1

Introduction to PL/SQL

Lesson Objectives

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

- Explain the need for PL/SQL
- Explain the benefits of PL/SQL
- Identify the different types of PL/SQL blocks
- Use *i*SQL*Plus as a development environment for PL/SQL
- Output messages in PL/SQL

What Is PL/SQL?

PL/SQL:

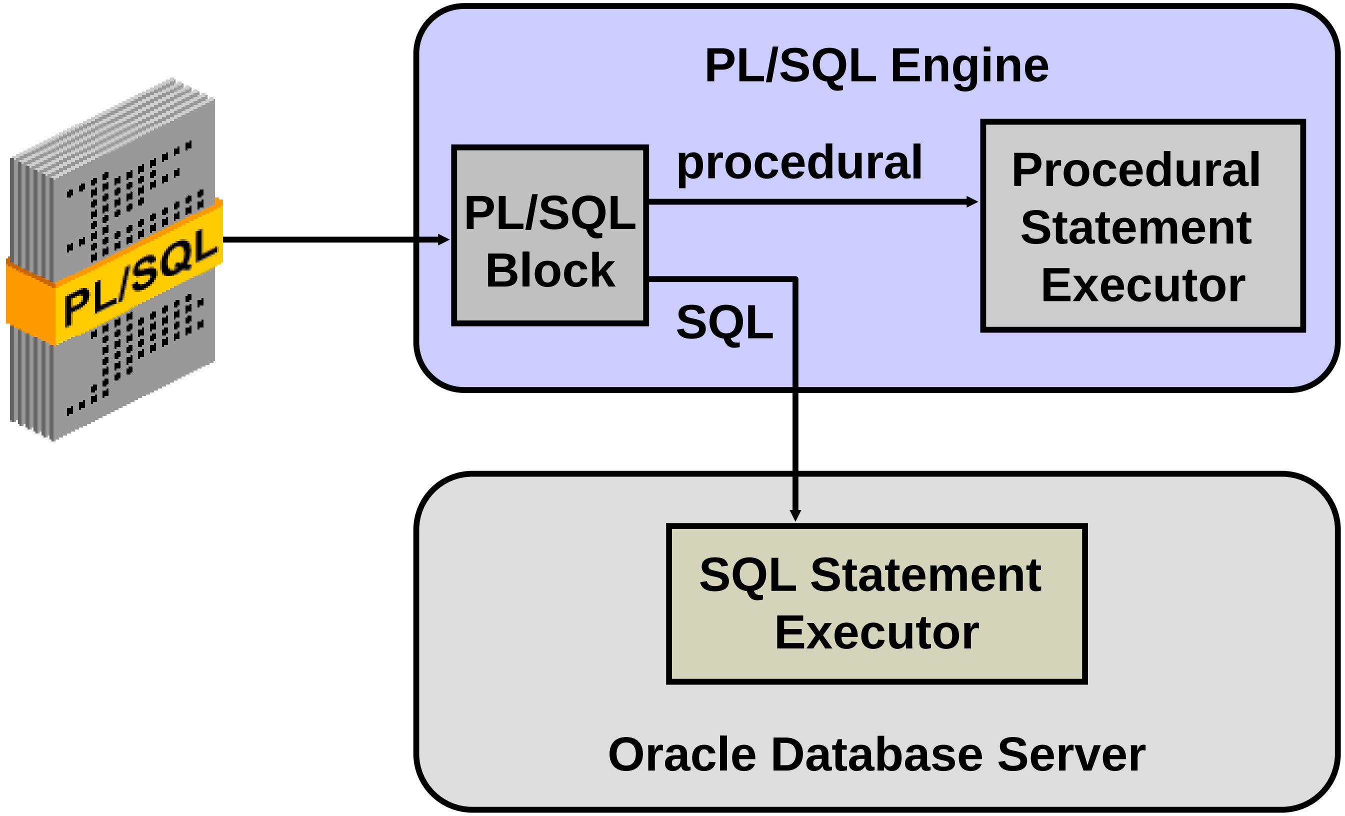
- Stands for Procedural Language extension to SQL
- Is Oracle Corporation's standard data access language for relational databases
- Seamlessly integrates procedural constructs with SQL

About PL/SQL

PL/SQL:

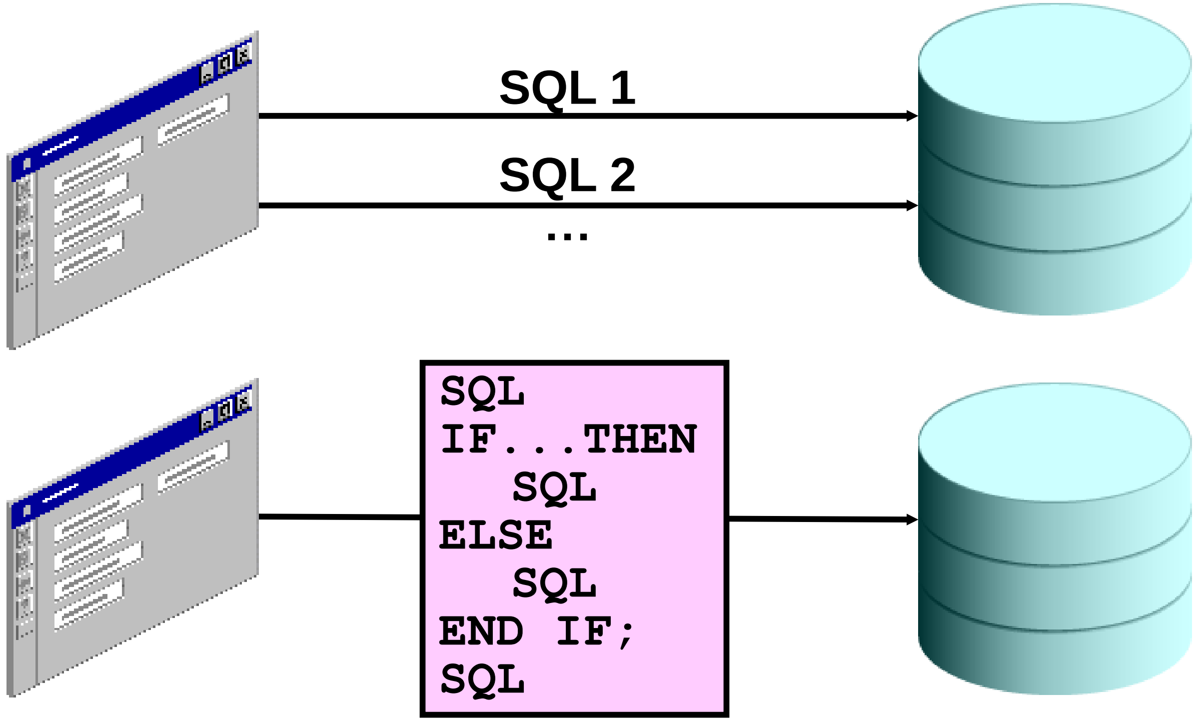
- Provides a block structure for executable units of code. Maintenance of code is made easier with such a well-defined structure.
- Provides procedural constructs such as:
  - Variables, constants, and types
  - Control structures such as conditional statements and loops
  - Reusable program units that are written once and executed many times

PL/SQL Environment



Benefits of PL/SQL

- Integration of procedural constructs with SQL
- Improved performance



Benefits of PL/SQL

- Modularized program development
- Integration with Oracle tools
- Portability
- Exception handling

PL/SQL Block Structure

`DECLARE` **(Optional)**

Variables, cursors, user-defined exceptions

`BEGIN` **(Mandatory)**

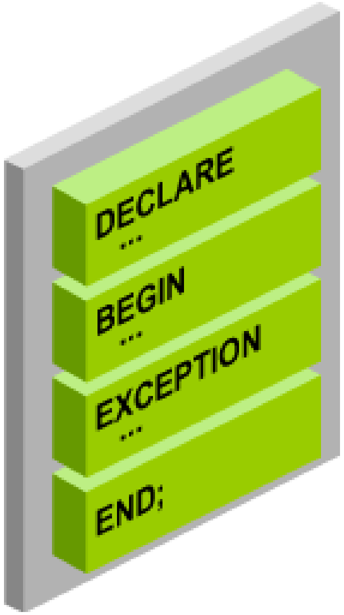
- SQL statements

- PL/SQL statements

`EXCEPTION` **(Optional)**

Actions to perform when errors occur

`END;` **(Mandatory)**



Block Types

Anonymous

```
[DECLARE]


BEGIN
  --statements

[EXCEPTION]

END;
```

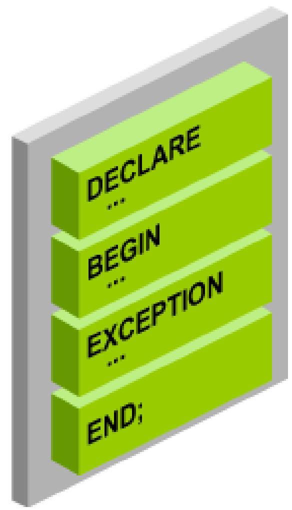
Procedure

```
PROCEDURE name
IS

BEGIN
  --statements

[EXCEPTION]

END;
```

Function

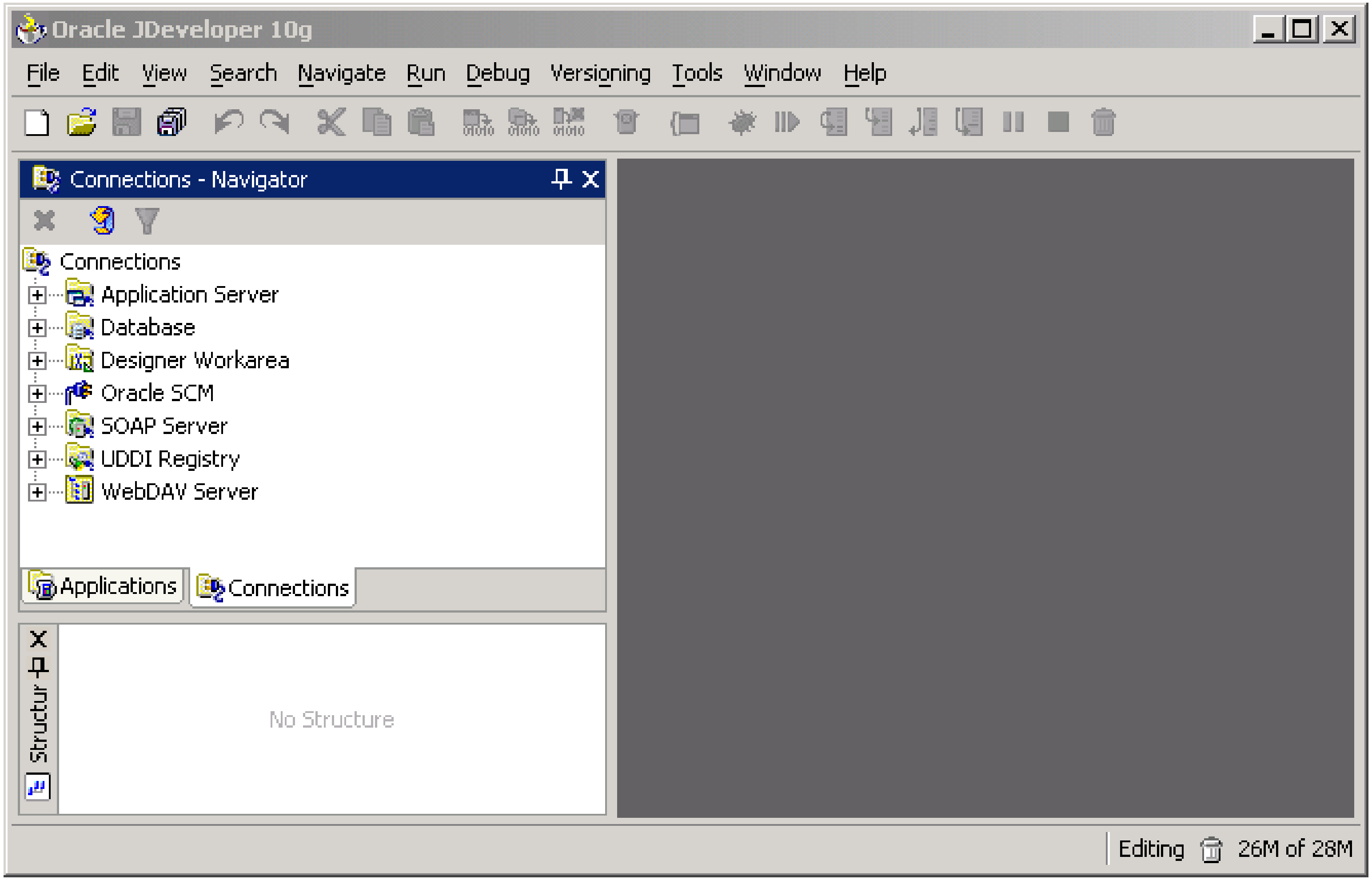
```
FUNCTION name
RETURN datatype
IS
BEGIN
  --statements
  RETURN value;
[EXCEPTION]

END;
```

Program Constructs



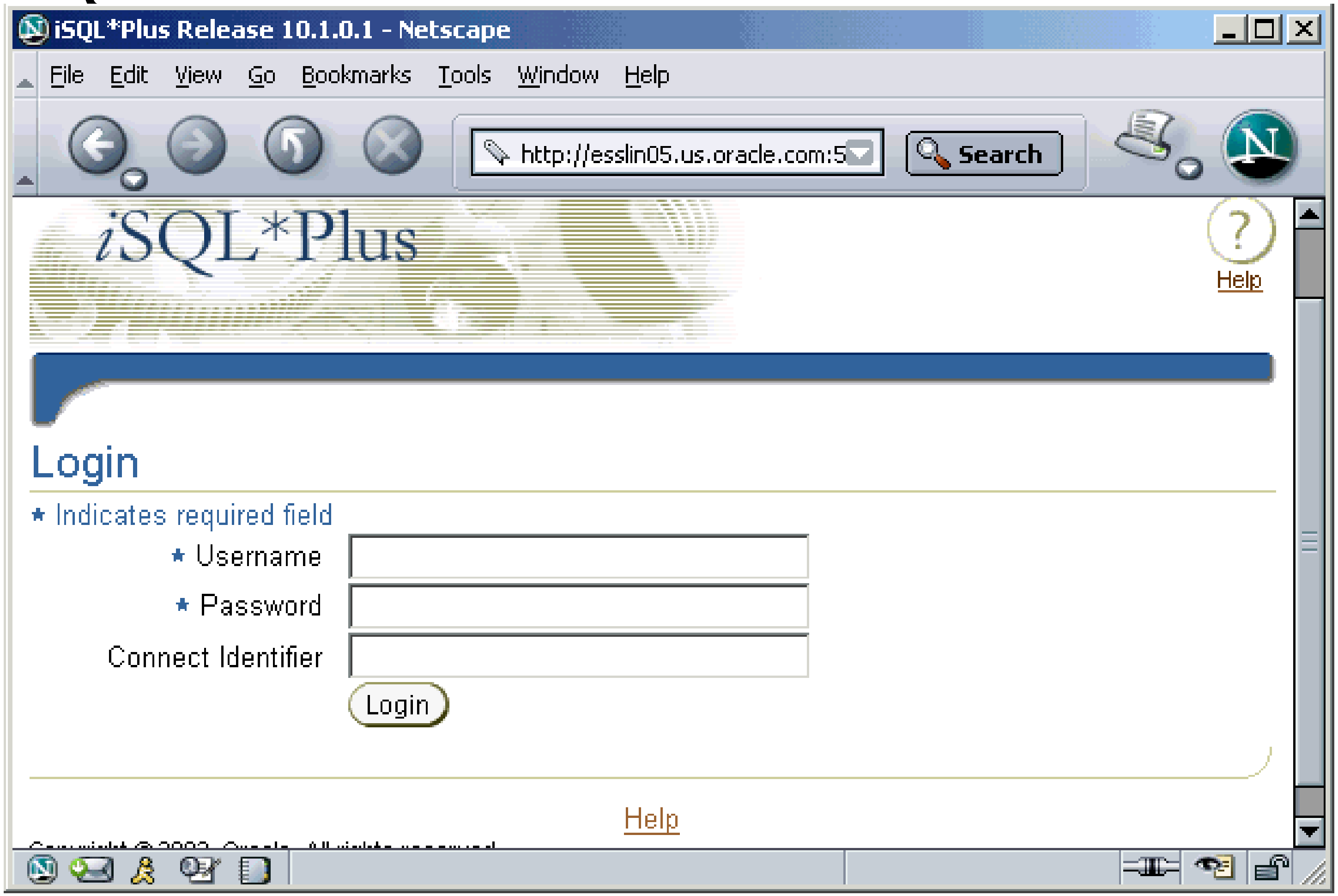
Tools Constructs	Database Server Constructs
Anonymous blocks	Anonymous blocks
Application procedures or functions	Stored procedures or functions
Application packages	Stored packages
Application triggers	Database triggers
Object types	Object types

