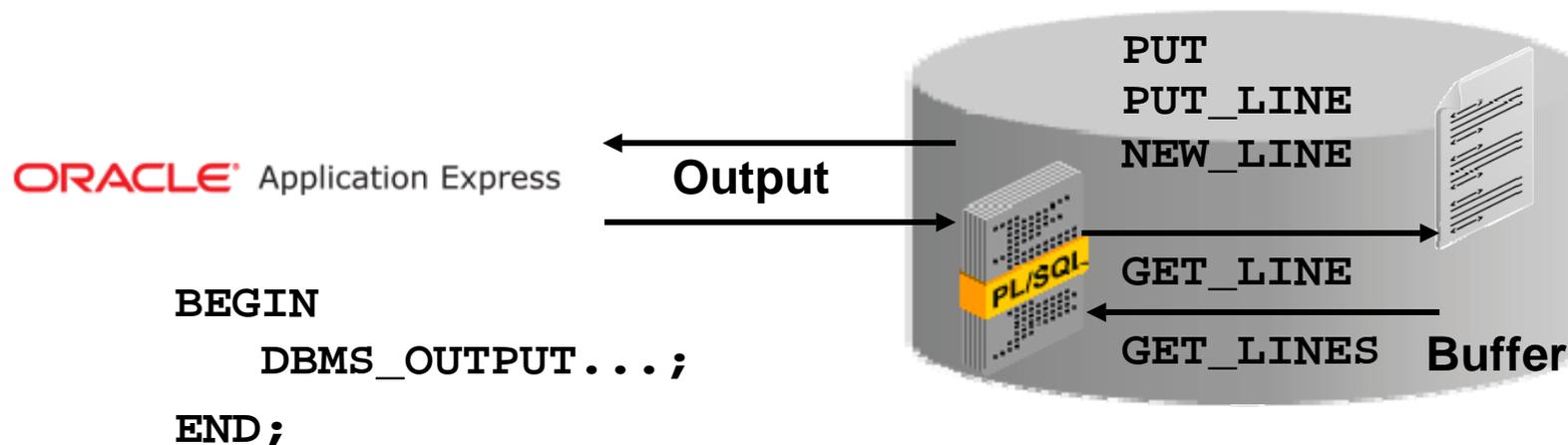
Common uses of `DBMS_OUTPUT` include:

- You can output results back to the developer for debugging purposes.
- You can trace the code execution path for a function or procedure.

How the DBMS_OUTPUT Package Works

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

- PUT places text in the buffer.
- NEW_LINE sends the buffer to the screen, then empties the buffer.
- PUT_LINE does a PUT followed by a NEW_LINE.
- GET_LINE and GET_LINES read the buffer.
- Messages are not sent until after the calling block finishes.



 **DBMS_OUTPUT Is Designed for Debugging Only**

For this reason, you should not use `DBMS_OUTPUT` in subprograms, but only in anonymous PL/SQL blocks for testing purposes. Instead of:

```
CREATE OR REPLACE PROCEDURE do_some_work IS BEGIN
    ... DBMS_OUTPUT.PUT_LINE('string'); ... END;

BEGIN    do_some_work;    END;    -- Test the procedure
```

you should use:

```
CREATE OR REPLACE PROCEDURE do_some_work
    (p_output OUT VARCHAR2) IS BEGIN
    ... p_output := 'string'; ... END;

DECLARE v_output VARCHAR2(100); BEGIN --Test the procedure
    do_some_work(v_output);
    DBMS_OUTPUT.PUT_LINE(v_output);    END;
```

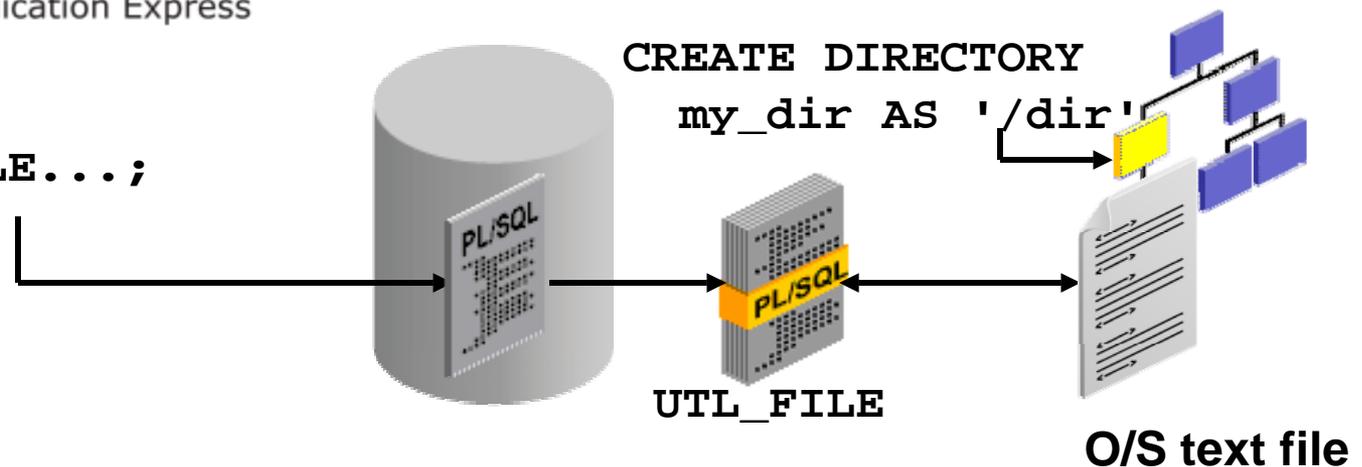
The UTL_FILE Package

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

UTL_FILE can access files in operating system directories defined by a CREATE DIRECTORY statement. You can also use the `utl_file_dir` database parameter.

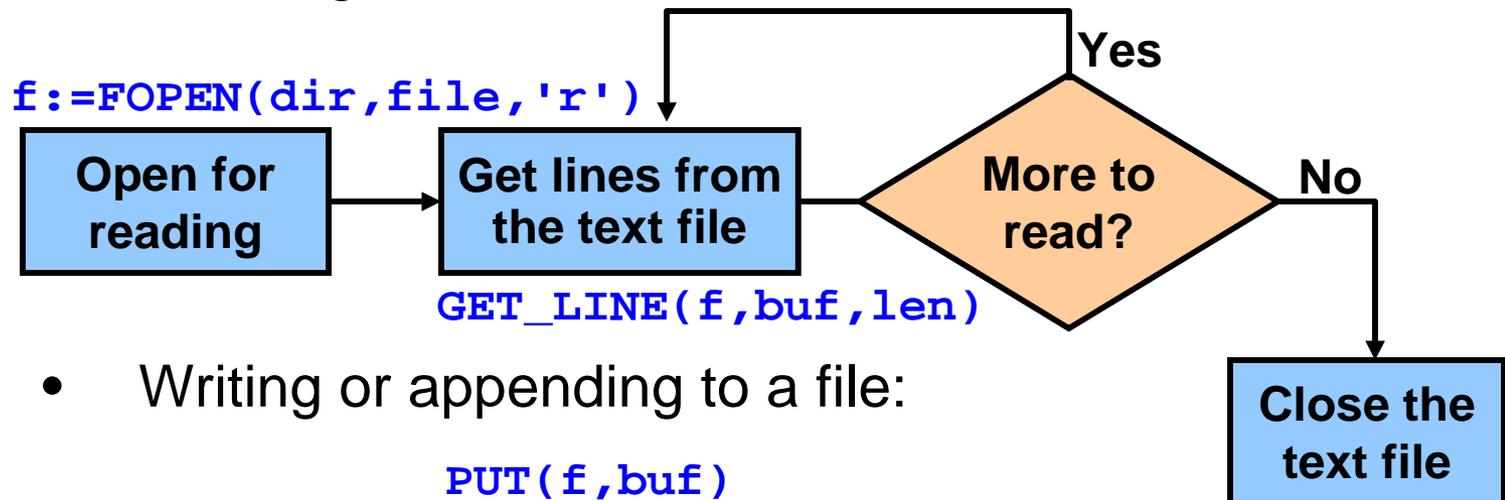
ORACLE Application Express

```
BEGIN
  UTL_FILE...;
END;
```

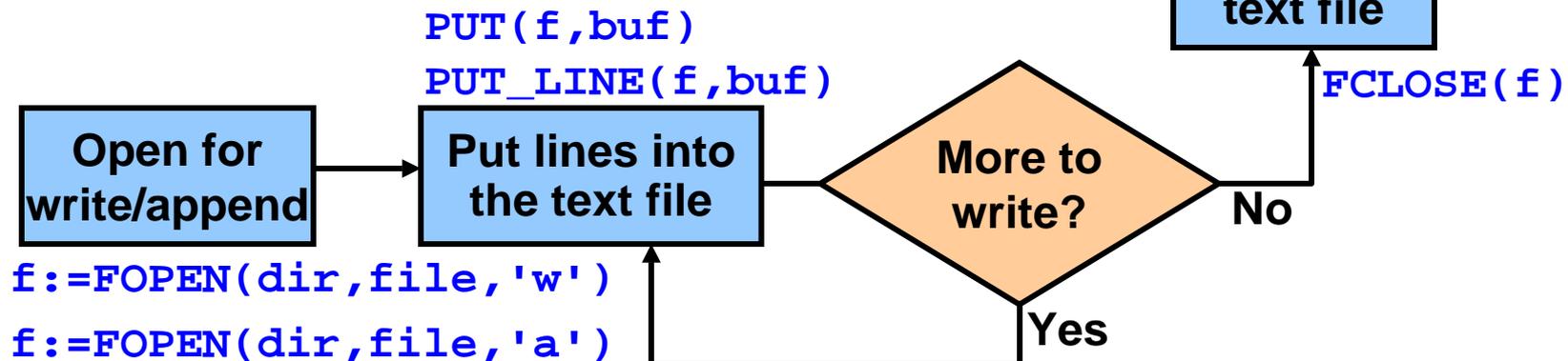


File Processing Using the UTL_FILE Package

- Reading a file:



- Writing or appending to a file:



File Processing Using the UTL_FILE Package

You open files for reading or writing with the `FOPEN` function. You then either read from or write or append to the file until processing is done. Then, close the file by using the `FCLOSE` procedure. The following are the main subprograms:

- The `FOPEN` function opens a file in a specified directory for input/output (I/O) and returns a file handle used in later I/O operations.
- 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.
- The `PUT_LINE` procedure writes text to the opened file.
- The `NEW_LINE` procedure writes a blank line to the file.
- The `FCLOSE` procedure closes an opened file.