PL/SQL Programming Environments

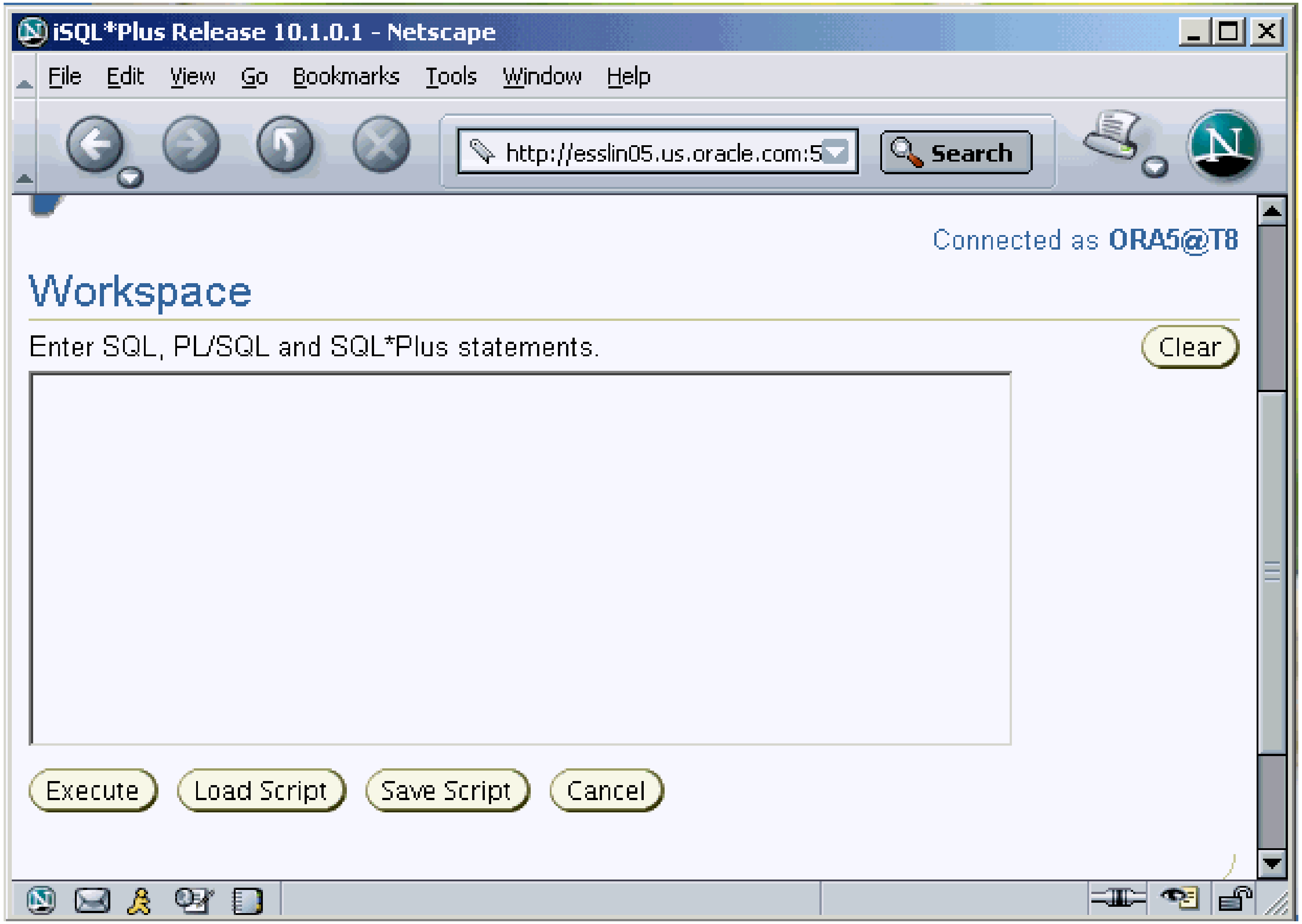


PL/SQL Programming Environments

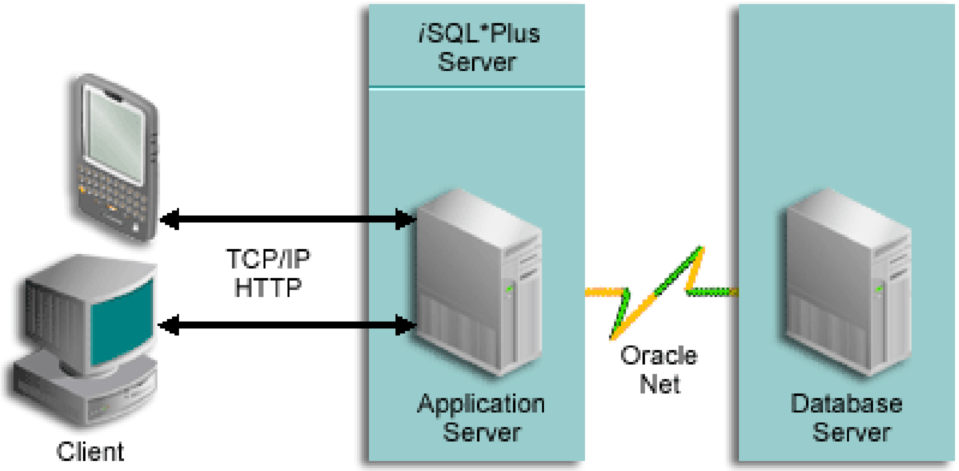
*i*SQL*Plus



PL/SQL Programming Environments

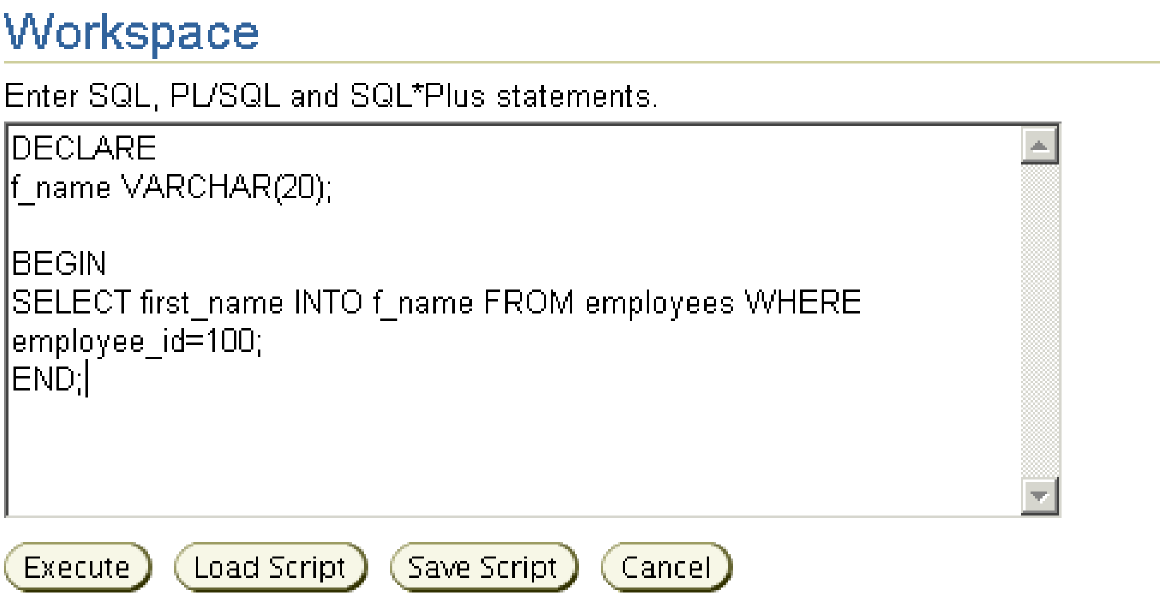


*i*SQL*Plus Architecture



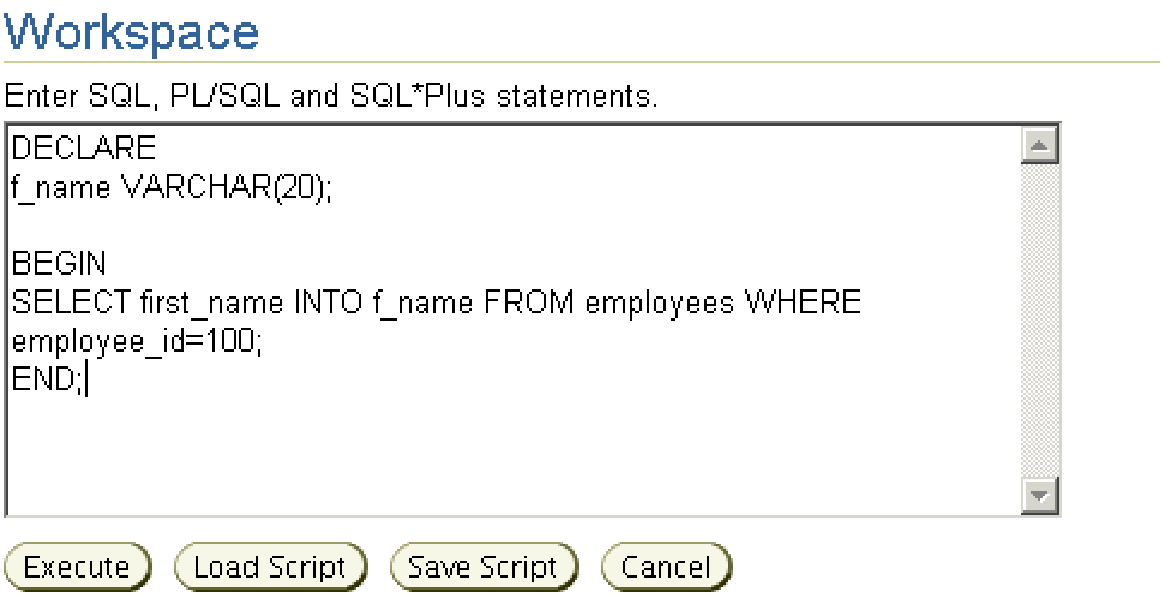
Create an Anonymous Block

Type the anonymous block in the *i*SQL*Plus workspace:



Execute an Anonymous Block

Click the Execute button to execute the anonymous block:



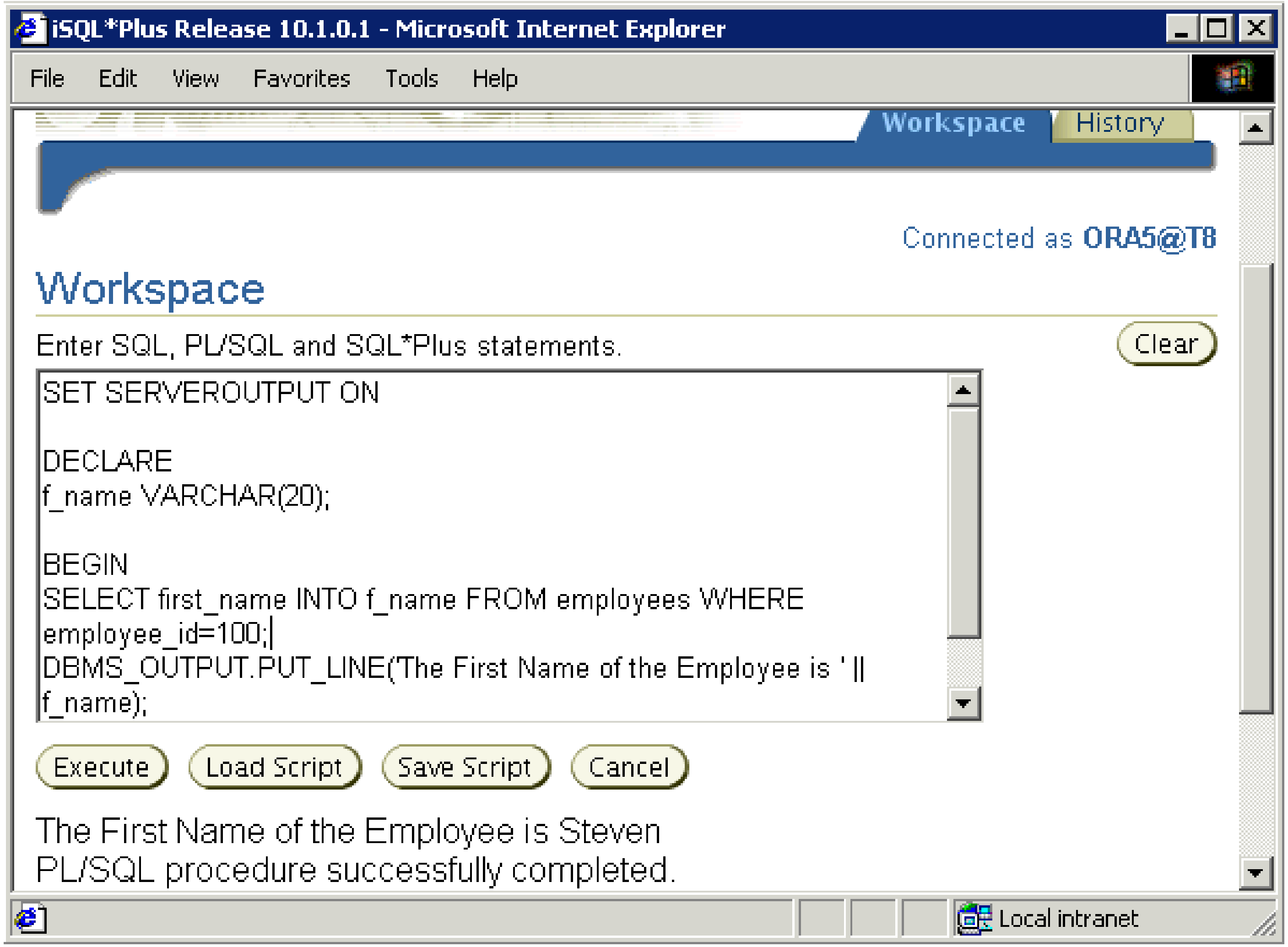
PL\SQL procedure successfully completed.

Test the Output of a PL/SQL Block

- Enable output in *i*SQL*Plus with the command
  `SET SERVEROUTPUT ON`
- Use a predefined Oracle package and its procedure:
  - **`DBMS_OUTPUT.PUT_LINE`**

```
SET SERVEROUTPUT ON
…
DBMS_OUTPUT.PUT_LINE(' The First Name of the
Employee is ' || f_name);
…
```

Test the Output of a PL/SQL Block



Summary

In this lesson, you should have learned how to:

- Integrate SQL statements with PL/SQL program constructs
- Identify the benefits of PL/SQL
- Differentiate different PL/SQL block types
- Use *i*SQL*Plus as the programming environment for PL/SQL
- Output messages in PL/SQL

Practice 1: Overview

This practice covers the following topics:

- **Identifying which PL/SQL blocks execute successfully**
- **Creating and executing a simple PL/SQL block**

1

Declaring PL/SQL Variables

Objectives

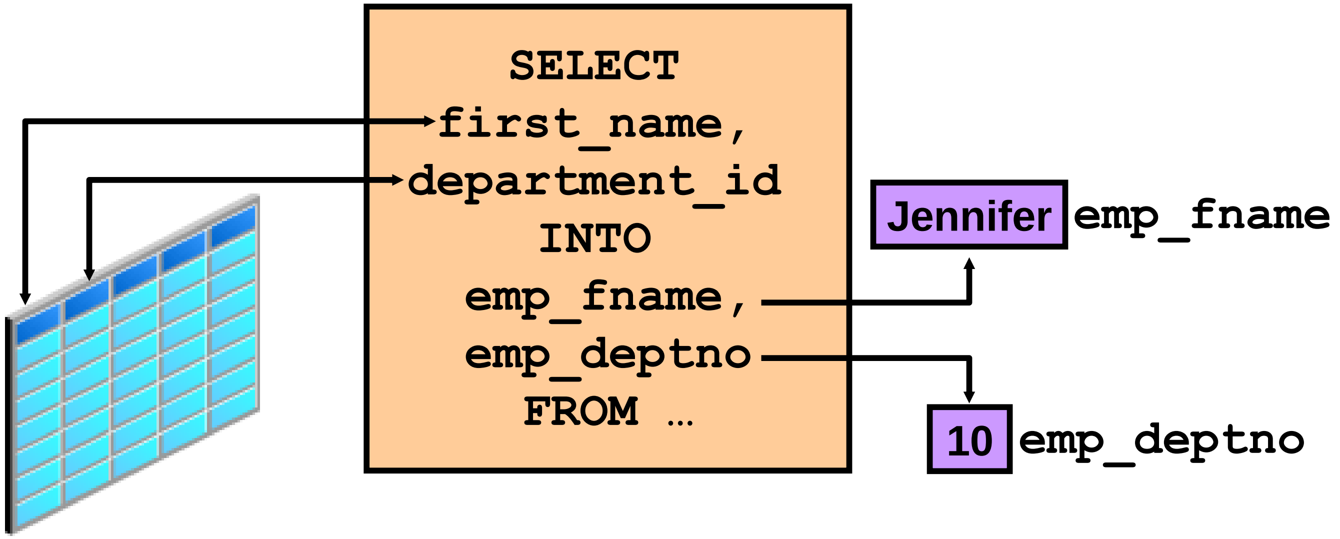
After completing this lesson, you should be able to do the following:

- Identify valid and invalid identifiers
- List the uses of variables
- Declare and initialize variables
- List and describe various data types
- Identify the benefits of using `%TYPE` attribute
- Declare, use, and print bind variables

Use of Variables

Variables can be used for:

- **Temporary storage of data**
- **Manipulation of stored values**
- **Reusability**



Identifiers

Identifiers are used for:

- **Naming a variable**
- **Providing a convention for variable names:**
  - **Must start with a letter**
  - **Can include letters or numbers**
  - **Can include special characters such as dollar sign, underscore, and pound sign**
  - **Must limit the length to 30 characters**
  - **Must not be reserved words**

Handling Variables in PL/SQL

Variables are:

- Declared and initialized in the declarative section
- Used and assigned new values in the executable section
- Passed as parameters to PL/SQL subprograms
- Used to hold the output of a PL/SQL subprogram

Declaring and Initializing PL/SQL Variables

Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]
     [:= | DEFAULT expr];
```

Examples:

```
DECLARE
  emp_hiredate    DATE;
  emp_deptno      NUMBER(2) NOT NULL := 10;
  location        VARCHAR2(13) := 'Atlanta';
  c_comm          CONSTANT NUMBER := 1400;
```

Declaring and Initializing PL/SQL Variables

1

```
SET SERVEROUTPUT ON
DECLARE
  Myname VARCHAR2(20);
BEGIN
  DBMS_OUTPUT.PUT_LINE('My name is: '||Myname);
  Myname := 'John';
  DBMS_OUTPUT.PUT_LINE('My name is: '||Myname);
END;
/
```

2

```
SET SERVEROUTPUT ON
DECLARE
  Myname VARCHAR2(20):= 'John';
BEGIN
  Myname := 'Steven';
  DBMS_OUTPUT.PUT_LINE('My name is: '||Myname);
END;
/
```

ORACLE

Delimiters in String Literals

```
SET SERVEROUTPUT ON
DECLARE
   event VARCHAR2(15);
BEGIN
  event := q'!Father's day!';
  DBMS_OUTPUT.PUT_LINE('3rd Sunday in June is :
   '||event);
  event := q'[Mother's day]';
  DBMS_OUTPUT.PUT_LINE('2nd Sunday in May is :
   '||event);
END;
/
```

```
3rd Sunday in June is : Father's day
2nd Sunday in May is : Mother's day
PL/SQL procedure successfully completed.
```

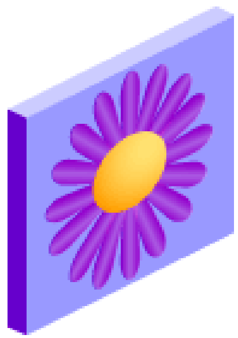
Types of Variables

- **PL/SQL variables:**
  - **Scalar**
  - **Composite**
  - **Reference**
  - **Large objects (LOB)**
- **Non-PL/SQL variables: Bind variables**

Types of Variables

TRUE

25-JAN-01



The soul of the lazy man desires, and has nothing; but the soul of the diligent shall be made rich.

256120.08

Atlanta

Guidelines for Declaring and Initializing PL/SQL Variables

- **Follow naming conventions.**
- **Use meaningful names for variables.**
- **Initialize variables designated as `NOT NULL` and `CONSTANT`.**
- **Initialize variables with the assignment operator (:=) or the `DEFAULT` keyword:**

```
Myname VARCHAR2(20):='John';
```

```
Myname VARCHAR2(20) DEFAULT 'John';
```

- **Declare one identifier per line for better readability and code maintenance.**

Guidelines for Declaring PL/SQL Variables

- **Avoid using column names as identifiers.**

```
DECLARE
  employee_id  NUMBER(6);
BEGIN
  SELECT     employee_id
  INTO       employee_id
  FROM       employees
  WHERE      last_name = 'Kochhar';
END;
/
```

- **Use the `NOT NULL` constraint when the variable must hold a value.**

Scalar Data Types

- Hold a single value
- Have no internal components

TRUE

25-JAN-01

The soul of the lazy man desires, and has nothing; but the soul of the diligent shall be made rich.

256120.08

Atlanta

Base Scalar Data Types

- `CHAR [(maximum_length)]`
- `VARCHAR2 (maximum_length)`
- `LONG`
- `LONG RAW`
- `NUMBER [(precision, scale)]`
- `BINARY_INTEGER`
- `PLS_INTEGER`
- `BOOLEAN`
- `BINARY_FLOAT`
- `BINARY_DOUBLE`

Base Scalar Data Types

- **DATE**
- **TIMESTAMP**
- **TIMESTAMP WITH TIME ZONE**
- **TIMESTAMP WITH LOCAL TIME ZONE**
- **INTERVAL YEAR TO MONTH**
- **INTERVAL DAY TO SECOND**

BINARY_FLOAT and BINARY_DOUBLE

- Represent floating point numbers in IEEE (Institute of Electrical and Electronics Engineers) 754 format
- Offer better interoperability and operational speed
- Store values beyond the values that the data type NUMBER can store
- Offer benefits of closed arithmetic operations and transparent rounding

Declaring Scalar Variables

Examples:

```
DECLARE
  emp_job         VARCHAR2(9);
  count_loop      BINARY_INTEGER := 0;
  dept_total_sal  NUMBER(9,2) := 0;
  orderdate       DATE := SYSDATE + 7;
  c_tax_rate      CONSTANT NUMBER(3,2) := 8.25;
  valid           BOOLEAN NOT NULL := TRUE;
  ...
```

The `%TYPE` Attribute

The `%TYPE` attribute

- Is used to declare a variable according to:
  - A database column definition
  - Another declared variable
- Is prefixed with:
  - The database table and column
  - The name of the declared variable

Declaring Variables with the %TYPE Attribute

Syntax:

```
identifier      table.column_name%TYPE;
```

Examples:

```
...
  emp_lname        employees.last_name%TYPE;
  balance          NUMBER(7,2);
  min_balance      balance%TYPE := 1000;
...
```

Declaring Boolean Variables

- Only the values `TRUE`, `FALSE`, and `NULL` can be assigned to a Boolean variable.
- Conditional expressions use logical operators `AND`, `OR`, and unary operator `NOT` to check the variable values.
- The variables always yield `TRUE`, `FALSE`, or `NULL`.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

Bind Variables

Bind variables are:

- Created in the environment
- Also called host variables
- Created with the `VARIABLE` keyword
- Used in SQL statements and PL/SQL blocks
- Accessed even after the PL/SQL block is executed
- Referenced with a preceding colon

Printing Bind Variables

Example:

```
VARIABLE emp_salary NUMBER
BEGIN
   SELECT salary  INTO :emp_salary
   FROM  employees WHERE employee_id = 178;
END;
/
PRINT emp_salary
SELECT first_name, last_name FROM employees
WHERE salary=:emp_salary;
```

Printing Bind Variables

Example:

```
VARIABLE emp_salary NUMBER
SET AUTOPRINT ON
BEGIN
   SELECT salary  INTO :emp_salary
   FROM  employees WHERE employee_id = 178;
END;
/
```

Substitution Variables

- Are used to get user input at run time
- Are referenced within a PL/SQL block with a preceding ampersand
- Are used to avoid hard coding values that can be obtained at run time