Exceptions in the UTL_FILE Package

You might have to handle one of the following exceptions when using UTL_FILE subprograms:

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

The other exceptions not in the UTL_FILE package are:

- NO_DATA_FOUND
- VALUE_ERROR

Opening a Text File using `UTL_FILE.FOPEN`

To open a text file, the `FOPEN` function needs to know:

- In which operating system directory the file resides
- The name of the file in that directory
- Whether you want to write to the file or read from it

You pass this information as three `IN` parameters to the `FOPEN` function. The `FOPEN` function opens the file (if it exists) and returns a “file handle” (a pointer to the file) into a variable whose datatype must be `UTL_FILE.FILE_TYPE`. This is a special datatype used only with `UTL_FILE`.

```
DECLARE
  v_file_handle UTL_FILE.FILE_TYPE;
BEGIN
  v_file_handle := UTL_FILE.FOPEN ('MYDIR', 'myfile.txt', 'r');
  ...
END;
```



Reading and Closing the Text File Using UTL_FILE.GET_LINE

When the file is open, all calls to UTL_FILE must pass the file handle that was returned by FOPEN. You can read one line of text at a time using GET_LINE. When you reach the end of the file, a NO_DATA_FOUND exception is raised. Then, you close the file using FCLOSE.

```
DECLARE
  v_file_handle UTL_FILE.FILE_TYPE;
  v_text_line   VARCHAR2(200);
BEGIN
  v_file_handle := UTL_FILE.FOPEN ('MYDIR','myfile.txt','r');
  LOOP
    UTL_FILE.GET_LINE(v_file_handle, v_text_line);
    DBMS_OUTPUT.PUT_LINE(v_text_line);
  END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    UTL_FILE.FCLOSE(v_file_handle);
END;
```



Writing a Text File using UTL_FILE

This is similar to reading a file, except that you must open the file using 'w' for "write," and write to it using PUT_LINE instead of GET_LINE. You do not need to test for NO_DATA_FOUND because you decide when you are finished writing!

```
DECLARE
  v_file_handle UTL_FILE.FILE_TYPE;
BEGIN
  v_file_handle := UTL_FILE.FOPEN ('MYDIR','newfile.txt','w');
  FOR I IN 1..5 LOOP
    UTL_FILE.PUT_LINE(v_file_handle, 'This is line: ' || i);
  END LOOP;
  UTL_FILE.FCLOSE(v_file_handle);
END;
```

Dynamic SQL



Execution Flow of SQL

All SQL statements in the database go through various stages, including:

- PARSE: Pre-execution “is this possible?” checks including syntax, object existence, privileges, and so on
- EXECUTE: The statement is executed
- FETCH: Results are returned to the user (SELECT only).

When an SQL statement is included in a PL/SQL subprogram, the parse phase is normally done at compile time, that is, when the procedure, function, or package body is `CREATED`.

If the text of the SQL statement is not exactly known when the subprogram is created, the PL/SQL compiler cannot parse it.



Dynamic SQL

You use dynamic SQL to create an SQL statement whose text is not completely known at compile time.

Dynamic SQL:

- Is constructed and stored as a character string within a subprogram
- Enables data definition language (DDL) and data control language (DCL) statements to be written and executed from PL/SQL
- Is executed with Native Dynamic SQL statements or the `DBMS_SQL` package



Native Dynamic SQL

- Provides the ability to execute SQL statements whose structure is unknown until execution time
- Is supported by the following PL/SQL statements:
 - EXECUTE IMMEDIATE
 - OPEN-FOR
 - FETCH
 - CLOSE



Using the EXECUTE IMMEDIATE Statement

```
EXECUTE IMMEDIATE dynamic_string;
```

dynamic_string is a SQL statement. It can be a character literal, or the name of a CHAR or VARCHAR2 variable containing an SQL statement:

```
CREATE PROCEDURE drop_any_table(p_table_name VARCHAR2) IS
BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE ' || p_table_name;
END;
```

```
CREATE PROCEDURE drop_any_table(p_table_name VARCHAR2) IS
    v_dynamic_stmt VARCHAR2(50);
BEGIN
    v_dynamic_stmt := 'DROP TABLE ' || p_table_name;
    EXECUTE IMMEDIATE v_dynamic_stmt;
END;
```

 **You Can Use Dynamic SQL With DML Statements:**

```
CREATE FUNCTION del_rows(p_table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
    EXECUTE IMMEDIATE 'DELETE FROM ' || p_table_name;
    RETURN SQL%ROWCOUNT;
END;
```

Invoke the function just created:

```
DECLARE
    v_count    NUMBER;
BEGIN
    v_count := del_rows('EMPLOYEE_NAMES');
    DBMS_OUTPUT.PUT_LINE(v_count || ' rows deleted.');
```

```
END;
```



Dynamic SQL with DML Statements

- Inserting a row into a table with two columns:

```
CREATE PROCEDURE add_row(p_table_name VARCHAR2,  
    p_id NUMBER, p_name VARCHAR2) IS  
BEGIN  
    EXECUTE IMMEDIATE 'INSERT INTO ' || p_table_name ||  
        ' VALUES (p_id, p_name)';  
END;
```

- Invoke the procedure just created:

```
BEGIN  
    add_row('EMPLOYEE_NAMES', 250, 'Chang');  
END;
```



Using Native Dynamic SQL to Compile PL/SQL Code

Compile PL/SQL code with the `ALTER` statement:

```
ALTER PROCEDURE procedure_name COMPILE;  
ALTER FUNCTION function_name COMPILE;  
ALTER PACKAGE package_name COMPILE SPECIFICATION;  
ALTER PACKAGE package_name COMPILE BODY;
```

Create a procedure that recompiles a PL/SQL object whose name and type is entered at run-time:

```
CREATE PROCEDURE compile_plsql  
  (p_name VARCHAR2,p_type VARCHAR2,p_options VARCHAR2 := NULL)  
IS  
  v_stmt VARCHAR2(200);  
BEGIN  
  v_stmt := 'ALTER ' || p_type || ' ' || p_name || ' COMPILE'  
           || ' ' || p_options;  
  EXECUTE IMMEDIATE v_stmt;  
END;
```



Using the DBMS_SQL Package

Before Oracle Version 8i, the `EXECUTE IMMEDIATE` statement did not exist in PL/SQL, and the presupplied `DBMS_SQL` package was the only way to write dynamic SQL.

Some of the procedures and functions of the `DBMS_SQL` package are:

- `OPEN_CURSOR`
- `PARSE`
- `BIND_VARIABLE`
- `EXECUTE`
- `FETCH_ROWS`
- `CLOSE_CURSOR`

Comparison of Native Dynamic SQL and the DBMS_SQL Package

Native Dynamic SQL:

- Is easier to use than DBMS_SQL
- Requires less code than DBMS_SQL
- Often executes faster than DBMS_SQL because there are fewer statements to execute.

Introduction to Triggers

What Is a Trigger?

A database trigger:

- Is a PL/SQL block associated with a specific action (an event) on a database object, such as a table or view.
- Is stored in the database.
- Executes automatically whenever the associated action occurs.

PL/SQL Block

|
association
|

**Table, View,
Schema,
Database, or
Application**

Application Triggers Compared to Database Triggers

- Database triggers execute automatically whenever a data event (such as DML or DDL) or a system event (such as a user connecting, or the DBA shutting down the database) occurs on a schema or database. Database triggers are created and stored in the database, just like PL/SQL procedures, functions and packages.
- Application triggers execute automatically whenever a particular event occurs within an application. An example of an application that uses triggers extensively is an application developed with Oracle Forms Developer.

Triggering Events for Database Triggers

- DML operations on a table
- DML operations on a view, with an `INSTEAD OF` trigger
- DDL statements, such as `CREATE` and `ALTER`
- Database system events, such as when a user logs on or the DBA shuts down the database



Uses of Triggers

You can use triggers to:

- Enhance complex database security rules
- Create auditing records automatically
- Enforce complex data integrity rules
- Create logging records automatically
- Prevent tables from being accidentally dropped
- Prevent invalid DML transactions from occurring
- Generate derived column values automatically
- Maintain synchronous table replication
- Gather statistics on table access
- Modify table data when DML statements are issued against views



Guidelines for Triggers

Do not define triggers to duplicate or replace items you can do more easily in other ways. For example, implement simple data integrity rules using constraints, not triggers.

Excessive use of triggers can result in complex interdependencies, which can be difficult to maintain. Use triggers only when necessary, and be aware of recursive and cascading effects.

Avoid lengthy trigger logic by creating stored procedures or packaged procedures that are invoked in the trigger body.

 **Comparison of Database Triggers and Stored Procedures**

Triggers	Procedures
<p>Defined with <code>CREATE TRIGGER</code></p> <p>Data Dictionary contains source code in <code>USER_TRIGGERS</code>.</p> <p>Implicitly invoked.</p> <p><code>COMMIT</code>, <code>SAVEPOINT</code>, and <code>ROLLBACK</code> are not allowed.</p>	<p>Defined with <code>CREATE PROCEDURE</code></p> <p>Data Dictionary contains source code in <code>USER_SOURCE</code>.</p> <p>Explicitly invoked.</p> <p><code>COMMIT</code>, <code>SAVEPOINT</code>, and <code>ROLLBACK</code> are allowed.</p>

Creating DML Triggers: Part I



What Is a DML Trigger?