```
VARIABLE emp_salary NUMBER
SET AUTOPRINT ON
DECLARE
  empno NUMBER(6):=&empno;
BEGIN
  SELECT salary  INTO :emp_salary
  FROM  employees WHERE employee_id = empno;
END;
/
```

Substitution Variables

Input Required

Cancel Continue

Enter value for empno: 100

Cancel Continue

1

```
old 2: empno NUMBER(6):=&empno;
new 2: empno NUMBER(6):=100;
PL/SQL procedure successfully completed.
```

EMP_SALARY
24000

2

```
PL/SQL procedure successfully completed.
```

EMP_SALARY
24000

3

Prompt for Substitution Variables

```
SET VERIFY OFF
VARIABLE emp_salary NUMBER
ACCEPT empno PROMPT 'Please enter a valid employee
number: '
SET AUTOPRINT ON
DECLARE
  empno NUMBER(6):= &empno;
BEGIN
  SELECT salary  INTO :emp_salary  FROM  employees
  WHERE employee_id = empno;
END;
/
```

Input Required

Cancel Continue

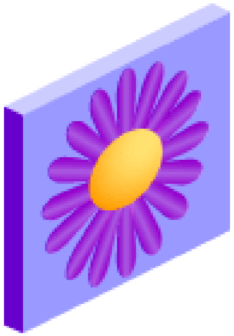
Please enter a valid employee number: 100

Using DEFINE for User Variable

Example:

```
SET VERIFY OFF
DEFINE lname= Urman
DECLARE
  fname VARCHAR2(25);
BEGIN
  SELECT first_name INTO fname FROM employees
  WHERE last_name='&lname';
END;
/
```

Composite Data Types

TRUE	23-DEC-98	ATLANTA	

PL/SQL table structure

1	SMITH
2	JONES
3	NANCY
4	TIM

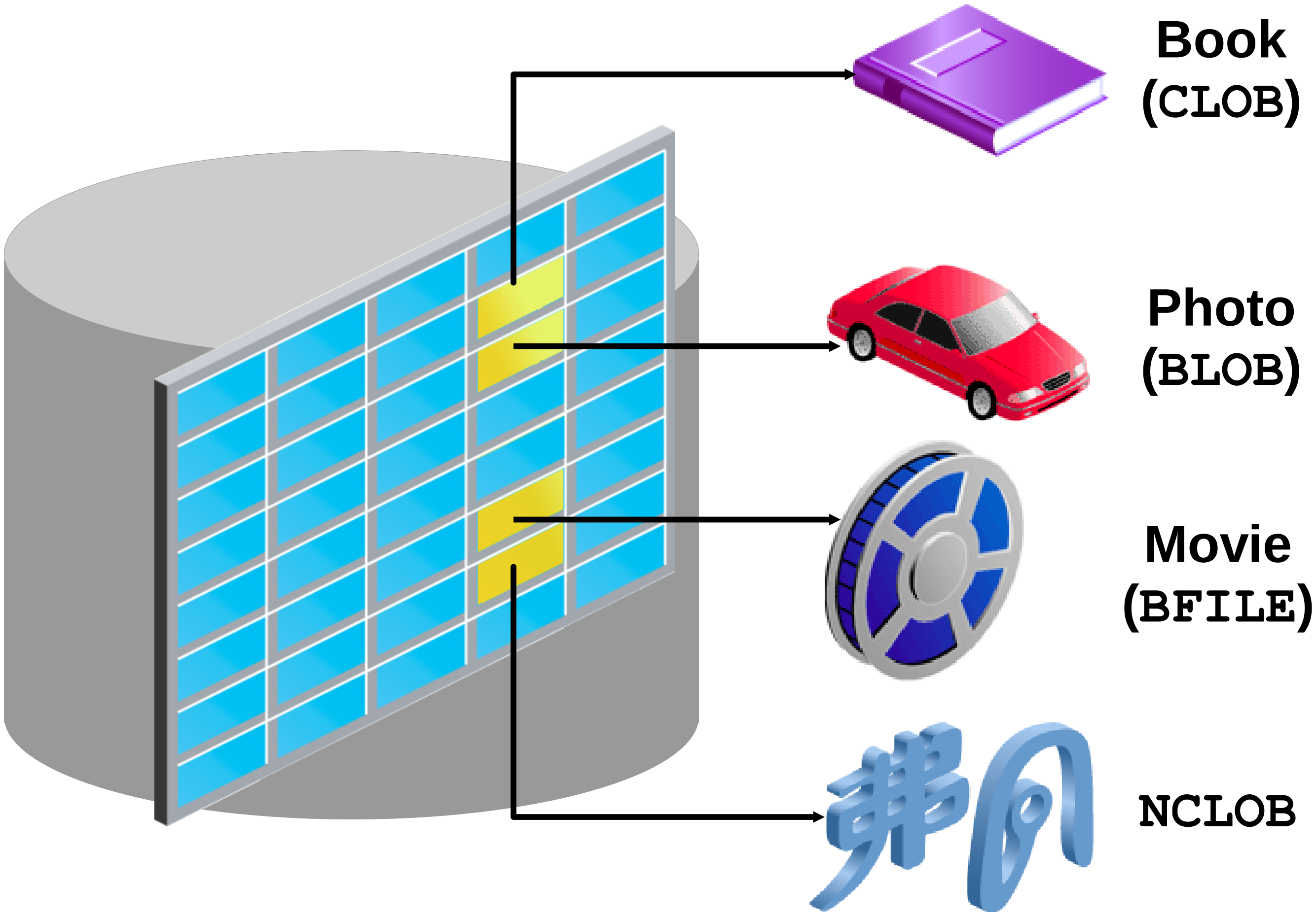
PLS_INTEGER VARCHAR2

PL/SQL table structure

1	5000
2	2345
3	12
4	3456

PLS_INTEGER NUMBER

LOB Data Type Variables



Summary

In this lesson, you should have learned how to:

- Identify valid and invalid identifiers
- Declare variables in the declarative section of a PL/SQL block
- Initialize variables and utilize them in the executable section
- Differentiate between scalar and composite data types
- Use the `%TYPE` attribute
- Make use of bind variables

ORACLE

Practice 2: Overview

This practice covers the following topics:

- **Determining valid identifiers**
- **Determining valid variable declarations**
- **Declaring variables within an anonymous block**
- **Using the `%TYPE` attribute to declare variables**
- **Declaring and printing a bind variable**
- **Executing a PL/SQL block**

1

Creating Stored Procedures

Objectives

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

- Describe and create a procedure
- Create procedures with parameters
- Differentiate between formal and actual parameters
- Use different parameter-passing modes
- Invoke a procedure
- Handle exceptions in procedures
- Remove a procedure

What Is a Procedure?

A procedure:

- Is a type of subprogram that performs an action
- Can be stored in the database as a schema object
- Promotes reusability and maintainability

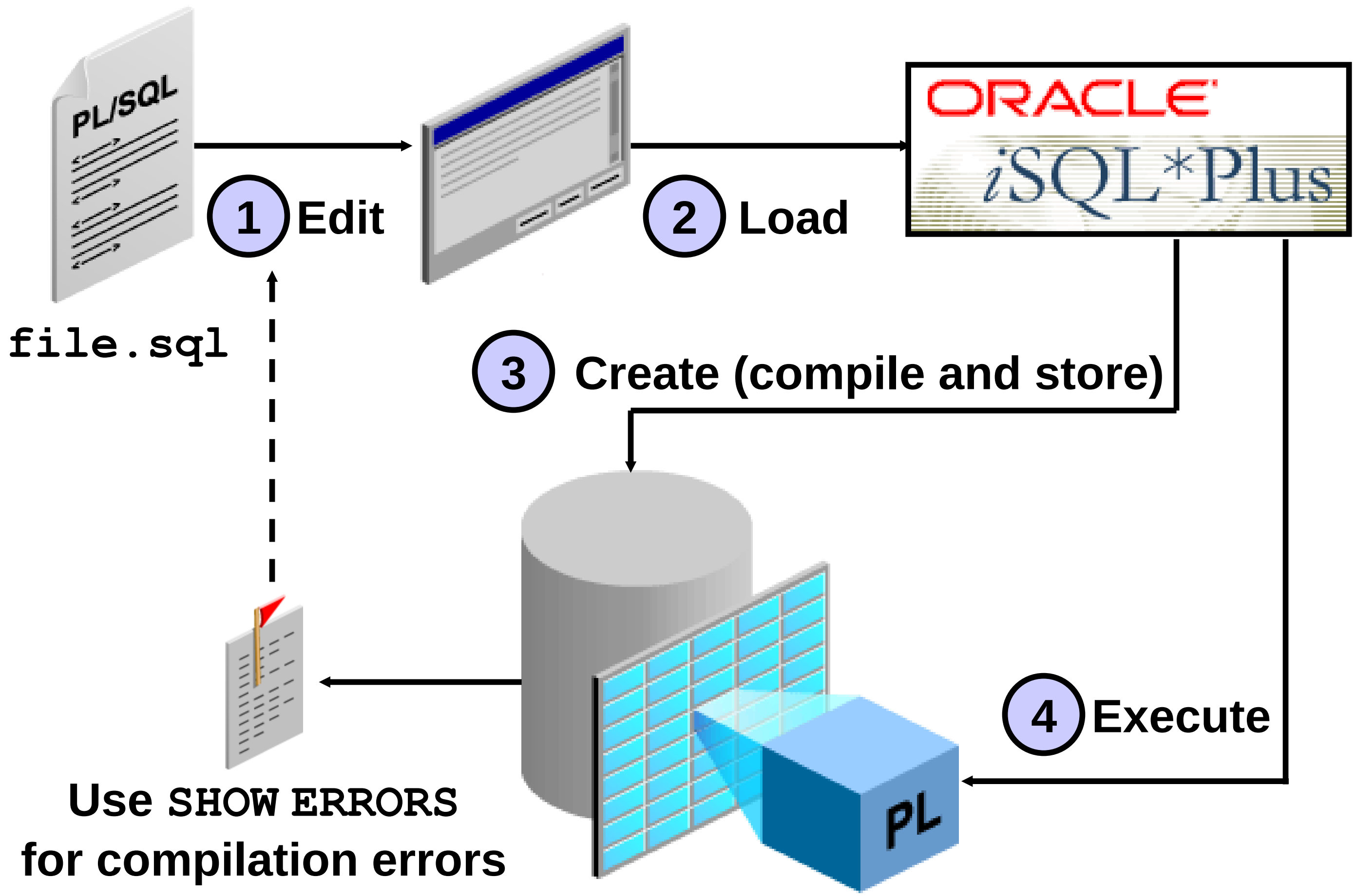
Syntax for Creating Procedures

- Use `CREATE PROCEDURE` followed by the name, optional parameters, and keyword `IS` or `AS`.
- Add the `OR REPLACE` option to overwrite an existing procedure.
- Write a PL/SQL block containing local variables, a `BEGIN`, and an `END` (or `END` *`procedure_name`*).

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode] datatype1,
   parameter2 [mode] datatype2, ...)]
IS|AS
  [local_variable_declarations; …]
BEGIN
  -- actions;
END [procedure_name];
```

PL/SQL Block

Developing Procedures



What Are Parameters?

Parameters:

- **Are declared after the subprogram name in the PL/SQL header**
- **Pass or communicate data between the caller and the subprogram**
- **Are used like local variables but are dependent on their parameter-passing mode:**
  - **An `IN` parameter (the default) provides values for a subprogram to process.**
  - **An `OUT` parameter returns a value to the caller.**
  - **An `IN OUT` parameter supplies an input value, which may be returned (output) as a modified value.**

Formal and Actual Parameters

- **Formal parameters: Local variables declared in the parameter list of a subprogram specification**

  Example:

```
CREATE PROCEDURE raise_sal(id NUMBER,sal NUMBER) IS
BEGIN ...
END raise_sal;
```

- **Actual parameters: Literal values, variables, or expressions used in the parameter list of the called subprogram**

  Example:

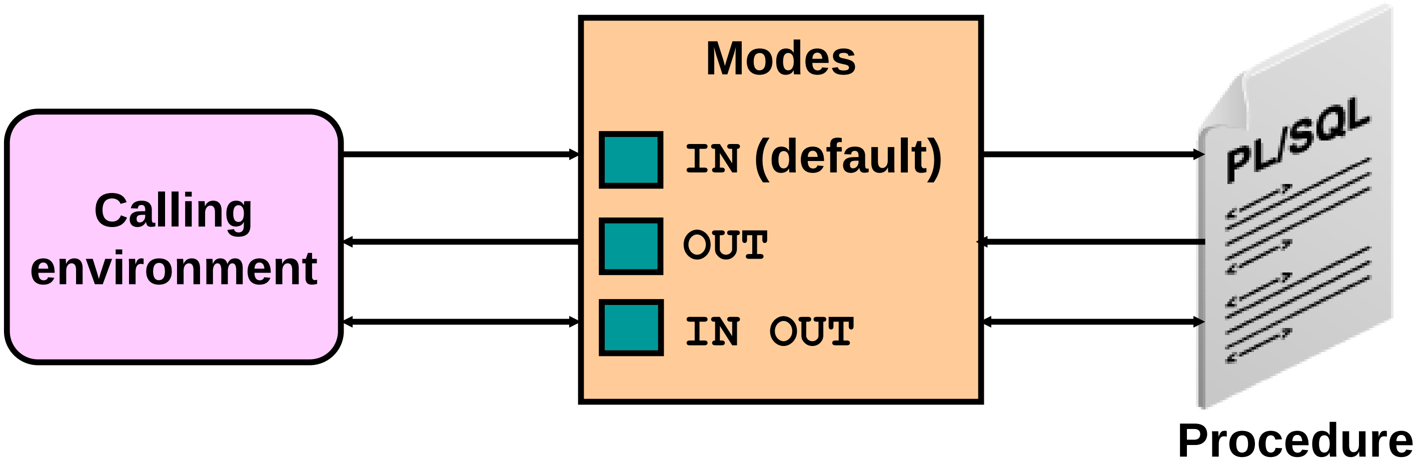
```
emp_id := 100;
raise_sal(emp_id, 2000)
```

ORACLE

Procedural Parameter Modes

- Parameter modes are specified in the formal parameter declaration, after the parameter name and before its data type.
- The `IN` mode is the default if no mode is specified.

```
CREATE PROCEDURE procedure(param [mode] datatype)
...
```



Using IN Parameters: Example

```
CREATE OR REPLACE PROCEDURE raise_salary
  (id      IN employees.employee_id%TYPE,
   percent IN NUMBER)
IS
BEGIN
  UPDATE employees
  SET    salary = salary * (1 + percent/100)
  WHERE  employee_id = id;
END raise_salary;
/
```

```
EXECUTE raise_salary(176,10)
```

Using OUT Parameters: Example

```
CREATE OR REPLACE PROCEDURE query_emp
 (id     IN  employees.employee_id%TYPE,
  name   OUT employees.last_name%TYPE,
  salary OUT employees.salary%TYPE) IS
BEGIN
  SELECT   last_name, salary INTO name, salary
   FROM    employees
   WHERE   employee_id = id;
END query_emp;
```

```
DECLARE
  emp_name employees.last_name%TYPE;
  emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, emp_name, emp_sal); ...
END;
```

Viewing OUT Parameters with *i*SQL*Plus

- Use PL/SQL variables that are printed with calls to the DBMS_OUTPUT.PUT_LINE procedure.

```
SET SERVEROUTPUT ON
DECLARE
  emp_name employees.last_name%TYPE;
  emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, emp_name, emp_sal);
  DBMS_OUTPUT.PUT_LINE('Name: ' || emp_name);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || emp_sal);
END;
```

- Use *i*SQL*Plus host variables, execute QUERY_EMP using host variables, and print the host variables.

```
VARIABLE name VARCHAR2(25)
VARIABLE sal  NUMBER
EXECUTE query_emp(171, :name, :sal)
PRINT name sal
```

Calling PL/SQL Using Host Variables

A host variable (also known as a bind or a global variable):

- Is declared and exists externally to the PL/SQL subprogram. A host variable can be created in:
  - *i*SQL*Plus by using the `VARIABLE` command
  - Oracle Forms internal and UI variables
  - Java variables
- Is preceded by a colon (`:`) when referenced in PL/SQL code
- Can be referenced in an anonymous block but not in a stored subprogram
- Provides a value to a PL/SQL block and receives a value from a PL/SQL block

ORACLE

Using `IN OUT` Parameters: Example

Calling environment

phone_no (before the call) **phone_no (after the call)**

'8006330575' **'(800)633-0575'**

```
CREATE OR REPLACE PROCEDURE format_phone
   (phone_no IN OUT VARCHAR2) IS
BEGIN
  phone_no := '(' || SUBSTR(phone_no,1,3) ||
              ')' || SUBSTR(phone_no,4,3) ||
              '-' || SUBSTR(phone_no,7);
END format_phone;
/
```

Syntax for Passing Parameters

- **Positional:**
  - Lists the actual parameters in the same order as the formal parameters
- **Named:**
  - Lists the actual parameters in arbitrary order and uses the association operator (=>) to associate a named formal parameter with its actual parameter
- **Combination:**
  - Lists some of the actual parameters as positional and some as named

Parameter Passing: Examples

```
CREATE OR REPLACE PROCEDURE add_dept(
  name IN departments.department_name%TYPE,
  loc  IN departments.location_id%TYPE) IS
BEGIN
  INSERT INTO departments(department_id,
              department_name, location_id)
  VALUES (departments_seq.NEXTVAL, name, loc);
END add_dept;
/
```

- Passing by positional notation

```
EXECUTE add_dept ('TRAINING', 2500)
```

- Passing by named notation

```
EXECUTE add_dept (loc=>2400, name=>'EDUCATION')
```

Using the DEFAULT Option for Parameters

- Defines default values for parameters:

```
CREATE OR REPLACE PROCEDURE add_dept(
 name departments.department_name%TYPE:='Unknown',
 loc  departments.location_id%TYPE DEFAULT 1700)
IS
BEGIN
  INSERT INTO departments (...)
  VALUES (departments_seq.NEXTVAL, name, loc);
END add_dept;
```

- Provides flexibility by combining the positional and named parameter-passing syntax:

```
EXECUTE add_dept
EXECUTE add_dept ('ADVERTISING', loc => 1200)
EXECUTE add_dept (loc => 1200)
```

Summary of Parameter Modes

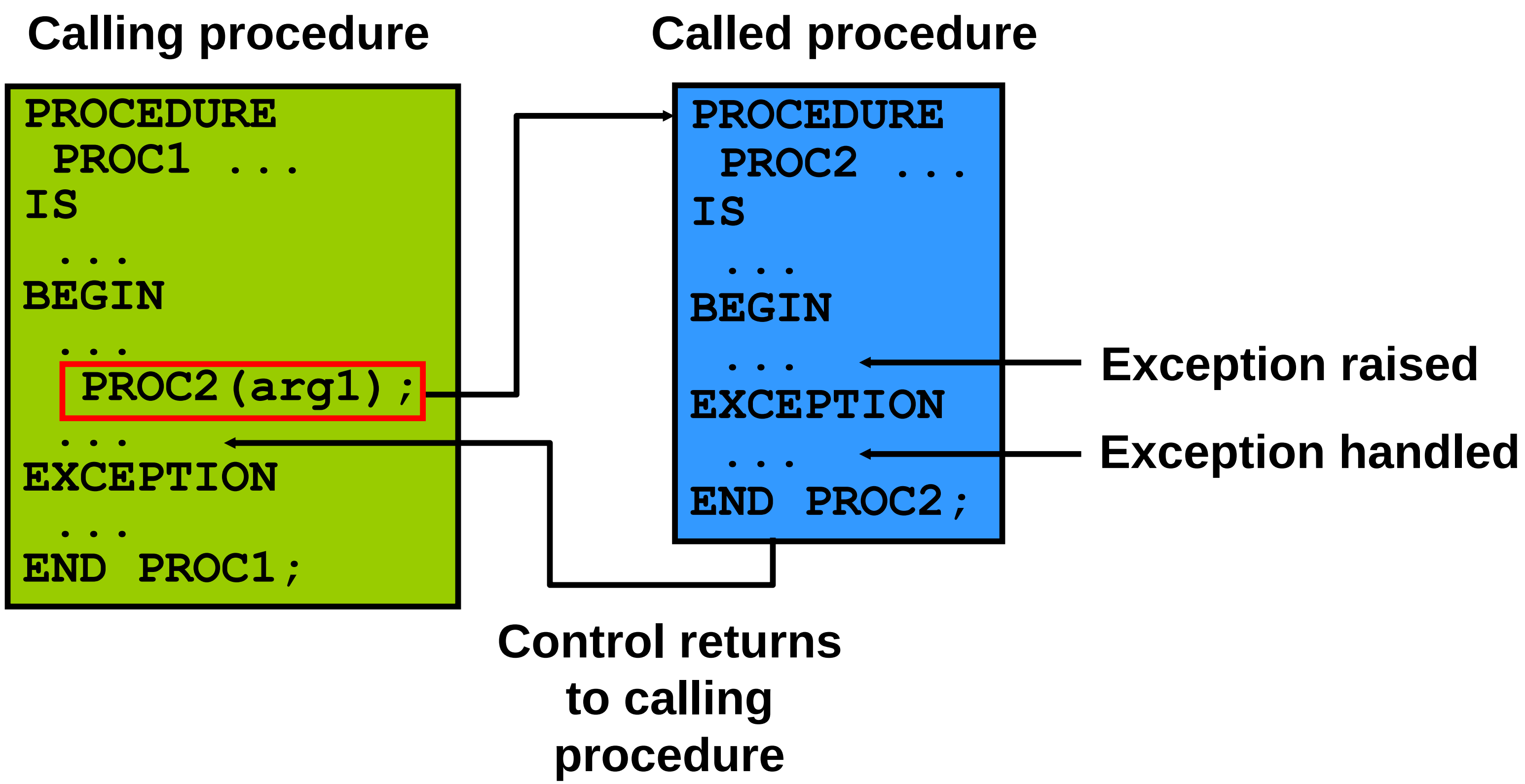
IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Returned to calling environment	Passed into subprogram; returned to calling environment
Formal parameter acts as a constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, expression, constant, or initialized variable	Must be a variable	Must be a variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value