A DML trigger is a trigger that is automatically fired (executed) whenever a SQL DML statement (`INSERT`, `UPDATE`, or `DELETE`) is executed. You classify DML triggers in two ways:

- By when they execute: `BEFORE`, `AFTER`, or `INSTEAD OF` the triggering DML statement
- By how many times they execute: Once for the whole DML statement (a statement trigger), or once for each row affected by the DML statement (a row trigger).



Types of DML Triggers

The trigger type determines if the body executes for each row or only once for the triggering statement.

- A statement trigger:
 - Executes once for the triggering event
 - Is the default type of trigger
 - Fires once even if no rows are affected at all
- A row trigger:
 - Executes once for each row affected by the triggering event
 - Is not executed if the triggering event does not affect any rows
 - Is indicated by specifying the `FOR EACH ROW` clause



Creating DML Statement Triggers

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
    event1 [OR event2 OR event3] ON object_name
    trigger_body
```

timing: When the trigger fires in relation to the triggering event. Values are BEFORE, AFTER, or INSTEAD OF.

event: Which DML operation causes the trigger to fire. Values are INSERT, UPDATE [OF column], and DELETE.

object_name: The table or view associated with the trigger.

trigger_body: The action(s) performed by the trigger; it is a PL/SQL block with BEGIN and END;

.



Trigger Timing

When should the trigger fire?

- **BEFORE:** Execute the trigger body before the triggering DML event completes on a table.
- **AFTER:** Execute the trigger body after the triggering DML event completes on a table.
- **INSTEAD OF:** This is used on views that are not otherwise modifiable. Execute the trigger body instead of the triggering statement.

Statement Trigger Firing Sequence

This slide shows the firing sequence for a statement trigger associated with the event `INSERT ON departments`. A statement trigger always fires exactly once, no matter how many rows are affected by the triggering statement.

```
INSERT INTO departments
  (department_id, department_name, location_id)
VALUES (400, 'CONSULTING', 2500);
```

Triggering action

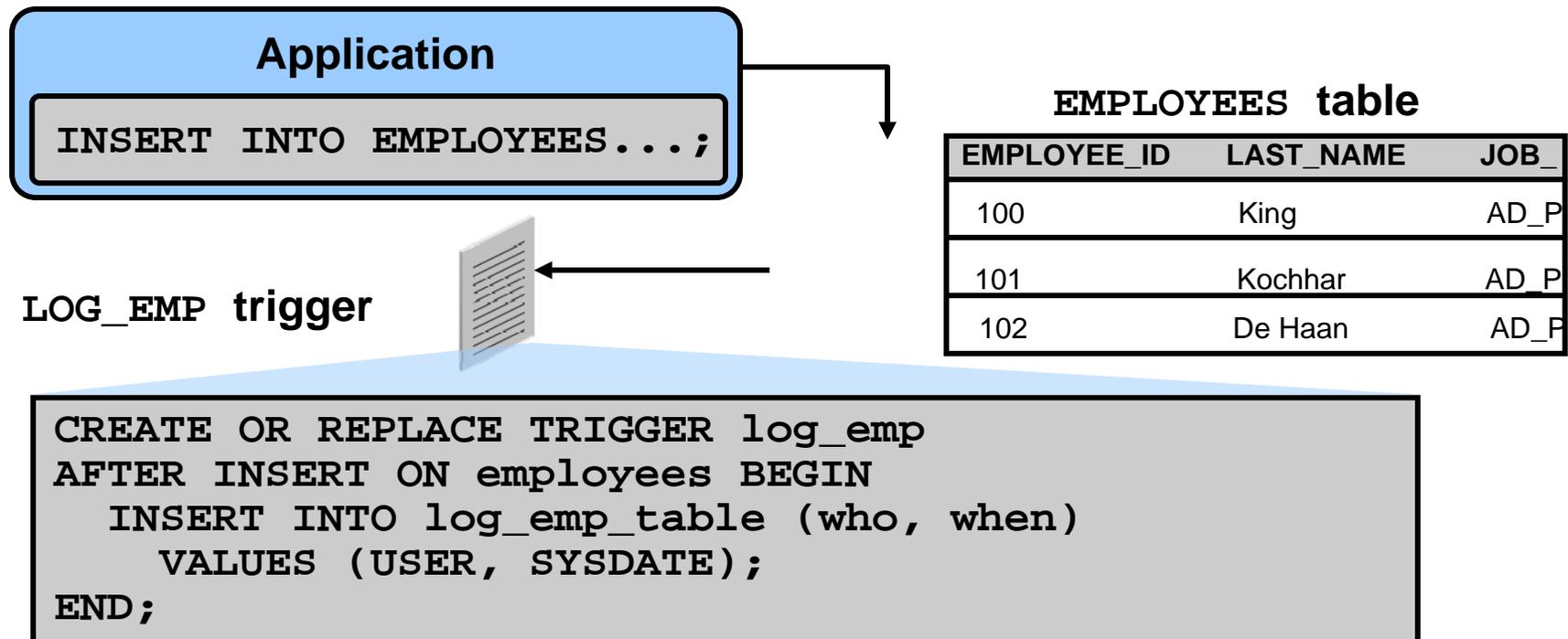
DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500

→ BEFORE statement trigger

400	CONSULTING	2500
-----	------------	------

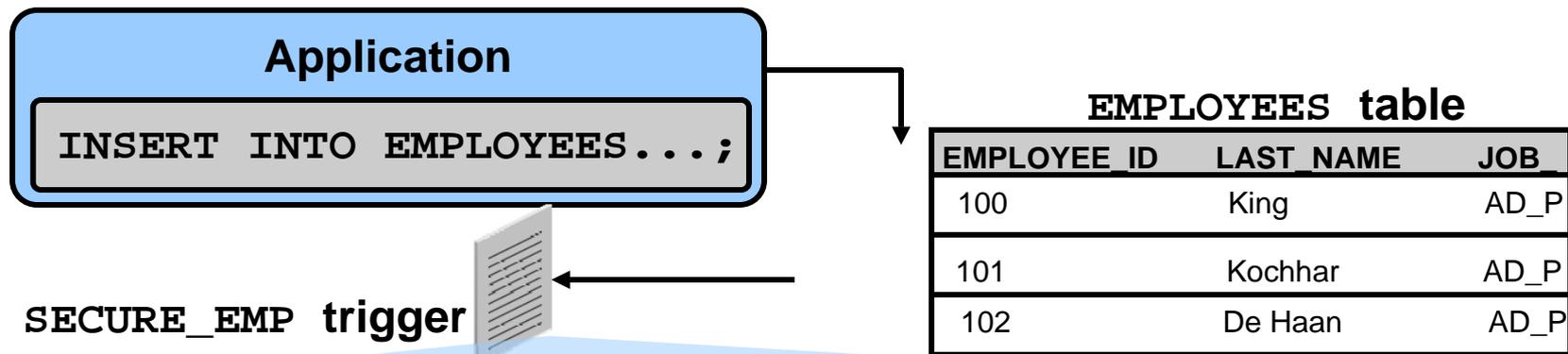
→ AFTER statement trigger

DML Statement Triggers: Example 1



This statement trigger automatically inserts a row into a logging table every time one or more rows are successfully inserted into EMPLOYEES.

Statement Trigger Firing Sequence