Invoking Procedures

- You can invoke parameters by:
  - Using anonymous blocks
  - Using another procedure, as in the following:

```
CREATE OR REPLACE PROCEDURE process_employees
IS
   CURSOR emp_cursor IS
    SELECT employee_id
    FROM   employees;
BEGIN
   FOR emp_rec IN emp_cursor
   LOOP
     raise_salary(emp_rec.employee_id, 10);
   END LOOP;
   COMMIT;
END process_employees;
/
```

Handled Exceptions

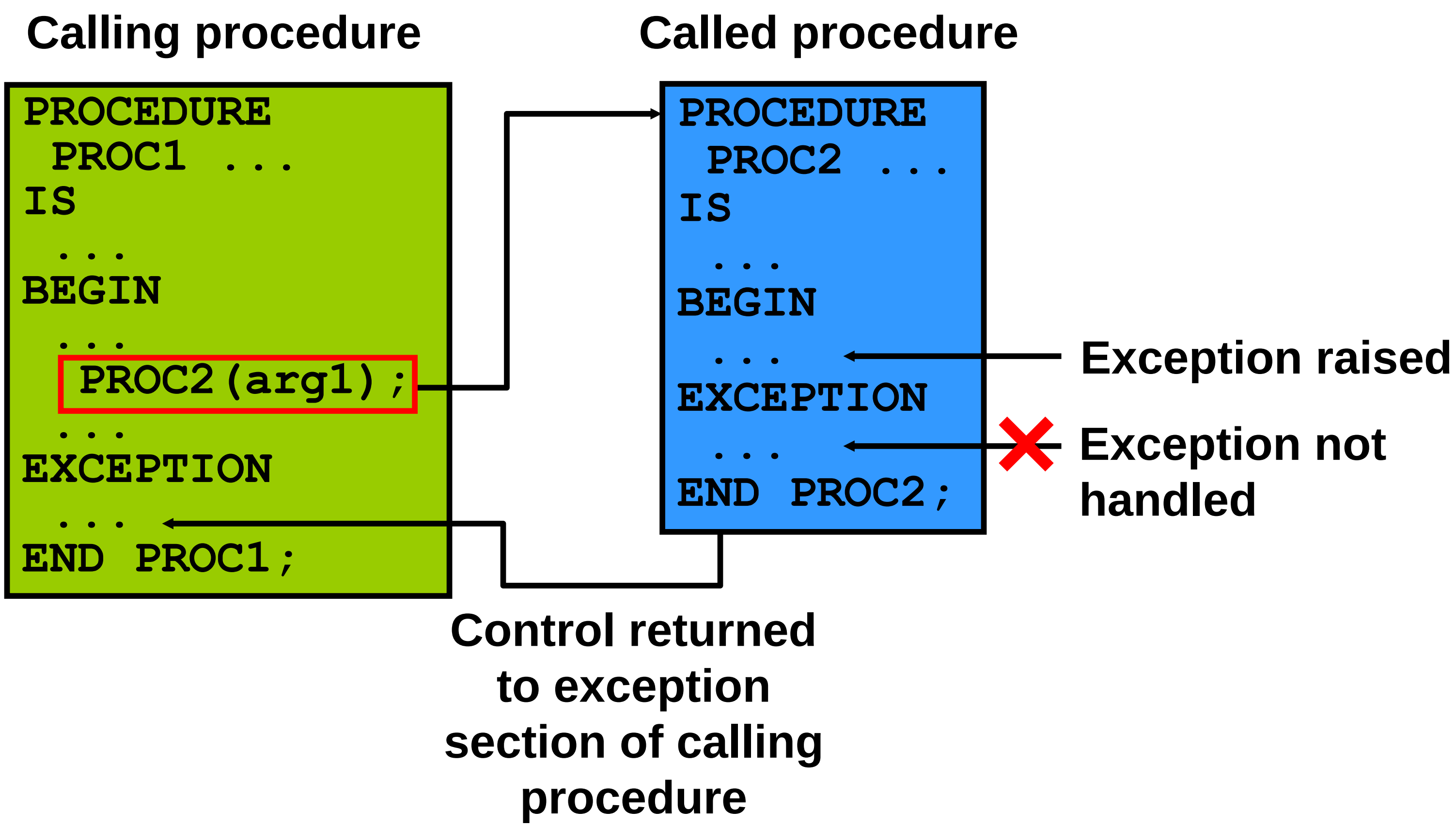


Handled Exceptions: Example

```
CREATE PROCEDURE add_department(
    name VARCHAR2, mgr NUMBER, loc NUMBER) IS
BEGIN
  INSERT INTO DEPARTMENTS (department_id,
    department_name, manager_id, location_id)
  VALUES (DEPARTMENTS_SEQ.NEXTVAL, name, mgr, loc);
  DBMS_OUTPUT.PUT_LINE('Added Dept: '||name);
EXCEPTION
 WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE('Err: adding dept: '||name);
END;
```

```
CREATE PROCEDURE create_departments IS
BEGIN
  add_department('Media', 100, 1800);
  add_department('Editing', 99, 1800);
  add_department('Advertising', 101, 1800);
END;
```

Exceptions Not Handled



Exceptions Not Handled: Example

```
CREATE PROCEDURE add_department_noex(
    name VARCHAR2, mgr NUMBER, loc NUMBER) IS
BEGIN
  INSERT INTO DEPARTMENTS (department_id,
    department_name, manager_id, location_id)
  VALUES (DEPARTMENTS_SEQ.NEXTVAL, name, mgr, loc);
  DBMS_OUTPUT.PUT_LINE('Added Dept: '||name);
END;
```

```
CREATE PROCEDURE create_departments_noex IS
BEGIN
  add_department_noex('Media', 100, 1800);
  add_department_noex('Editing', 99, 1800);
  add_department_noex('Advertising', 101, 1800);
END;
```

Removing Procedures

You can remove a procedure that is stored in the database.

- **Syntax:**

```
DROP PROCEDURE procedure_name
```

- **Example:**

```
DROP PROCEDURE raise_salary;
```

Viewing Procedures in the Data Dictionary

Information for PL/SQL procedures is saved in the following data dictionary views:

- View source code in the `USER_SOURCE` table to view the subprograms that you own, or the `ALL_SOURCE` table for procedures that are owned by others who have granted you the `EXECUTE` privilege.

```
SELECT text
FROM   user_source
WHERE  name='ADD_DEPARTMENT' and type='PROCEDURE'
ORDER BY line;
```

- View the names of procedures in `USER_OBJECTS`.

```
SELECT object_name
FROM   user_objects
WHERE  object_type = 'PROCEDURE';
```

ORACLE

Benefits of Subprograms

- Easy maintenance
- Improved data security and integrity
- Improved performance
- Improved code clarity

Summary

In this lesson, you should have learned how to:

- Write a procedure to perform a task or an action
- Create, compile, and save procedures in the database by using the `CREATE PROCEDURE` SQL command
- Use parameters to pass data from the calling environment to the procedure using three different parameter modes: `IN` (the default), `OUT`, and `IN OUT`
- Recognize the effect of handling and not handling exceptions on transactions and calling procedures

Summary

- Remove procedures from the database by using the `DROP PROCEDURE` SQL command
- Modularize your application code by using procedures as building blocks

Practice 1: Overview

This practice covers the following topics:

- Creating stored procedures to:
  - Insert new rows into a table using the supplied parameter values
  - Update data in a table for rows that match the supplied parameter values
  - Delete rows from a table that match the supplied parameter values
  - Query a table and retrieve data based on supplied parameter values
- Handling exceptions in procedures
- Compiling and invoking procedures

ORACLE

Creating Stored Functions

Objectives

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

- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Differentiate between a procedure and a function

Overview of Stored Functions

A function:

- **Is a named PL/SQL block that returns a value**
- **Can be stored in the database as a schema object for repeated execution**
- **Is called as part of an expression or is used to provide a parameter value**

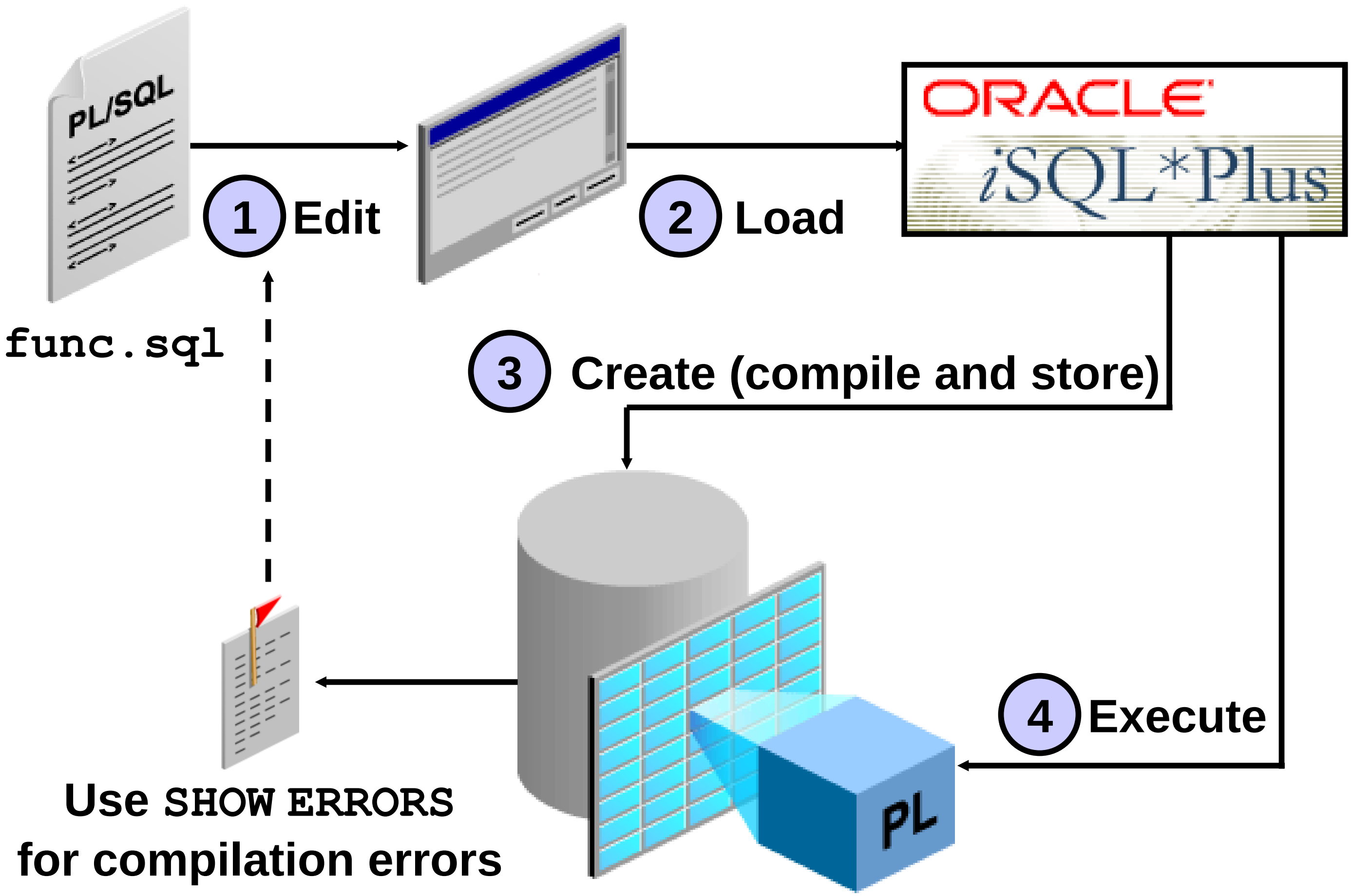
Syntax for Creating Functions

The PL/SQL block must have at least one `RETURN` statement.

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1, ...)]
RETURN datatype IS|AS
 [local_variable_declarations; …]
BEGIN
  -- actions;
  RETURN expression;
END [function_name];
```

PL/SQL Block

Developing Functions



Stored Function: Example

- Create the function:

```
CREATE OR REPLACE FUNCTION get_sal
 (id employees.employee_id%TYPE) RETURN NUMBER IS
  sal employees.salary%TYPE := 0;
BEGIN
  SELECT salary
  INTO   sal
  FROM   employees
  WHERE  employee_id = id;
  RETURN sal;
END get_sal;
/
```

- Invoke the function as an expression or as a parameter value:

```
EXECUTE dbms_output.put_line(get_sal(100))
```

Ways to Execute Functions

- Invoke as part of a PL/SQL expression
  - Using a host variable to obtain the result

```
VARIABLE salary NUMBER
EXECUTE :salary := get_sal(100)
```

  - Using a local variable to obtain the result

```
DECLARE sal employees.salary%type;
BEGIN
  sal := get_sal(100); ...
END;
```

- Use as a parameter to another subprogram

```
EXECUTE dbms_output.put_line(get_sal(100))
```

- Use in a SQL statement (subject to restrictions)

```
SELECT job_id, get_sal(employee_id) FROM employees;
```

Advantages of User-Defined Functions in SQL Statements

- Can extend SQL where activities are too complex, too awkward, or unavailable with SQL
- Can increase efficiency when used in the `WHERE` clause to filter data, as opposed to filtering the data in the application
- Can manipulate data values

Function in SQL Expressions: Example

```
CREATE OR REPLACE FUNCTION tax(value IN NUMBER)
 RETURN NUMBER IS
BEGIN
   RETURN (value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```

```
Function created.
```

EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
108	Greenberg	12000	960
109	Faviet	9000	720
110	Chen	8200	656
111	Sciarra	7700	616
112	Urman	7800	624
113	Popp	6900	552

```
6 rows selected.
```

Locations to Call User-Defined Functions

User-defined functions act like built-in single-row functions and can be used in:

- The `SELECT` list or clause of a query
- Conditional expressions of the `WHERE` and `HAVING` clauses
- The `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses of a query
- The `VALUES` clause of the `INSERT` statement
- The `SET` clause of the `UPDATE` statement

Restrictions on Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
  - Be stored in the database
  - Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types
  - Return valid SQL data types, not PL/SQL-specific types
- When calling functions in SQL statements:
  - Parameters must be specified with positional notation
  - You must own the function or have the `EXECUTE` privilege

Controlling Side Effects When Calling Functions from SQL Expressions

Functions called from:

- **A `SELECT` statement cannot contain DML statements**
- **An `UPDATE` or `DELETE` statement on a table `T` cannot query or contain DML on the same table `T`**
- **SQL statements cannot end transactions (that is, cannot execute `COMMIT` or `ROLLBACK` operations)**

Note: Calls to subprograms that break these restrictions are also not allowed in the function.

Restrictions on Calling Functions from SQL: Example

```
CREATE OR REPLACE FUNCTION dml_call_sql(sal NUMBER)
   RETURN NUMBER IS
BEGIN
  INSERT INTO employees(employee_id, last_name,
                 email, hire_date, job_id, salary)
  VALUES(1, 'Frost', 'jfrost@company.com',
         SYSDATE, 'SA_MAN', sal);
  RETURN (sal + 100);
END;
```

```
UPDATE employees
  SET salary = dml_call_sql(2000)
WHERE employee_id = 170;
```

```
UPDATE employees SET salary = dml_call_sql(2000)
              *
ERROR at line 1:
ORA-04091: table PLSQL.EMPLOYEES is mutating,
trigger/function may not see it
ORA-06512: at "PLSQL.DML_CALL_SQL", line 4
```

Removing Functions

Removing a stored function:

- You can drop a stored function by using the following syntax:

```
DROP FUNCTION function_name
```

Example:

```
DROP FUNCTION get_sal;
```

- All the privileges that are granted on a function are revoked when the function is dropped.
- The `CREATE OR REPLACE` syntax is equivalent to dropping a function and re-creating it. Privileges granted on the function remain the same when this syntax is used.

Viewing Functions in the Data Dictionary

Information for PL/SQL functions is stored in the following Oracle data dictionary views:

- You can view source code in the `USER_SOURCE` table for subprograms that you own, or the `ALL_SOURCE` table for functions owned by others who have granted you the `EXECUTE` privilege.

```
SELECT text
FROM   user_source
WHERE  type = 'FUNCTION'
ORDER BY line;
```

- You can view the names of functions by using `USER_OBJECTS`.

```
SELECT object_name
FROM   user_objects
WHERE  object_type = 'FUNCTION';
```

Procedures Versus Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain `RETURN` clause in the header	Must contain a `RETURN` clause in the header
Can return values (if any) in output parameters	Must return a single value
Can contain a `RETURN` statement without a value	Must contain at least one `RETURN` statement

Summary

In this lesson, you should have learned how to:

- Write a PL/SQL function to compute and return a value by using the `CREATE FUNCTION` SQL statement
- Invoke a function as part of a PL/SQL expression
- Use stored PL/SQL functions in SQL statements
- Remove a function from the database by using the `DROP FUNCTION` SQL statement

Practice 2: Overview

This practice covers the following topics:

- Creating stored functions
  - To query a database table and return specific values
  - To be used in a SQL statement
  - To insert a new row, with specified parameter values, into a database table
  - Using default parameter values
- Invoking a stored function from a SQL statement
- Invoking a stored function from a stored procedure

1

Writing Explicit Cursors

Objectives

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

- Distinguish between an implicit and an explicit cursor
- Discuss when and why to use an explicit cursor
- Use a PL/SQL record variable
- Write a cursor FOR loop