```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees
BEGIN
  IF TO_CHAR(SYSDATE, 'DY') IN ('SAT', 'SUN') THEN
    RAISE_APPLICATION_ERROR(-20500, 'You may insert'
      || ' into EMPLOYEES table only during '
      || ' business hours.');
```

```
  END IF;
END;
```

 **Testing SECURE_EMP**

```
INSERT INTO employees (employee_id, last_name,  
    first_name, email, hire_date,  
    job_id, salary, department_id)  
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,  
    'IT_PROG', 4500, 60);
```

```
ORA-20500: You may insert into EMPLOYEES table only during business hours.  
ORA-06512: at "USVA_TEST_SQL01_T01.SECURE_EMP", line 4  
ORA-04088: error during execution of trigger 'USVA_TEST_SQL01_T01.SECURE_EMP'  
2.  VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE, 'IT_PROG', 4500, 60);
```

Creating DML Triggers: Part II



Using Conditional Predicates

The advantage of this is a single trigger with a different action for each triggering DML event.

```
CREATE OR REPLACE TRIGGER secure_emp BEFORE
INSERT OR UPDATE OR DELETE ON employees BEGIN
  IF TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN') THEN
    IF DELETING THEN RAISE_APPLICATION_ERROR(
      -20502,'You may delete from EMPLOYEES table' ||
        'only during business hours.');
```

```
    ELSIF INSERTING THEN RAISE_APPLICATION_ERROR(
      -20500,'You may insert into EMPLOYEES table' ||
        'only during business hours.');
```

```
    ELSIF UPDATING('SALARY') THEN
      RAISE_APPLICATION_ERROR(-20503, 'You may ' ||
        'update SALARY only during business hours.');
```

```
    ELSE RAISE_APPLICATION_ERROR(-20504,'You may' ||
      ' update EMPLOYEES table only during' ||
      ' normal hours.');
```

```
  END IF;
END IF;
END;
```



Creating a DML Row Trigger

```
CREATE OR REPLACE TRIGGER log_deleted_emps
AFTER UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    INSERT INTO log_emp_table (who, when)
        VALUES (USER, SYSDATE);
END;
```

You specify a row trigger using `FOR EACH ROW`.

A row trigger fires once for each row affected by the triggering DML statement.



Row Trigger Firing Sequence

A row trigger fires (executes) once for each row affected by the triggering DML statement, either just **BEFORE** the row is processed or just **AFTER**. If there are five employees in department 50, the row trigger on the previous slide executes five times:

```
UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 50;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
124	Mourgos	50
141	Rajs	50
142	Davies	50
143	Matos	50
144	Vargas	50

→ **BEFORE** row trigger

→ **AFTER** row trigger

...