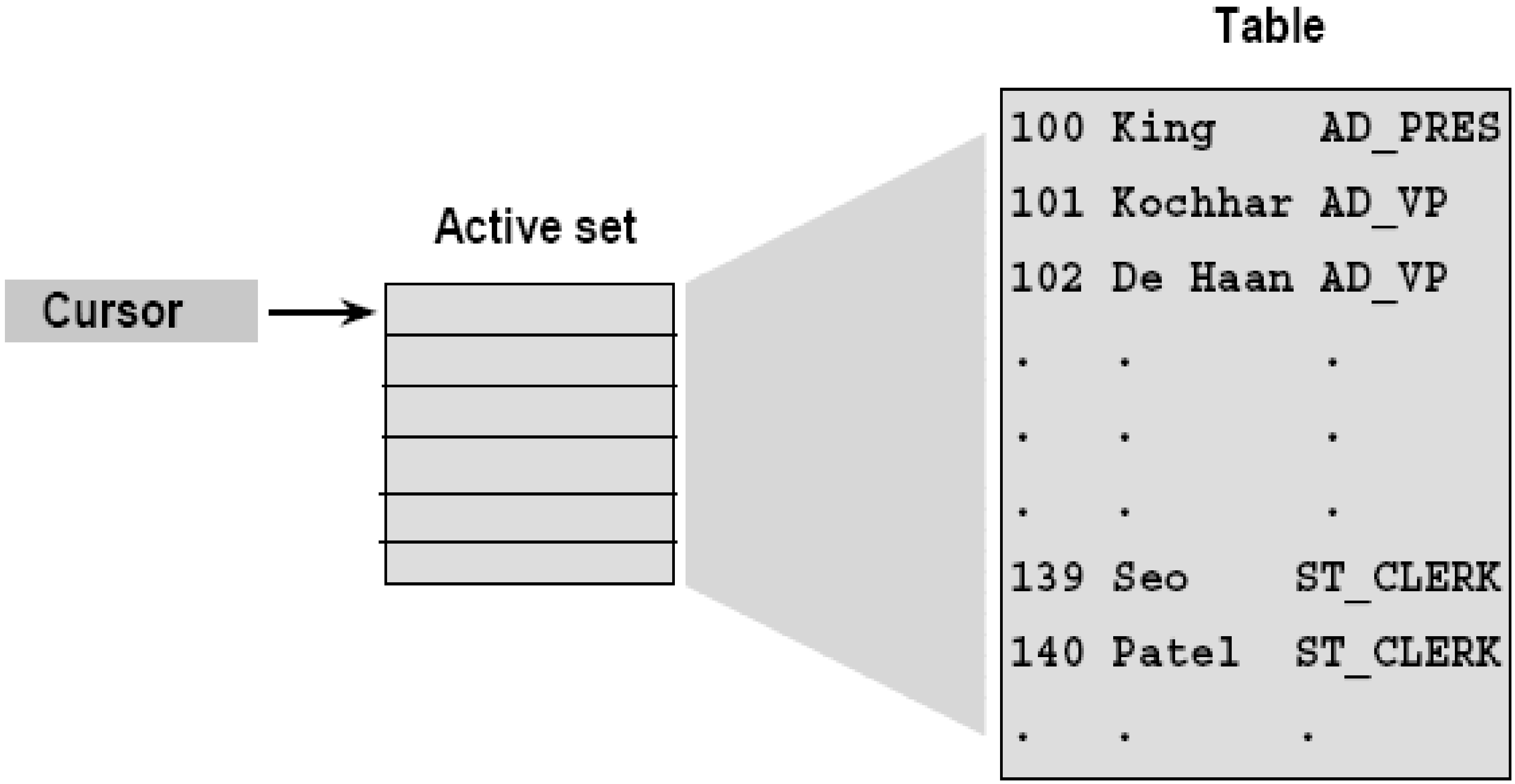
About Cursors

Every SQL statement executed by the Oracle Server has an individual cursor associated with it:

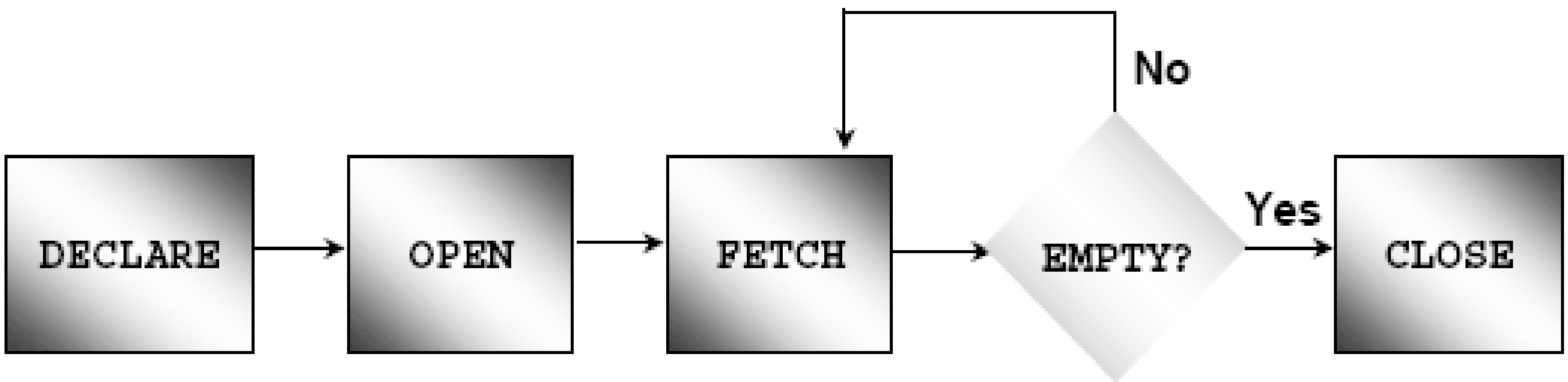
Implicit cursors: Declared for all DML and PL/SQL SELECT statements

Explicit cursors: Declared and named by the programmer

Explicit Cursor Functions



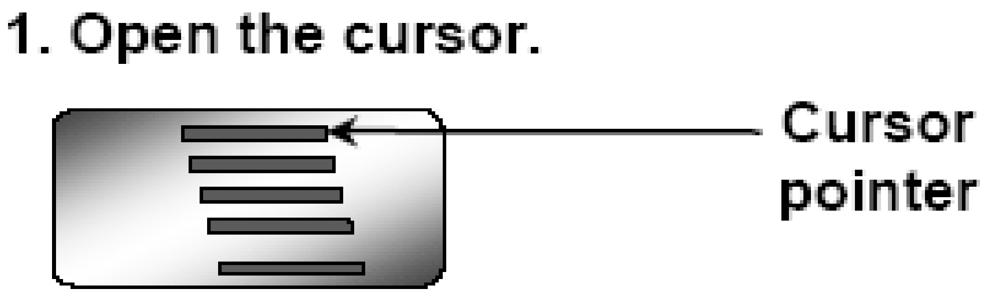
Controlling Explicit Cursors



- Create a named SQL area
- Identify the active set
- Load the current row into variables
- Test for existing rows
- Return to `FETCH` if rows are found
- Release the active set

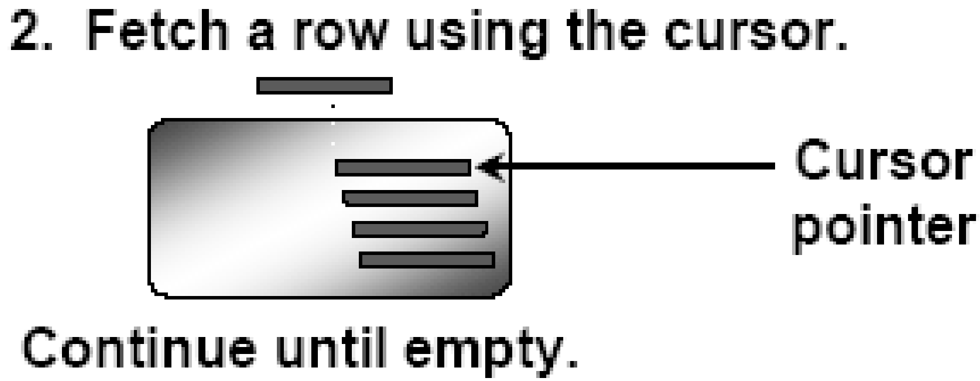
Controlling Explicit Cursors

1. Open the cursor
2. Fetch a row
3. Close the Cursor



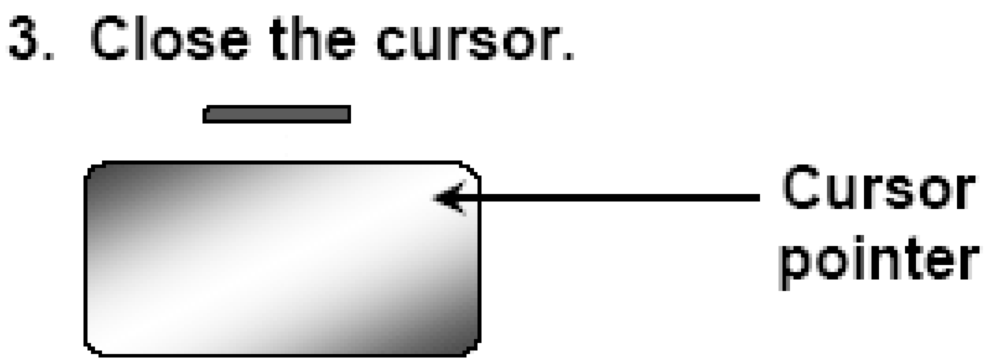
Controlling Explicit Cursors

1. Open the cursor
2. Fetch a row
3. Close the Cursor



Controlling Explicit Cursors

1. Open the cursor
2. Fetch a row
3. Close the Cursor



Declaring the Cursor

Syntax:

CURSOR ***cursor_name*** **IS**
select_statement;

Do not include the INTO clause in the cursor declaration.

If processing rows in a specific sequence is required, use the ORDER BY clause in the query.

Declaring the Cursor

Example:

```
DECLARE
     CURSOR emp_cursor IS
               SELECT employee_id, last_name
               FROM employees;
     CURSOR dept_cursor IS
               SELECT *
               FROM departments
               WHERE location_id = 170;
BEGIN
...
```

Opening the Cursor

Syntax:

```
OPEN cursor_name;
```

- **Open the cursor to execute the query and identify the active set.**
- **If the query returns no rows, no exception is raised.**
- **Use cursor attributes to test the outcome after a fetch.**

Fetching Data from the Cursor

Syntax:

```
FETCH cursor_name INTO [variable1, variable2, ...]
                                          | record_name];
```

- **Retrieve the current row values into variables.**
- **Include the same number of variables.**
- **Match each variable to correspond to the columns positionally.**
- **Test to see whether the cursor contains rows.**

Fetching Data from the Cursor

Example:

```
LOOP
     FETCH emp_cursor INTO v_empno,v_ename;
     EXIT WHEN ...;
     ...
          -- Process the retrieved data
     …
END LOOP;
```

Closing the Cursor

Syntax:

```
CLOSE cursor_name;
```

- **Close the cursor after completing the processing of the rows.**
- **Reopen the cursor, if required.**
- **Do not attempt to fetch data from a cursor after it has been closed.**

Explicit Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
`%ISOPEN`	Boolean	Evaluates to `TRUE` if the cursor is open
`%NOTFOUND`	Boolean	Evaluates to `TRUE` if the most recent fetch does not return a row
`%FOUND`	Boolean	Evaluates to `TRUE` if the most recent fetch returns a row; complement of `%NOTFOUND`
`%ROWCOUNT`	Number	Evaluates to the total number of rows returned so far

ORACLE

The %ISOPEN Attribute

- Fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.

Example:

```
IF  NOT  emp_cursor%ISOPEN  THEN
        OPEN emp_cursor;
END IF;
LOOP
        FETCH emp_cursor ...
```

Controlling Multiple Fetches

- Process several rows from an explicit cursor using a loop.
- Fetch a row with each iteration.
- Use explicit cursor attributes to test the success of each fetch.

The %NOTFOUND and %ROWCOUNT Attributes

- Use the %ROWCOUNT cursor attribute to retrieve an exact number of rows.
- Use the %NOTFOUND cursor attribute to determine when to exit the loop.

Example

```
SET SERVEROUTPUT ON
DECLARE
    v_empno employees.employee_id%TYPE;
    v_ename employees.last_name%TYPE;
    CURSOR emp_cursor IS
          SELECT employee_id, last_name FROM employees;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_empno, v_ename;
    EXIT WHEN emp_cursor%ROWCOUNT > 10 OR emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_empno) ||' '|| v_ename);
  END LOOP;
  CLOSE emp_cursor;
END ;
```

Cursors and Records

Process the rows of the active set by fetching values into a PL/SQL RECORD.

```
CREATE TABLE temp_list AS SELECT employee_id, last_name
                          FROM employees WHERE employee_id = 50;
DECLARE
  CURSOR emp_cursor IS SELECT employee_id, last_name  FROM employees;
  emp_record emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
        FETCH emp_cursor INTO emp_record;
        EXIT WHEN emp_cursor%NOTFOUND;
        INSERT INTO temp_list (empid, empname)
        VALUES (emp_record.employee_id, emp_record.last_name);
  END LOOP;
  COMMIT;
  CLOSE emp_cursor;
END;
```

Cursor FOR Loops

Syntax:

```
FOR record_name IN cursor_name LOOP
        statement1;
        statement2;
        . . .
END LOOP;
```

- **The cursor FOR loop is a shortcut to process explicit cursors.**
- **Implicit open, fetch, exit, and close occur.**
- **The record is implicitly declared.**

Cursor FOR Loops

Print a list of the employees who work for the sales department.

```
SET SERVEROUTPUT ON
DECLARE
     CURSOR emp_cursor IS  SELECT last_name, department_id
                           FROM employees;
BEGIN
  FOR emp_record IN emp_cursor LOOP
  --implicit open and implicit fetch occur
    IF emp_record.department_id = 80 THEN
     DBMS_OUTPUT.PUT_LINE ('Employee ' ||
     emp_record.last_name || ' works in the Sales Dept. ');
    END IF;
  END LOOP; --implicit close and implicit loop exit
END ;
```

Cursor FOR Loops Using Subqueries

No need to declare the cursor.
Example:

```
SET SERVEROUTPUT ON
BEGIN
 FOR emp_record IN (SELECT last_name, department_id FROM employees) LOOP
  --implicit open and implicit fetch occur
  IF emp_record.department_id = 80 THEN
    DBMS_OUTPUT.PUT_LINE ('Employee ' || emp_record.last_name
                          || ' works in the Sales Dept. ');
  END IF;
 END LOOP; --implicit close occurs
END ;
```

Summary

In this lesson you should have learned to:

- Distinguish cursor types:
  - Implicit cursors: used for all DML statements and single-row queries
  - Explicit cursors: used for queries of zero, one, or more rows
- Manipulate explicit cursors
- Evaluate the cursor status by using cursor attributes
- Use cursor FOR loops

Practice 6 Overview

This practice covers the following topics:

- Declaring and using explicit cursors to query rows of a table
- Using a cursor FOR loop
- Applying cursor attributes to test the cursor status

1

Advanced Explicit Cursor Concepts

Objectives

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

- Write a cursor that uses parameters
- Determine when a FOR UPDATE clause in a cursor is required
- Determine when to use the WHERE CURRENT OF clause
- Write a cursor that uses a subquery

Cursors with Parameters

Syntax:

```
CURSOR   cursor_name
            [(parameter_name datatype, ...)]
IS
            select_statement;
```

- **Pass parameter values to a cursor when the cursor**
- **is opened and the query is executed.**
- **Open an explicit cursor several times with a**
- **different active set each time.**

```
OPEN cursor_name(parameter_value,.....) ;
```

Cursors with Parameters

Pass the department number and job title to the WHERE clause, in the cursor SELECT statement.

```
SET SERVEROUTPUT ON
DECLARE
      CURSOR emp_cursor (p_deptno NUMBER, p_job VARCHAR2) IS
            SELECT employee_id, last_name    FROM employees
            WHERE department_id = p_deptno   AND job_id = p_job;
      emp_c    emp_cursor%rowtype;
BEGIN
        OPEN emp_cursor (80, 'SA_REP');
        LOOP
             FETCH emp_cursor INTO  emp_c;
             EXIT WHEN  emp_cursor%NOTFOUND;
             DBMS_OUTPUT.PUT_LINE ('ROWS : '||emp_cursor%rowcount);
       END LOOP;
       CLOSE emp_cursor;
        OPEN emp_cursor (60, 'IT_PROG');
                 . . .
END;
```

```
SET SERVEROUTPUT ON
DECLARE
     CURSOR dept_c IS SELECT * FROM departments WHERE department_id < 60;
     CURSOR emp_c (p_deptno NUMBER) IS
            SELECT employee_id, last_name   FROM employees
            WHERE department_id = p_deptno;
     dept_r   dept_c%rowtype;
     emp_r    emp_c%rowtype;
BEGIN
       OPEN dept_c;
       LOOP
           FETCH dept_c INTO dept_r;
           EXIT WHEN dept_c%NOTFOUND;
           DBMS_OUTPUT.PUT_LINE (dept_r.department_id||' '||dept_r.department_name);
            OPEN emp_c (dept_r.department_id);
             LOOP
                 FETCH emp_c INTO  emp_r;
                 EXIT WHEN  emp_c%NOTFOUND;
                 DBMS_OUTPUT.PUT_LINE ('NV : '||emp_r.employee_id || '-'||emp_r.last_name);
            END LOOP;
            CLOSE emp_c;
     END LOOP;
     CLOSE dept_c;
END;
```

```
SET SERVEROUTPUT ON
DECLARE
      CURSOR dept_c IS SELECT * FROM departments
                         WHERE department_id < 60;
      CURSOR emp_c (p_deptno NUMBER) IS
               SELECT employee_id, last_name FROM employees
               WHERE department_id = p_deptno;
BEGIN
 FOR dept_r IN dept_c LOOP
   DBMS_OUTPUT.PUT_LINE (dept_r.department_id||' -  '||
                                       dept_r.department_name);
   FOR emp_r IN emp_c (dept_r.department_id)
   LOOP
     DBMS_OUTPUT.PUT_LINE ('NV : '||emp_r.employee_id || '-'||
                                              emp_r.last_name);
   END LOOP;
 END LOOP;
END;
```

ORACLE

```
SET SERVEROUTPUT ON
BEGIN
  FOR dept_r IN (SELECT * FROM departments
                WHERE department_id<60)
  LOOP
    DBMS_OUTPUT.PUT_LINE (dept_r.department_id||' -  '||
                                dept_r.department_name);
    FOR emp_r IN (SELECT employee_id, last_name
                FROM employees
                WHERE department_id =dept_r.department_id)
     LOOP
       DBMS_OUTPUT.PUT_LINE ('NV : '||emp_r.employee_id || '-'||
                                          emp_r.last_name);
     END LOOP;
    END LOOP;
END;
```

The FOR UPDATE Clause

Syntax:

```
SELECT ...
FROM ...
FOR UPDATE [OF column_reference][NOWAIT];
```

- **Use explicit locking to deny access for the**
- **duration of a transaction.**
- **Lock the rows before the update or delete.**

The FOR UPDATE Clause

Retrieve the employees who work in department 80 and update their salary.

```
DECLARE
     CURSOR emp_cursor IS
         SELECT employee_id, last_name, department_name
         FROM employees,departments
         WHERE employees.department_id =
              departments.department_id
         AND employees.department_id = 80
         FOR UPDATE OF salary NOWAIT;
```

The WHERE CURRENT OF Clause

Syntax:

```
WHERE CURRENT OF cursor ;
```

- **Use cursors to update or delete the current row.**
- **Include the FOR UPDATE clause in the cursor**
- **query to lock the rows first.**
- **Use the WHERE CURRENT OF clause to reference**
- **the current row from an explicit cursor.**

The WHERE CURRENT OF Clause

```
SET SERVEROUTPUT ON
DECLARE
CURSOR sal_cursor IS
      SELECT department_id, employee_id emp_id, last_name, salary
      FROM employees    WHERE department_id = 20
      FOR UPDATE OF salary NOWAIT;
BEGIN
      FOR emp_r IN sal_cursor      LOOP
           DBMS_OUTPUT.PUT_LINE (emp_r.emp_id||'-'||emp_r.salary);
           IF  emp_r.salary  >  5000  THEN
           UPDATE employees  SET salary = emp_r.salary * 1.10
           WHERE CURRENT OF sal_cursor;
           END IF;
      END LOOP;
      COMMIT;
END;
/
SELECT department_id, employee_id emp_id, last_name, salary
FROM employees      WHERE department_id = 20;
```

Cursors with Subqueries

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR my_cursor IS
    SELECT t1.department_id, t1.department_name, t2.staff
    FROM departments t1,
      (SELECT department_id dept_id, COUNT(*) AS STAFF
       FROM employees GROUP BY department_id) t2
    WHERE t1.department_id = t2.dept_id AND t2.staff  >=  3;
BEGIN
  FOR c1 IN my_cursor
  LOOP
    DBMS_OUTPUT.PUT_LINE (c1.department_name||'-'|| c1.staff);
  END LOOP;
END;
/
```

Summary

In this lesson, you should have learned to:

Return different active sets using cursors with parameters.

Define cursors with subqueries and correlated subqueries.

Manipulate explicit cursors with commands using the:

- FOR UPDATE clause
- WHERE CURRENT OF clause

Practice 7 Overview

This practice covers the following topics:

- Declaring and using explicit cursors with parameters
- Using a FOR UPDATE cursor

1

Design Considerations

Objectives

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

- Identify guidelines for cursor design
- Use cursor variables
- Create subtypes based on existing types for an application

Guidelines for Cursor Design

Fetch into a record when fetching from a cursor.

```
DECLARE
  CURSOR cur_cust IS
    SELECT customer_id, cust_last_name, cust_email
    FROM customers
    WHERE credit_limit = 1200;
  v_cust_record    cur_cust%ROWTYPE;
BEGIN
  OPEN cur_cust;
  LOOP
    FETCH cur_cust INTO v_cust_record;
...
```

Guidelines for Cursor Design

Create cursors with parameters.

```
CREATE OR REPLACE PROCEDURE cust_pack
 (p_crd_limit_in NUMBER, p_acct_mgr_in NUMBER)
IS
  v_credit_limit NUMBER := 1500;

  CURSOR cur_cust (p_crd_limit NUMBER, p_acct_mgr NUMBER)
    IS
    SELECT customer_id, cust_last_name, cust_email
    FROM customers
    WHERE credit_limit = p_crd_limit
         AND  account_mgr_id = p_acct_mgr;

  cust_record    cur_cust%ROWTYPE;
BEGIN
   OPEN cur_cust(p_crd_limit_in, p_acct_mgr_in);
...
   CLOSE cur_cust;
...
   OPEN cur_cust(v_credit_limit, 145);
...
END;
```

Guidelines for Cursor Design

Reference implicit cursor attributes immediately after the SQL statement executes.

```
BEGIN
  UPDATE customers
    SET   credit_limit = p_credit_limit
    WHERE customer_id  = p_cust_id;
  get_avg_order(p_cust_id);  -- procedure call
  IF SQL%NOTFOUND THEN
      ...
```

Guidelines for Cursor Design

Simplify coding with cursor `FOR` loops.

```
CREATE OR REPLACE PROCEDURE cust_pack
 (p_crd_limit_in NUMBER, p_acct_mgr_in NUMBER)
IS
  v_credit_limit NUMBER := 1500;
   CURSOR cur_cust
           (p_crd_limit NUMBER, p_acct_mgr NUMBER)
     IS
     SELECT customer_id, cust_last_name, cust_email
     FROM customers
     WHERE credit_limit = p_crd_limit
     AND   account_mgr_id = p_acct_mgr;
   cust_record    cur_cust%ROWTYPE;
BEGIN
  FOR cust_record IN cur_cust (p_crd_limit_in, p_acct_mgr_in)
  LOOP        -- implicit open and fetch
  ...
  END LOOP;    -- implicit close
  ...
END;
```

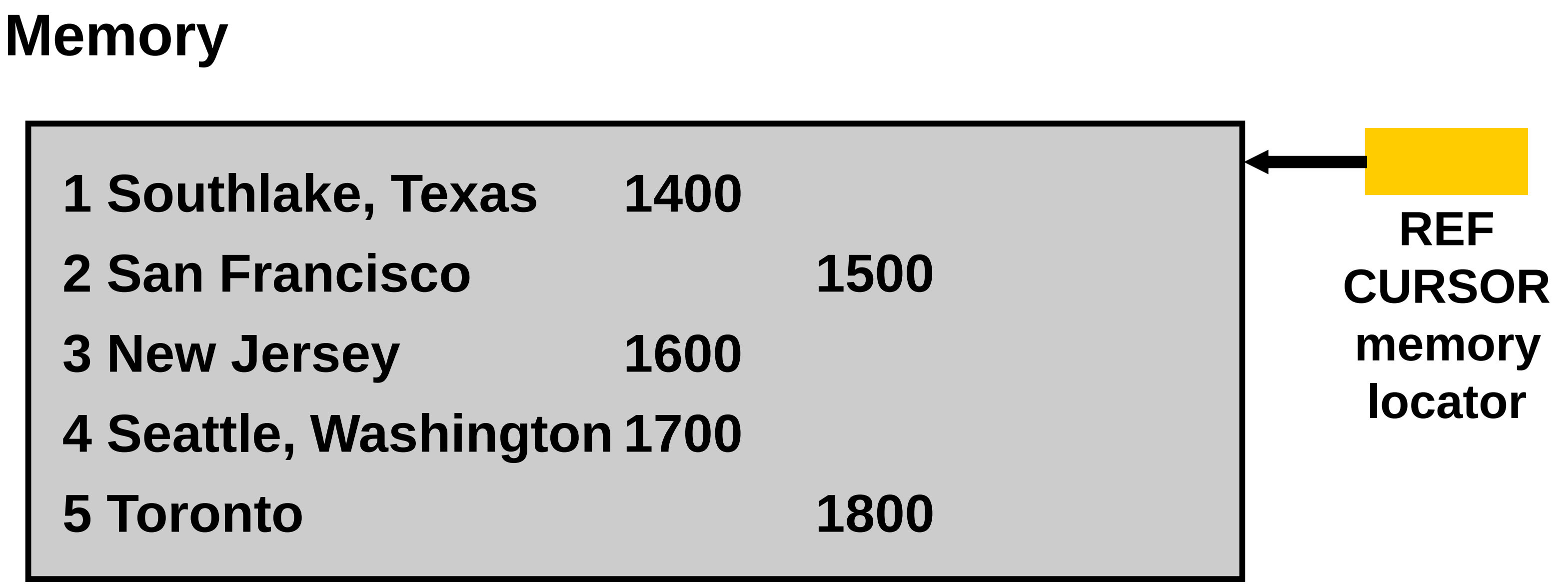
Guidelines for Cursor Design

- Close a cursor when it is no longer needed.
- Use column aliases in cursors for calculated columns fetched into records declared with `%ROWTYPE`.

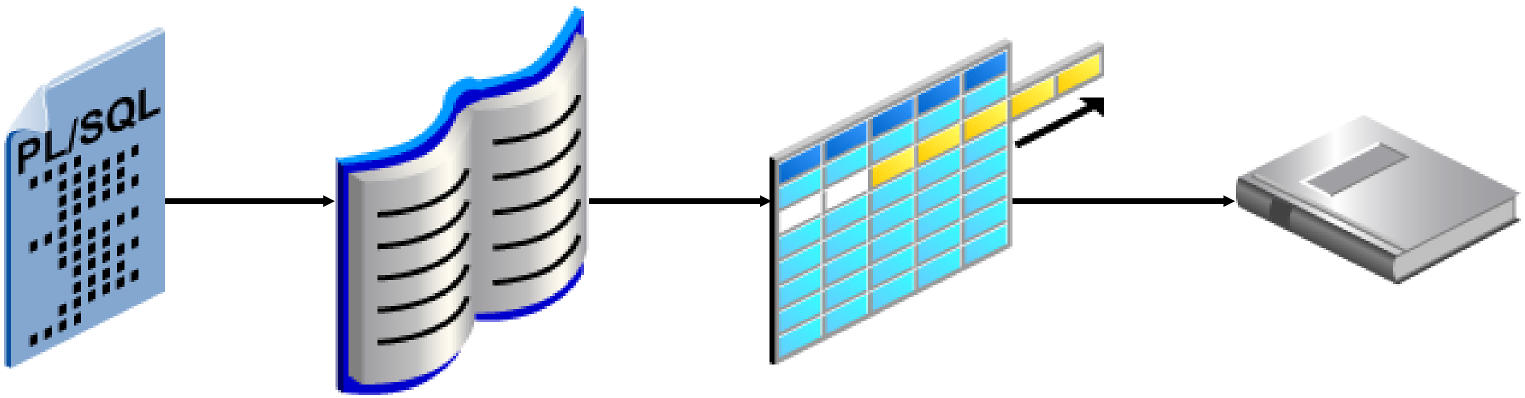
```
CREATE OR REPLACE PROCEDURE cust_list
IS
 CURSOR  cur_cust IS
  SELECT customer_id, cust_last_name, credit_limit*1.1
  FROM   customers;
 cust_record cur_cust%ROWTYPE;
BEGIN
  OPEN cur_cust;
  LOOP
    FETCH cur_cust INTO cust_record;
    DBMS_OUTPUT.PUT_LINE('Customer ' ||
      cust_record.cust_last_name || ' wants credit '
      || cust_record.(credit_limit * 1.1));
    EXIT WHEN cur_cust%NOTFOUND;
  END LOOP;
...
```

Use col. alias

Cursor Variables



Using a Cursor Variable



Define and declare the cursor variable.	Open the cursor variable.	Fetch rows from the result set.	Close the cursor variable.
1	2	3	4

Strong Versus Weak Cursors

- Strong cursor:
  - Is restrictive
  - Specifies a `RETURN` type
  - Associates with type-compatible queries only
  - Is less error prone
- Weak cursor:
  - Is nonrestrictive
  - Associates with any query
  - Is very flexible

Step 1: Defining a `REF CURSOR` Type

Define a `REF CURSOR` type:

```
TYPE  ref_type_name IS REF CURSOR
  [RETURN return_type];
```

- ***ref_type_name*** is a type specifier in subsequent declarations.
- ***return_type*** represents a record type.
- ***return_type*** indicates a strong cursor.

```
DECLARE
  TYPE rt_cust IS REF CURSOR
   RETURN customers%ROWTYPE;
  ...
```

Step 1: Declaring a Cursor Variable

Declare a cursor variable of a cursor type:

```
CURSOR_VARIABLE_NAME    REF_TYPE_NAME
```

- ***cursor_variable_name*** **is the name of the cursor variable.**
- ***ref_type_name*** **is the name of a `REF CURSOR` type.**

```
DECLARE
  TYPE rt_cust IS REF CURSOR
    RETURN customers%ROWTYPE;
  cv_cust rt_cust;
```

Step 1: Declaring a REF CURSOR Return Type

Options:

- Use %TYPE and %ROWTYPE.
- Specify a user-defined record in the RETURN clause.
- Declare the cursor variable as the formal parameter of a stored procedure or function.

Step 2: Opening a Cursor Variable

- Associate a cursor variable with a multirow `SELECT` statement.
- Execute the query.
- Identify the result set:

```
OPEN cursor_variable_name
   FOR select_statement
```

  - *cursor_variable_name* is the name of the cursor variable.
  - *select_statement* is the SQL `SELECT` statement.

Step 3: Fetching from a Cursor Variable

- **Retrieve rows from the result set one at a time.**

```
FETCH cursor_variable_name
 INTO variable_name1
     [,variable_name2,. . .]
     | record_name;
```

- **The return type of the cursor variable must be compatible with the variables named in the `INTO` clause of the `FETCH` statement.**

Step 4: Closing a Cursor Variable

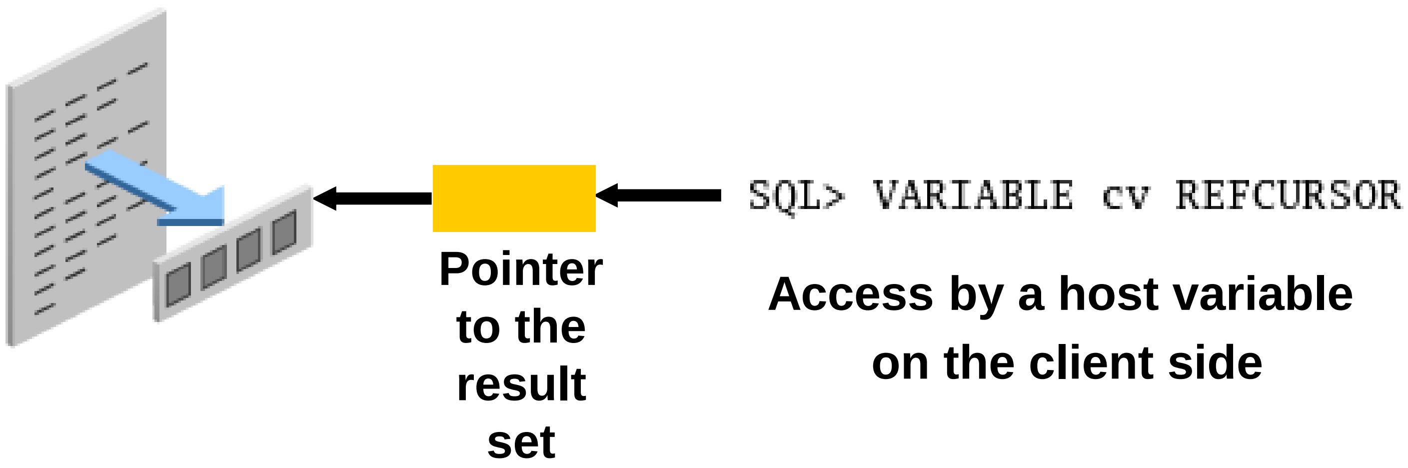
- Disable a cursor variable.
- The result set is undefined.

```
CLOSE cursor_variable_name ;
```

- Accessing the cursor variable after it is closed raises the predefined exception `INVALID_CURSOR`.

Passing Cursor Variables as Arguments

You can pass query result sets among PL/SQL stored subprograms and various clients.



Passing Cursor Variables as Arguments

```
SQL> EXECUTE cust_data.get_cust(112, :cv)
PL/SQL procedure successfully completed.

SQL> print cv
```

```
CUSTOMER_ID CUST_FIRST_NAME      CREDIT_LIMIT CUST_EMAIL
----------- -------------------- ------------ ------------------------------
        112 Guillaume                     200 Guillaume.Jackson@MOORHEN.COM
```

ORACLE

Rules for Cursor Variables

- Cursor variables cannot be used with remote subprograms on another server.
- The query associated with a cursor variable in an `OPEN-FOR` statement should not be `FOR UPDATE`.
- You cannot use comparison operators to test cursor variables.
- Cursor variables cannot be assigned a null value.
- You cannot use `REF CURSOR` types in `CREATE TABLE` or `VIEW` statements.
- Cursors and cursor variables are not interoperable.

Comparing Cursor Variables with Static Cursors

Cursor variables have the following benefits:

- **Are dynamic and ensure more flexibility**
- **Are not tied to a single `SELECT` statement**
- **Hold the value of a pointer**
- **Can reduce network traffic**
- **Give access to query work area after a block completes**

Predefined Data Types

Scalar Types

BINARY_DOUBLE
BINARY_FLOAT
BINARY_INTEGER
DEC
DECIMAL
DOUBLE_ PRECISION
FLOAT
INT
INTEGER
NATURAL
NATURALN
NUMBER
NUMERIC
PLS_INTEGER
POSITIVE
POSITIVEN
REAL
SINGTYPE
SMALLINT

CHAR
CHARACTER
LONG
LONG RAW
NCHAR
NVARCHAR2
RAW
ROWID
STRING
UROWID
VARCHAR
VARCHAR2

BOOLEAN

DATE

Composite Types

RECORD
TABLE
VARRAY

Reference Types

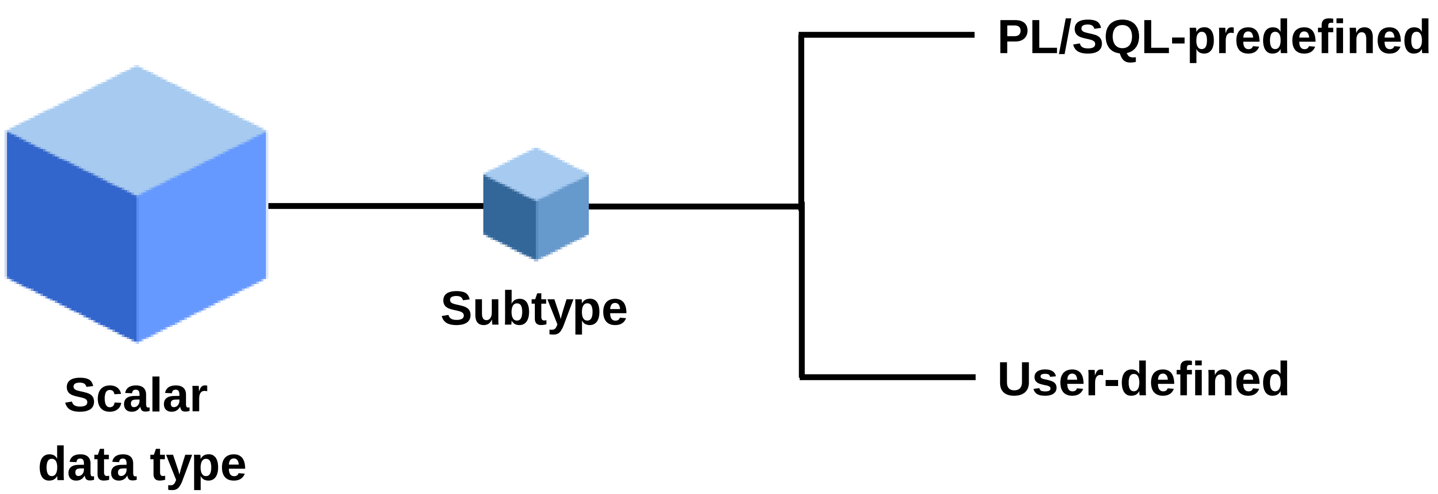
REF CURSOR
REF object_type

LOB Types

BFILE
BLOB
CLOB
NCLOB

Subtypes

A subtype is a subset of an existing data type that may place a constraint on its base type.



Benefits of Subtypes

Subtypes:

- **Increase reliability**
- **Provide compatibility with ANSI/ISO and IBM types**
- **Promote reusability**
- **Improve readability**
  - **Clarity**
  - **Code self-documents**

Declaring Subtypes

- **Subtypes are defined in the declarative section of any PL/SQL block.**

```
SUBTYPE subtype_name IS base_type [(constraint)] [NOT NULL];
```

- ***subtype_name*** **is a type specifier used in subsequent declarations.**
- ***base_type*** **is any scalar or user-defined PL/SQL type.**

Using Subtypes

- Define an identifier that uses the subtype in the declarative section.

```
identifier_name  subtype_name
```

- You can constrain a user-defined subtype when declaring variables of that type.

```
identifier_name  subtype_name(size)
```

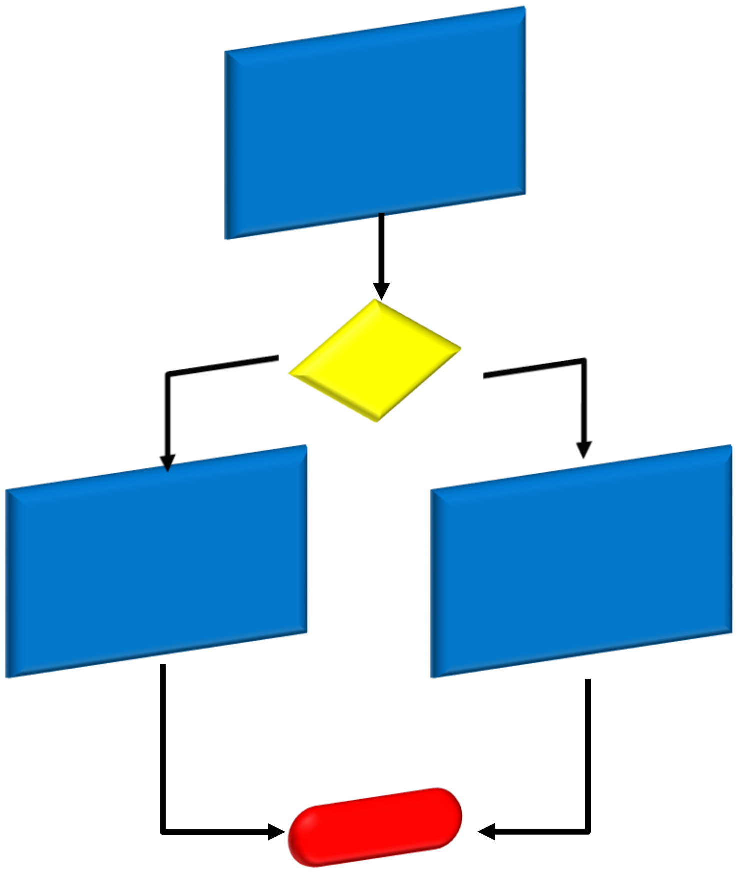
- You can constrain a user-defined subtype when declaring the subtype.

Subtype Compatibility

An unconstrained subtype is interchangeable with its base type.