→ **BEFORE** row trigger

→ **AFTER** row trigger

...



Using OLD and NEW Qualifiers in a Row-Level Trigger

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO audit_emp(user_name, time_stamp, id,
        old_last_name, new_last_name, old_title,
        new_title, old_salary, new_salary)
VALUES (USER, SYSDATE, :OLD.employee_id,
        :OLD.last_name, :NEW.last_name, :OLD.job_id,
        :NEW.job_id, :OLD.salary, :NEW.salary);
END;
```

:OLD and :NEW Qualifiers: Example Using audit_emp_values

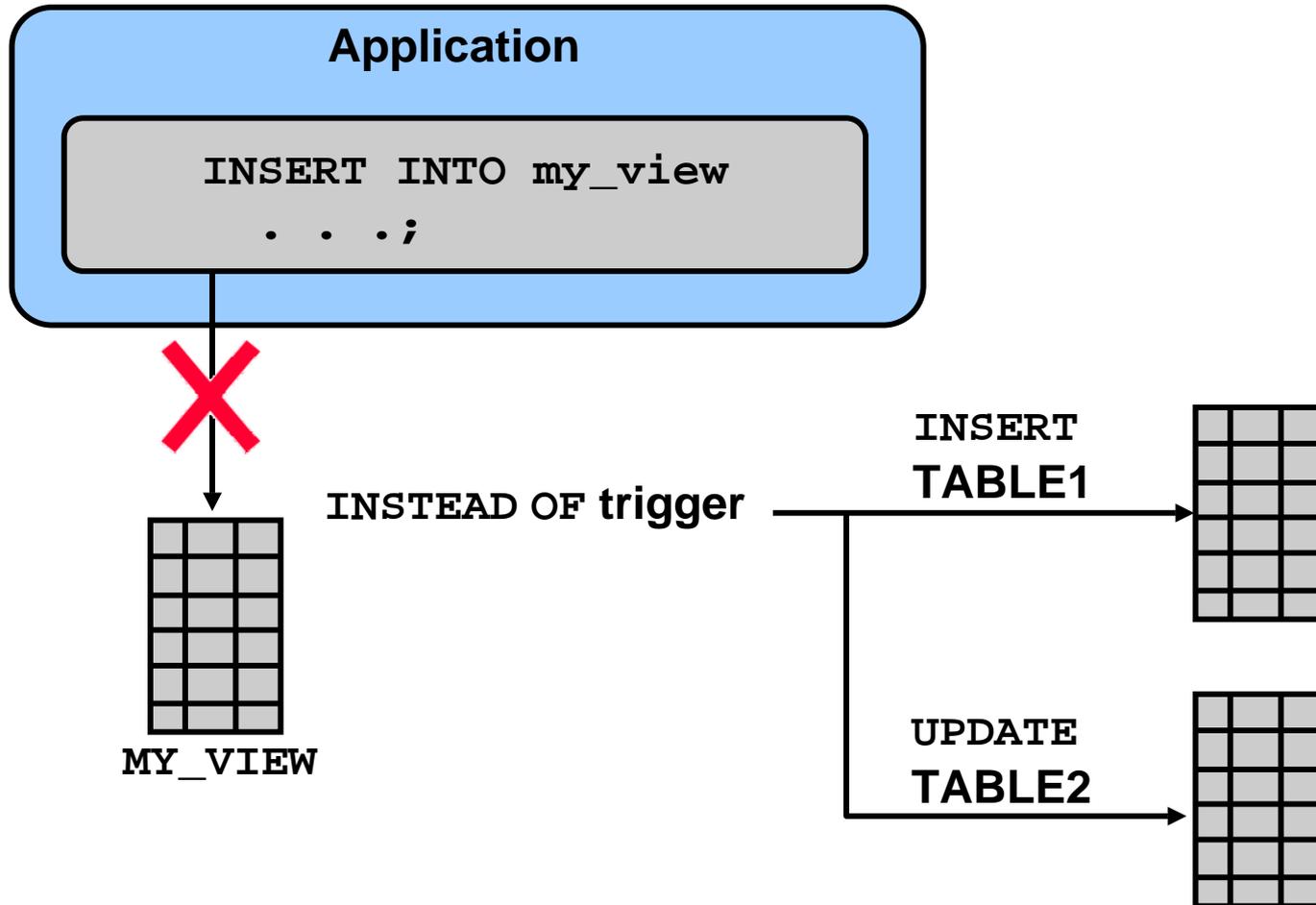
```
INSERT INTO employees
  (employee_id, last_name, job_id, salary, ...)
VALUES (999, 'Temp emp', 'SA_REP', 6000,...);
```

```
UPDATE employees
  SET salary = 7000, last_name = 'Smith'
  WHERE employee_id = 999;
```