```
DECLARE
  SUBTYPE Accumulator IS NUMBER;
  v_amount  NUMBER(4,2);
  v_total   Accumulator;
BEGIN
  v_amount := 99.99;
  v_total  := 100.00;
  dbms_output.put_line('Amount is: ' || v_amount);
  dbms_output.put_line('Total is: '  || v_total);
  v_total := v_amount;
  dbms_output.put_line('This works too: '  ||
  v_total);
  --  v_amount := v_amount + 1; Will show value error
END;
/
```

Conditional Logic



Conditional logic

Condition:

```
If <cond>
    then <command>
elsif <cond2>
    then <command2>
else
         <command3>
end if;
```

Nested conditions:

```
If <cond>
   then
       if <cond2>
           then
             <command1>
        end if;
 else <command2>
 end if;
```

IF-THEN-ELSIF Statements

Example:

```
. . .
IF rating > 7 THEN
  v_message := 'You are great';
ELSIF rating >= 5 THEN
  v_message := 'Not bad';
ELSE
  v_message := 'Pretty bad';
END IF;
. . .
```

IF-THEN-ELSE Statements

Example:

```
DECLARE
  cnt  NUMBER;
BEGIN
  SELECT count(*)
   INTO cnt
  FROM  mylog
  WHERE who = user;

  IF cnt > 0 THEN
    UPDATE mylog
      SET logon_num = logon_num + 1
    WHERE who = user;
  ELSE
    INSERT INTO mylog VALUES (user, 1);
  END IF;
  COMMIT;
END;
/
```

Loops: Simple Loop

Example:

```
create table number_table(
   num NUMBER(10)
);
```

```
DECLARE
  i  number_table.num%TYPE := 1;
BEGIN
  LOOP
    INSERT INTO number_table
      VALUES(i);
    i := i + 1;
    EXIT WHEN i > 10;
  END LOOP;
END;
```

Loops: Simple Cursor Loop

Example:

```
create table number_table(
  num NUMBER(10)
);
```

```
DECLARE
  cursor c is select * from number_table;
  cVal c%ROWTYPE;
BEGIN
  open c;
  LOOP
    fetch c into cVal;
    EXIT WHEN c%NOTFOUND;
    insert into doubles values(cVal.num*2);
  END LOOP;
END;
```

Loops: FOR Loop

Example:

```
DECLARE
  i   number_table.num%TYPE;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO number_table VALUES(i);
  END LOOP;
END;
```

Notice that i is incremented automatically

Loops: For Cursor Loops

Example:

```
DECLARE
  cursor c is select * from number_table;

BEGIN
   for num_row in c loop
       insert into doubles_table
                    values(num_row.num*2);
   end loop;
END;
/
```

Loops: WHILE Loop

Example:

```
DECLARE
TEN number:=10;
i  number_table.num%TYPE:=1;
BEGIN
  WHILE i <= TEN LOOP
     INSERT INTO number_table
     VALUES(i);
     i := i + 1;
  END LOOP;
END;
/
```

Summary

In this lesson, you should have learned how to:

- Use guidelines for cursor design
- Declare, define, and use cursor variables
- Use subtypes as data types

Practice Overview

This practice covers the following topics:

- Determining the output of a PL/SQL block
- Improving the performance of a PL/SQL block
- Implementing subtypes
- Using cursor variables

1

Dynamic SQL and Metadata

Objectives

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

- Describe the execution flow of SQL statements
- Build and execute SQL statements dynamically using Native Dynamic SQL (that is, with `EXECUTE IMMEDIATE` statements)
- Compare Native Dynamic SQL with the `DBMS_SQL` package approach
- Use the `DBMS_METADATA` package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects

Execution Flow of SQL

- **All SQL statements go through various stages:**
  - **Parse**
  - **Bind**
  - **Execute**
  - **Fetch**
- **Some stages may not be relevant for all statements—for example, the fetch phase is applicable to queries.**

Note: For embedded SQL statements (`SELECT`, DML, `COMMIT` and `ROLLBACK`), the parse and bind phases are done at compile time. For dynamic SQL statements, all phases are performed at run time.

Dynamic SQL

Use dynamic SQL to create a SQL statement whose structure may change during run time. Dynamic SQL:

- **Is constructed and stored as a character string within the application**
- **Is a SQL statement with varying column data, or different conditions with or without placeholders (bind variables)**
- **Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL**
- **Is executed with Native Dynamic SQL statements or the `DBMS_SQL` package**

ORACLE

Native Dynamic SQL

- Provides native support for dynamic SQL directly in the PL/SQL language
- 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

Use the `EXECUTE IMMEDIATE` statement for Native Dynamic SQL or PL/SQL anonymous blocks:

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

- `INTO` is used for single-row queries and specifies the variables or records into which column values are retrieved.
- `USING` is used to hold all bind arguments. The default parameter mode is `IN`, if not specified.

Dynamic SQL with a DDL Statement

- **Creating a table:**

```
CREATE PROCEDURE create_table(
  table_name VARCHAR2, col_specs  VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE '||table_name||
                    ' (' || col_specs || ')';
END;
/
```

- **Call example:**

```
BEGIN
  create_table('EMPLOYEE_NAMES',
   'id NUMBER(4) PRIMARY KEY, name VARCHAR2(40)');
END;
/
```

Dynamic SQL with DML Statements

- Deleting rows from any table:

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

```
BEGIN DBMS_OUTPUT.PUT_LINE(
  del_rows('EMPLOYEE_NAMES')|| ' rows deleted.');
END;
```

- Inserting a row into a table with two columns:

```
CREATE PROCEDURE add_row(table_name VARCHAR2,
   id NUMBER, name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO '||table_name||
        ' VALUES (:1, :2)' USING id, name;
END;
```

ORACLE

Dynamic SQL with a Single-Row Query

Example of a single-row query:

```
CREATE FUNCTION get_emp(emp_id NUMBER)
RETURN employees%ROWTYPE IS
  stmt VARCHAR2(200);
  emprec employees%ROWTYPE;
BEGIN
  stmt := 'SELECT * FROM employees ' ||
          'WHERE employee_id = :id';
  EXECUTE IMMEDIATE stmt INTO emprec USING emp_id;
  RETURN emprec;
END;
/
```

```
DECLARE
  emprec employees%ROWTYPE := get_emp(100);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Emp: '||emprec.last_name);
END;
/
```

Dynamic SQL with a Multirow Query

Use `OPEN-FOR`, `FETCH`, and `CLOSE` processing:

```
CREATE PROCEDURE list_employees(deptid NUMBER) IS
  TYPE emp_refcsr IS REF CURSOR;
  emp_cv  emp_refcsr;
  emprec  employees%ROWTYPE;
  stmt varchar2(200) := 'SELECT * FROM employees';
BEGIN
  IF deptid IS NULL THEN OPEN emp_cv FOR stmt;
  ELSE
    stmt := stmt || ' WHERE department_id = :id';
    OPEN emp_cv FOR stmt USING deptid;
  END IF;
  LOOP
    FETCH emp_cv INTO emprec;
    EXIT WHEN emp_cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emprec.department_id||
                      ' ' ||emprec.last_name);
  END LOOP;
  CLOSE emp_cv;
END;
```

Declaring Cursor Variables

- Declare a cursor type as `REF CURSOR`:

```
CREATE PROCEDURE process_data IS
  TYPE ref_ctype IS REF CURSOR; -- weak ref cursor
  TYPE emp_ref_ctype IS REF CURSOR -- strong
     RETURN employees%ROWTYPE;
  :
```

- Declare a cursor variable using the cursor type:

```
  :
  dept_csrvar ref_ctype;
  emp_csrvar  emp_ref_ctype;
BEGIN
  OPEN emp_csrvar FOR SELECT * FROM employees;
  OPEN dept_csrvar FOR SELECT * from departments;
  -- Then use as normal cursors
END;
```

Dynamically Executing a PL/SQL Block

Executing a PL/SQL anonymous block dynamically:

```
CREATE FUNCTION annual_sal(emp_id NUMBER)
RETURN NUMBER IS
  plsql varchar2(200) :=
    'DECLARE '||
    ' emprec employees%ROWTYPE; '||
    'BEGIN '||
    ' emprec := get_emp(:empid); ' ||
    ' :res := emprec.salary * 12; ' ||
    'END;';
  result NUMBER;
BEGIN
 EXECUTE IMMEDIATE plsql
        USING IN emp_id, OUT result;
  RETURN result;
END;
/
```

```
EXECUTE DBMS_OUTPUT.PUT_LINE(annual_sal(100))
```

Using Native Dynamic SQL to Compile PL/SQL Code

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

- `ALTER PROCEDURE name COMPILE`
- `ALTER FUNCTION name COMPILE`
- `ALTER PACKAGE name COMPILE SPECIFICATION`
- `ALTER PACKAGE name COMPILE BODY`

```
CREATE PROCEDURE compile_plsql(name VARCHAR2,
 plsql_type VARCHAR2, options VARCHAR2 := NULL) IS
  stmt varchar2(200) := 'ALTER '|| plsql_type ||
                        ' '|| name || ' COMPILE';
BEGIN
 IF options IS NOT NULL THEN
   stmt := stmt || ' ' || options;
 END IF;
 EXECUTE IMMEDIATE stmt;
END;
/
```

Using the DBMS_SQL Package

The DBMS_SQL package is used to write dynamic SQL in stored procedures and to parse DDL statements. Some of the procedures and functions of the package include:

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

ORACLE

Using DBMS_SQL with a DML Statement

Example of deleting rows:

```
CREATE OR REPLACE FUNCTION delete_all_rows
  (table_name VARCHAR2) RETURN NUMBER IS
  csr_id INTEGER;
  rows_del    NUMBER;
BEGIN
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id,
    'DELETE FROM '||table_name, DBMS_SQL.NATIVE);
  rows_del := DBMS_SQL.EXECUTE (csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  RETURN rows_del;
END;
/
```

```
BEGIN DBMS_OUTPUT.PUT_LINE('Rows Deleted: ' ||
   delete_all_rows('employees'));
END;
```

Using DBMS_SQL with a Parameterized DML Statement

```
CREATE PROCEDURE insert_row (table_name VARCHAR2,
 id VARCHAR2, name VARCHAR2, region NUMBER) IS
  csr_id     INTEGER;
  stmt       VARCHAR2(200);
  rows_added NUMBER;
BEGIN
  stmt := 'INSERT INTO '||table_name||
         ' VALUES (:cid, :cname, :rid)';
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id, stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cid', id);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cname', name);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':rid', region);
  rows_added := DBMS_SQL.EXECUTE(csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  DBMS_OUTPUT.PUT_LINE(rows_added||' row added');
END;
/
```

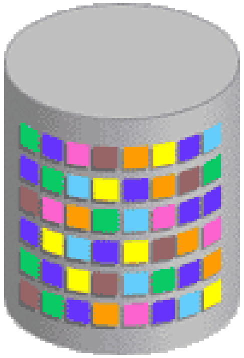
Comparison of Native Dynamic SQL and the DBMS_SQL Package

Native Dynamic SQL:

- Is easier to use than `DBMS_SQL`
- Requires less code than `DBMS_SQL`
- Enhances performance because the PL/SQL interpreter provides native support for it
- Supports all types supported by static SQL in PL/SQL, including user-defined types
- Can fetch rows directly into PL/SQL records

DBMS_METADATA Package

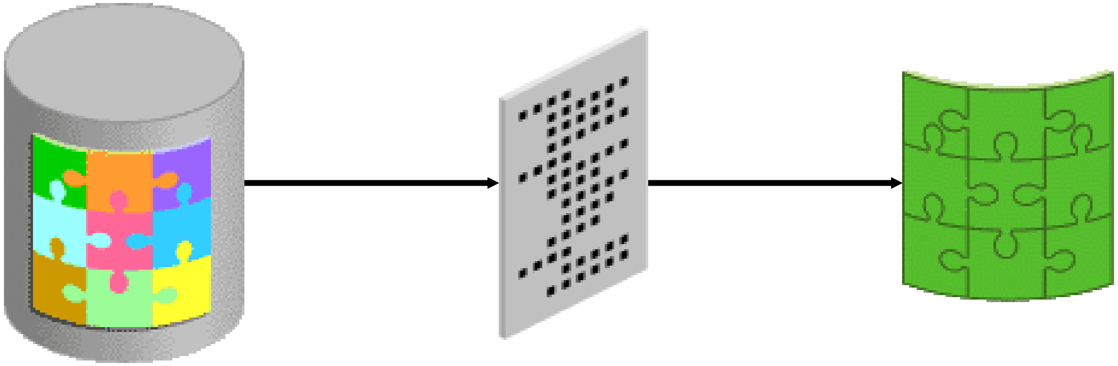
The DBMS_METADATA package provides a centralized facility for the extraction, manipulation, and resubmission of dictionary metadata.



Metadata API

Processing involves the following steps:

1. **Fetch an object's metadata as XML.**
2. **Transform the XML in a variety of ways (including transforming it into SQL DDL).**
3. **Submit the XML to re-create the object.**



Subprograms in DBMS_METADATA

Name	Description
OPEN	Specifies the type of object to be retrieved, the version of its metadata, and the object model. The return value is an opaque context handle for the set of objects.
SET_FILTER	Specifies restrictions on the objects to be retrieved such as the object name or schema
SET_COUNT	Specifies the maximum number of objects to be retrieved in a single FETCH_xxx call
GET_QUERY	Returns the text of the queries that will be used by FETCH_xxx
SET_PARSE_ITEM	Enables output parsing and specifies an object attribute to be parsed and returned
ADD_TRANSFORM	Specifies a transform that FETCH_xxx applies to the XML representation of the retrieved objects
SET_TRANSFORM_PARAM, SET_REMAP_PARAM	Specifies parameters to the XSLT stylesheet identified by transform_handle
FETCH_*XXX*	Returns metadata for objects meeting the criteria established by OPEN, SET_FILTER
CLOSE	Invalidates the handle returned by OPEN and cleans up the associated state

`FETCH_xxx` Subprograms

Name	Description
`FETCH_XML`	This function returns the XML metadata for an object as an XMLType.
`FETCH_DDL`	This function returns the DDL (either to create or to drop the object) into a predefined nested table.
`FETCH_CLOB`	This function returns the objects, transformed or not, as a CLOB.
`FETCH_XML_CLOB`	This procedure returns the XML metadata for the objects as a CLOB in an `IN OUT NOCOPY` parameter to avoid expensive LOB copies.

SET_FILTER Procedure

- **Syntax:**

```
PROCEDURE set_filter
( handle IN NUMBER,
  name   IN VARCHAR2,
  value  IN VARCHAR2|BOOLEAN|NUMBER,
  object_type_path VARCHAR2
);
```

- **Example:**

```
...
DBMS_METADATA.SET_FILTER (handle, 'NAME', 'HR');
...
```

Filters

There are over 70 filters, which are organized into object type categories such as:

- Named objects
- Tables
- Objects dependent on tables
- Index
- Dependent objects
- Granted objects
- Table data
- Index statistics
- Constraints
- All object types
- Database export

Examples of Setting Filters

To set up the filter to fetch the **`HR`** schema objects excluding the object types of functions, procedures, and packages, as well as any views that contain **`PAYROLL`** in the start of the view name:

```
DBMS_METADATA.SET_FILTER(handle, 'SCHEMA_EXPR',
  'IN (''PAYROLL'', ''HR'')');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''FUNCTION''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''PROCEDURE''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''PACKAGE''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_NAME_EXPR',
  'LIKE ''PAYROLL%''', 'VIEW');
```

Programmatic Use: Example 1

```
CREATE PROCEDURE example_one IS
  h    NUMBER; th1  NUMBER; th2  NUMBER;
  doc  sys.ku$_ddls;                                   -- 1
BEGIN
  h := DBMS_METADATA.OPEN('SCHEMA_EXPORT');            -- 2
  DBMS_METADATA.SET_FILTER (h,'SCHEMA','HR');          -- 3
  th1 := DBMS_METADATA.ADD_TRANSFORM (h,               -- 4
    'MODIFY', NULL, 'TABLE');
  DBMS_METADATA.SET_REMAP_PARAM(th1,                   -- 5
    'REMAP_TABLESPACE', 'SYSTEM', 'TBS1');
  th2 :=DBMS_METADATA.ADD_TRANSFORM(h, 'DDL');         -- 4
  DBMS_METADATA.SET_TRANSFORM_PARAM(th2,               -- 6
    'SQLTERMINATOR', TRUE);
  DBMS_METADATA.SET_TRANSFORM_PARAM(th2,               -- 6
    'REF_CONSTRAINTS', FALSE, 'TABLE');
  LOOP
    doc := DBMS_METADATA.FETCH_DDL(h);                 -- 7
    EXIT WHEN doc IS NULL;
  END LOOP;
  DBMS_METADATA.CLOSE(h);                              -- 8
END;
```

Programmatic Use: Example 2

```
CREATE FUNCTION get_table_md RETURN CLOB IS
 h    NUMBER; -- returned by 'OPEN'
 th   NUMBER; -- returned by 'ADD_TRANSFORM'
 doc  CLOB;
BEGIN
 -- specify the OBJECT TYPE
 h := DBMS_METADATA.OPEN('TABLE');
 -- use FILTERS to specify the objects desired
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
 DBMS_METADATA.SET_FILTER(h,'NAME','EMPLOYEES');
 -- request to be TRANSFORMED into creation DDL
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');
 -- FETCH the object
 doc := DBMS_METADATA.FETCH_CLOB(h);
 -- release resources
 DBMS_METADATA.CLOSE(h);
 RETURN doc;
END;
/
SQL> SET PAGESIZE 0
SQL> SET LONG 1000000
SQL> SELECT get_table_md FROM dual;
```

Browsing APIs

Name	Description
`GET_`***`XXX`***	The `GET_XML` and `GET_DDL` functions return metadata for a single named object.
`GET_DEPENDENT_`***`XXX`***	This function returns metadata for a dependent object.
`GET_GRANTED_`***`XXX`***	This function returns metadata for a granted object.

Where ***`xxx`*** is:	`DDL` or `XML`

Browsing APIs: Examples

1. Get XML representation of **HR.EMPLOYEES**:

```
SELECT DBMS_METADATA.GET_XML
                     ('TABLE', 'EMPLOYEES', 'HR')
FROM   dual;
```

2. Fetch the DDL for all object grants on **HR.EMPLOYEES**:

```
SELECT DBMS_METADATA.GET_DEPENDENT_DDL
                    ('OBJECT_GRANT', 'EMPLOYEES', 'HR')
FROM   dual;
```

3. Fetch the DDL for all system grants granted to **HR**:

```
SELECT DBMS_METADATA.GET_GRANTED_DDL
                 ('SYSTEM_GRANT', 'HR')
FROM   dual;
```

Browsing APIs: Examples

```
BEGIN
 DBMS_METADATA.SET_TRANSFORM_PARAM(
   DBMS_METADATA.SESSION_TRANSFORM,
   'STORAGE', false);
END;
/
SELECT DBMS_METADATA.GET_DDL('TABLE',u.table_name)
FROM   user_all_tables u
WHERE  u.nested = 'NO'
AND    (u.iot_type IS NULL OR u.iot_type = 'IOT');