```
SELECT user_name, timestamp, ...
FROM audit_emp;
```

USER_NAME	TIME_STAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
APEX_PUBLIC_USER	04-DEC-06	999	Temp emp	Smith	SA_REP	SA_REP	100	2000
APEX_PUBLIC_USER	04-DEC-06	-	-	Temp emp	-	SA_REP	-	1000

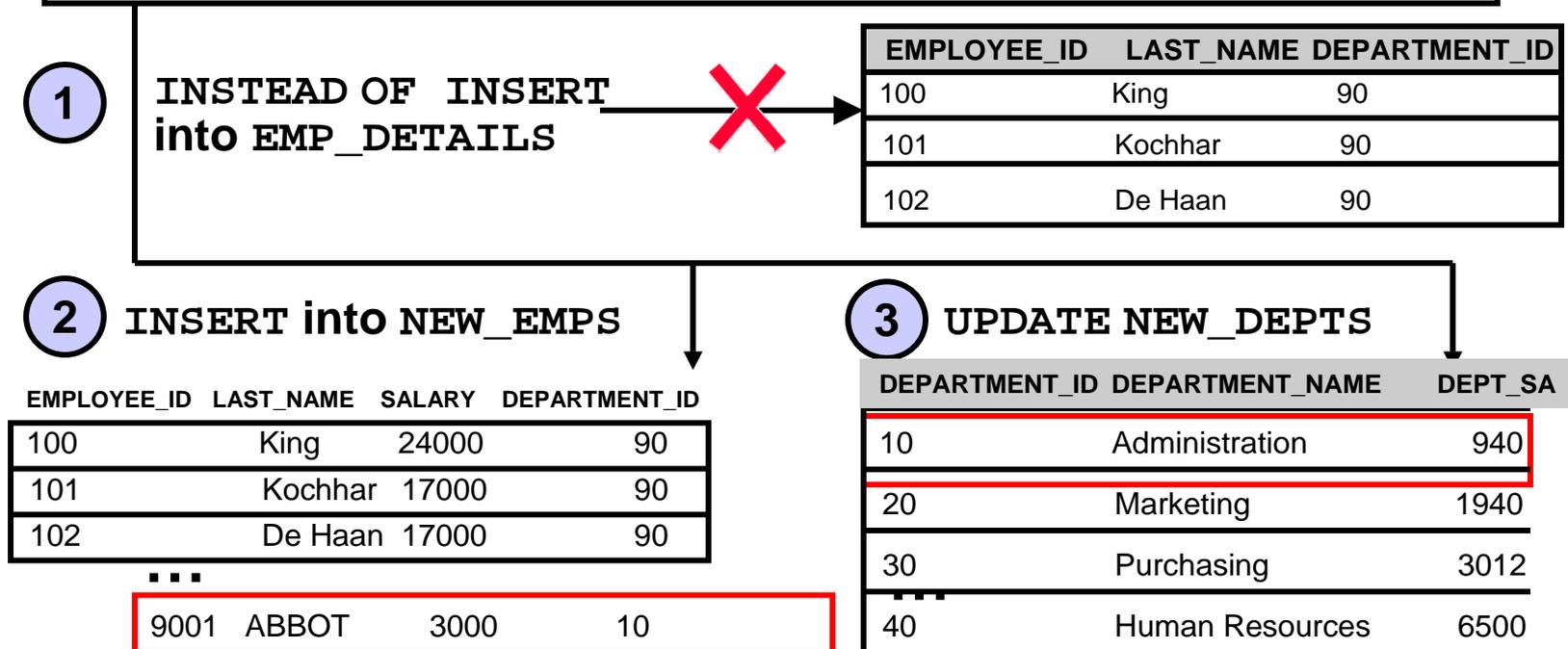
INSTEAD OF Trigger on a VIEW



Creating an INSTEAD OF Trigger

Perform the INSERT into the EMP_DETAILS view that is based on the NEW_EMPS and NEW_DEPTS tables:

```
INSERT INTO emp_details
VALUES (9001,'ABBOTT',3000, 10, 'Administration');
```





Creating an INSTEAD OF Trigger (continued)

Use INSTEAD OF to perform DML on complex views:

```
CREATE TABLE new_emps AS
  SELECT employee_id,last_name,salary,department_id
  FROM employees;

CREATE TABLE new_depts AS
  SELECT d.department_id,d.department_name,
         sum(e.salary) dept_sal
  FROM employees e, departments d
  WHERE e.department_id = d.department_id
  GROUP BY d.department_id,d.department_name;

CREATE VIEW emp_details AS
  SELECT e.employee_id, e.last_name, e.salary,
         e.department_id, d.department_name
  FROM employees e, departments d
  WHERE e.department_id = d.department_id;
```



Creating an INSTEAD OF Trigger on a complex view (continued)

```
CREATE OR REPLACE TRIGGER new_emp_dept
INSTEAD OF INSERT ON emp_details
FOR EACH ROW
BEGIN
    INSERT INTO new_emps VALUES (:NEW.employee_id,
        :NEW.last_name, :NEW.salary, :NEW.department_id);
    UPDATE new_depts
        SET dept_sal = dept_sal + :NEW.salary
        WHERE department_id = :NEW.department_id;
END;
```

INSTEAD OF triggers are always row triggers.