BEGIN
 DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM, 'DEFAULT'):
END;
/
```

1

2

3

Summary

In this lesson, you should have learned how to:

- Explain the execution flow of SQL statements
- Create SQL statements dynamically and execute them using either Native Dynamic SQL statements or the `DBMS_SQL` package
- Recognize the advantages of using Native Dynamic SQL compared to the `DBMS_SQL` package
- Use `DBMS_METADATA` subprograms to programmatically obtain metadata from the data dictionary

Practice 6: Overview

This practice covers the following topics:

- Creating a package that uses Native Dynamic SQL to create or drop a table and to populate, modify, and delete rows from a table
- Creating a package that compiles the PL/SQL code in your schema
- Using `DBMS_METADATA` to display the statement to regenerate a PL/SQL subprogram

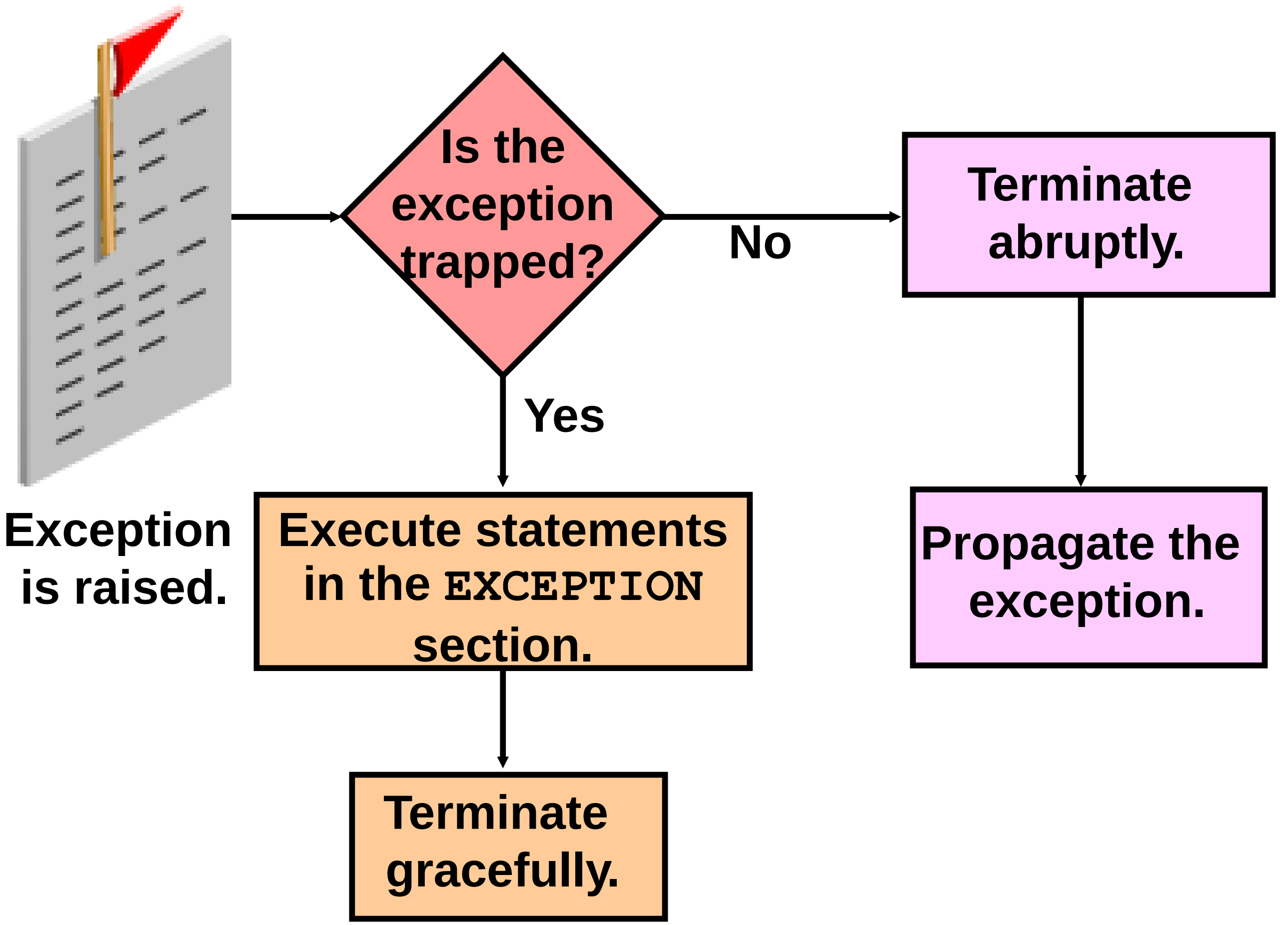
1

Handling Exceptions

Handling Exceptions with PL/SQL

- An exception is a PL/SQL error that is raised during program execution.
- An exception can be raised:
  - Implicitly by the Oracle server
  - Explicitly by the program
- An exception can be handled:
  - By trapping it with a handler
  - By propagating it to the calling environment

Handling Exceptions



Exception Types

- Predefined Oracle server
- Non-predefined Oracle server

Implicitly raised

- User-defined

Explicitly raised

Trapping Exceptions

Syntax:

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

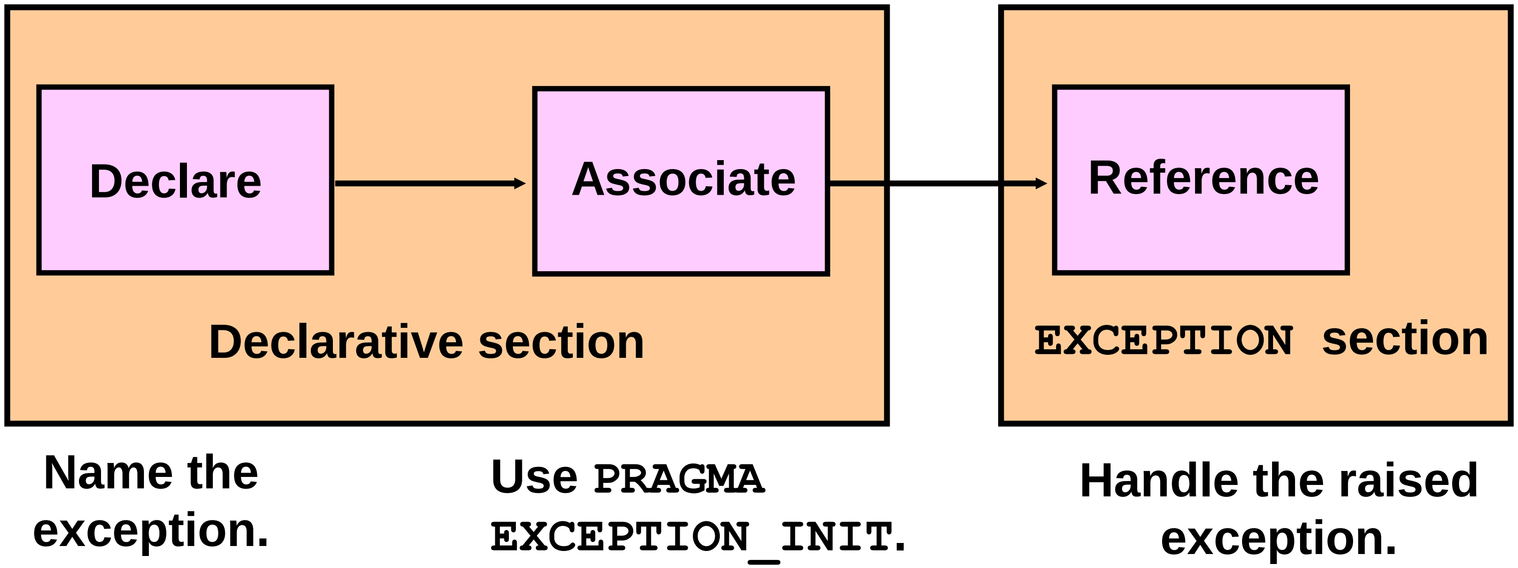
Guidelines for Trapping Exceptions

- The `EXCEPTION` keyword starts the exception handling section.
- Several exception handlers are allowed.
- Only one handler is processed before leaving the block.
- `WHEN OTHERS` is the last clause.

Trapping Predefined Oracle Server Errors

- Reference the predefined name in the exception-handling routine.
- Sample predefined exceptions:
  - `NO_DATA_FOUND`
  - `TOO_MANY_ROWS`
  - `INVALID_CURSOR`
  - `ZERO_DIVIDE`
  - `DUP_VAL_ON_INDEX`

Trapping Non-Predefined Oracle Server Errors



Non-Predefined Error

To trap Oracle server error number –01400 (“cannot insert NULL”):

```
SET SERVEROUTPUT ON
DECLARE
 insert_excep EXCEPTION;
 PRAGMA EXCEPTION_INIT
 (insert_excep, -01400);
BEGIN
 INSERT INTO departments
 (department_id, department_name) VALUES (280, NULL);
EXCEPTION
 WHEN insert_excep THEN
 DBMS_OUTPUT.PUT_LINE('INSERT OPERATION FAILED');
 DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

1
2
3

Functions for Trapping Exceptions

- **`SQLCODE`: Returns the numeric value for the error code**
- **`SQLERRM`: Returns the message associated with the error number**

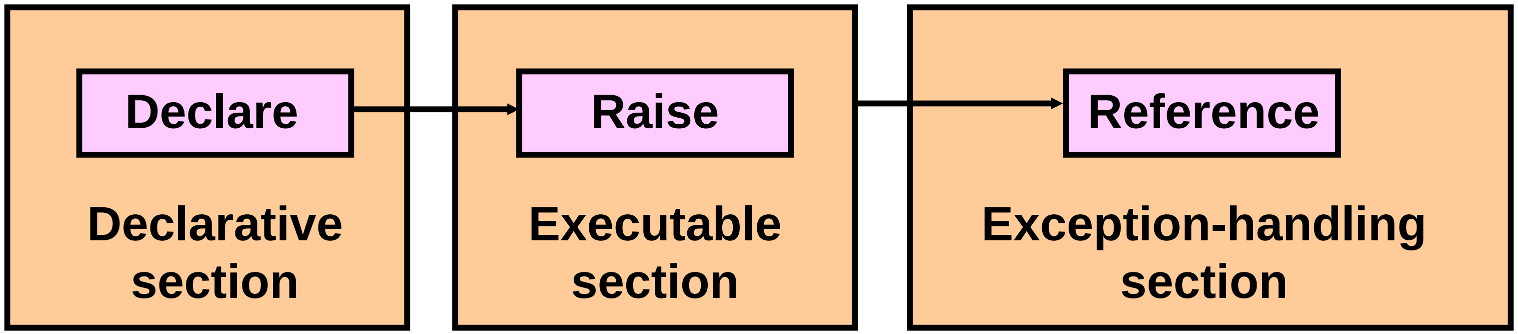
Functions for Trapping Exceptions

Example

```
DECLARE
  error_code      NUMBER;
  error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    error_code := SQLCODE ;
    error_message := SQLERRM ;
   INSERT INTO errors (e_user, e_date, error_code,
   error_message) VALUES(USER,SYSDATE,error_code,
   error_message);
END;
/
```

ORACLE

Trapping User-Defined Exceptions



Name the exception.

Explicitly raise the exception by using the `RAISE` statement.

Handle the raised exception.

Trapping User-Defined Exceptions

```
...
ACCEPT deptno PROMPT 'Please enter the department number:'
ACCEPT name   PROMPT 'Please enter the department name:'
DECLARE
  invalid_department EXCEPTION;
  name VARCHAR2(20):='&name';
  deptno NUMBER :=&deptno;
BEGIN
  UPDATE  departments
  SET     department_name = name
  WHERE   department_id = deptno;
  IF SQL%NOTFOUND THEN
    RAISE invalid_department;
  END IF;
  COMMIT;
EXCEPTION
  WHEN invalid_department  THEN
    DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
/
```

Calling Environments

i**SQL*Plus**	**Displays error number and message to screen**
Procedure Builder	**Displays error number and message to screen**
Oracle Developer Forms	**Accesses error number and message in an `ON-ERROR` trigger by means of the `ERROR_CODE` and `ERROR_TEXT` packaged functions**
Precompiler application	**Accesses exception number through the `SQLCA` data structure**
An enclosing PL/SQL block	**Traps exception in exception-handling routine of enclosing block**

Propagating Exceptions in a Subblock

Subblocks can handle an exception or pass the exception to the enclosing block.

```
DECLARE
  . . .
  no_rows          exception;
  integrity        exception;
  PRAGMA EXCEPTION_INIT (integrity, -2292);
BEGIN
  FOR c_record IN emp_cursor LOOP
    BEGIN
     SELECT ...
     UPDATE ...
     IF SQL%NOTFOUND THEN
       RAISE no_rows;
     END IF;
    END;
  END LOOP;
EXCEPTION
  WHEN integrity THEN ...
  WHEN no_rows THEN ...
END;
/
```

`RAISE_APPLICATION_ERROR` Procedure

Syntax:

```
raise_application_error (error_number,
                  message[, {TRUE | FALSE}]);
```

- **You can use this procedure to issue user-defined error messages from stored subprograms.**
- **You can report errors to your application and avoid returning unhandled exceptions.**

RAISE_APPLICATION_ERROR Procedure

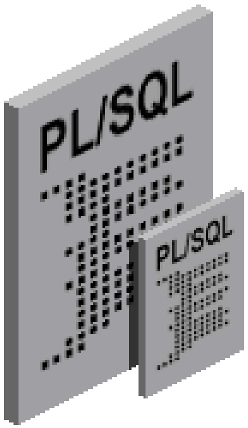
- Used in two different places:
  - Executable section
  - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle server errors

1

Creating Stored Procedures and Functions

Procedures and Functions

- Are named PL/SQL blocks
- Are called PL/SQL subprograms
- Have block structures similar to anonymous blocks:
  - Optional declarative section (without `DECLARE` keyword)
  - Mandatory executable section
  - Optional section to handle exceptions



Differences Between Anonymous Blocks and Subprograms

Anonymous Blocks	Subprograms
Unnamed PL/SQL blocks	Named PL/SQL blocks
Compiled every time	Compiled only once
Not stored in the database	Stored in the database
Cannot be invoked by other applications	Named and therefore can be invoked by other applications
Do not return values	Subprograms called functions must return values.
Cannot take parameters	Can take parameters

Procedure: Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
IS|AS
procedure_body;
```

Invoking the Procedure

```
BEGIN
 add_dept;
END;
/
SELECT department_id, department_name FROM
dept WHERE department_id=280;
```

Inserted 1 row
PL/SQL procedure successfully completed.

DEPARTMENT_ID	DEPARTMENT_NAME
280	ST-Curriculum

Function: Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
 [(argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
function_body;
```

Invoking the Function

```
SET SERVEROUTPUT ON
BEGIN
 IF (check_sal IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal) THEN
 DBMS_OUTPUT.PUT_LINE('Salary > average');
 ELSE
 DBMS_OUTPUT.PUT_LINE('Salary < average');
 END IF;
END;
/
```

```
Salary > average
PL/SQL procedure successfully completed.
```

Passing a Parameter to the Function

```
DROP FUNCTION check_sal;
CREATE FUNCTION check_sal(empno employees.employee_id%TYPE)
RETURN Boolean IS
 dept_id employees.department_id%TYPE;
 sal     employees.salary%TYPE;
 avg_sal employees.salary%TYPE;
BEGIN
 SELECT salary,department_id INTO sal,dept_id
 FROM employees WHERE employee_id=empno;
 SELECT avg(salary) INTO avg_sal FROM employees
 WHERE department_id=dept_id;
 IF sal > avg_sal THEN
  RETURN TRUE;
 ELSE
  RETURN FALSE;
 END IF;
EXCEPTION ...
...
```

Invoking the Function with a Parameter

```
BEGIN
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 205');
 IF (check_sal(205) IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal(205)) THEN
 DBMS_OUTPUT.PUT_LINE('Salary > average');
 ELSE
 DBMS_OUTPUT.PUT_LINE('Salary < average');
 END IF;
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 70');
 IF (check_sal(70) IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal(70)) THEN
 ...
 END IF;
END;
/
```

Practice 4: Overview

This practice covers the following topics:

- Converting an existing anonymous block to a procedure
- Modifying the procedure to accept a parameter
- Writing an anonymous block to invoke the procedure

1

Creating Triggers

ORACLE

Objectives

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

- Describe the different types of triggers
- Describe database triggers and their uses
- Create database triggers
- Describe database trigger-firing rules
- Remove database triggers

Types of Triggers

A trigger:

- Is a PL/SQL block or a PL/SQL procedure associated with a table, view, schema, or database
- Executes implicitly whenever a particular event takes place
- Can be either of the following:
  - Application trigger: Fires whenever an event occurs with a particular application
  - Database trigger: Fires whenever a data event (such as DML) or system event (such as logon or shutdown) occurs on a schema or database

Guidelines for Designing Triggers

- You can design triggers to:
  - Perform related actions
  - Centralize global operations
- You must not design triggers:
  - Where functionality is already built into the Oracle server
  - That duplicate other triggers
- You can create stored procedures and invoke them in a trigger, if the PL/SQL code is very lengthy.
- The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications.

Creating DML Triggers

Create DML statement or row type triggers by using:

```
CREATE [OR REPLACE] TRIGGER trigger_name
 timing
 event1 [OR event2 OR event3]
ON object_name
[[REFERENCING OLD AS old | NEW AS new]
 FOR EACH ROW
 [WHEN (condition)]]
trigger_body
```

- A statement trigger fires once for a DML statement.
- A row trigger fires once for each row affected.

Note: Trigger names must be unique with respect to other triggers in the same schema.

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

ORACLE

Trigger Timing

When should the trigger fire?

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

Note: If multiple triggers are defined for the same object, then the order of firing triggers is arbitrary.

Trigger-Firing Sequence

Use the following firing sequence for a trigger on a table when a single row is manipulated:

DML statement

```
INSERT INTO departments
    (department_id,department_name, location_id)
VALUES (400, 'CONSULTING', 2400);
```

Triggering action

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
...		
400	CONSULTING	2400

→ **BEFORE statement trigger**

→ **BEFORE row trigger**

→ **AFTER row trigger**

→ **AFTER statement trigger**

Trigger-Firing Sequence

Use the following firing sequence for a trigger on a table when many rows are manipulated:

```
UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 30;
```

BEFORE statement trigger

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
114	Raphaely	30
115	Khoo	30
116	Baida	30
117	Tobias	30
118	Himuro	30
119	Colmenares	30

BEFORE row trigger

AFTER row trigger

...

BEFORE row trigger

AFTER row trigger

...

AFTER statement trigger

Trigger Event Types and Body

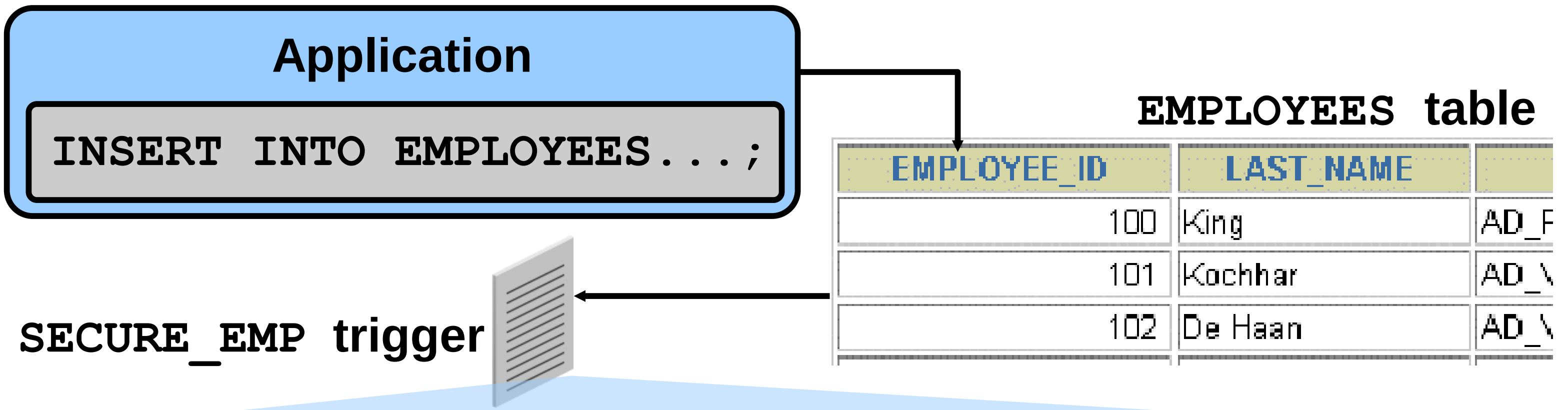
A trigger event:

- **Determines which DML statement causes the trigger to execute**
- **Types are:**
  - `INSERT`
  - `UPDATE [OF column]`
  - `DELETE`

A trigger body:

- **Determines what action is performed**
- **Is a PL/SQL block or a `CALL` to a procedure**

Creating a DML Statement Trigger




```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees BEGIN
 IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
     (TO_CHAR(SYSDATE,'HH24:MI')
           NOT BETWEEN '08:00' AND '18:00') 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);
```

```
INSERT INTO employees (employee_id, last_name, first_name, email,
      *
ERROR at line 1:
ORA-20500: You may insert into EMPLOYEES table only during business hours.
ORA-06512: at "PLSQL.SECURE_EMP", line 4
ORA-04088: error during execution of trigger 'PLSQL.SECURE_EMP'
```

Using Conditional Predicates

```
CREATE OR REPLACE TRIGGER secure_emp BEFORE
INSERT OR UPDATE OR DELETE ON employees BEGIN
 IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
   (TO_CHAR(SYSDATE,'HH24')
       NOT BETWEEN '08' AND '18') 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 restrict_salary
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
  IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
     AND :NEW.salary > 15000 THEN
    RAISE_APPLICATION_ERROR (-20202,
      'Employee cannot earn more than $15,000.');
  END IF;
END;
/
```

Using OLD and NEW Qualifiers

```
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;
/
```

Using `OLD` and `NEW` Qualifiers: Example Using `audit_emp`

```
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	TIMESTAMP	ID	OLD_LAST_N	NEW_LAST_N	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
PLSQL	28-SEP-01			Temp emp		SA_REP		1000
PLSQL	28-SEP-01	999	Temp emp	Smith	SA_REP	SA_REP	1000	2000

Restricting a Row Trigger: Example

```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
 IF INSERTING THEN
   :NEW.commission_pct := 0;
 ELSIF :OLD.commission_pct IS NULL THEN
   :NEW.commission_pct := 0;
 ELSE
   :NEW.commission_pct := :OLD.commission_pct+0.05;
 END IF;
END;
/
```

Summary of Trigger Execution Model

1. Execute all `BEFORE STATEMENT` triggers.
2. Loop for each row affected:
   a. Execute all `BEFORE ROW` triggers.
   b. Execute the DML statement and perform integrity constraint checking.
   c. Execute all `AFTER ROW` triggers.
3. Execute all `AFTER STATEMENT` triggers.

Note: Integrity checking can be deferred until the `COMMIT` operation is performed.

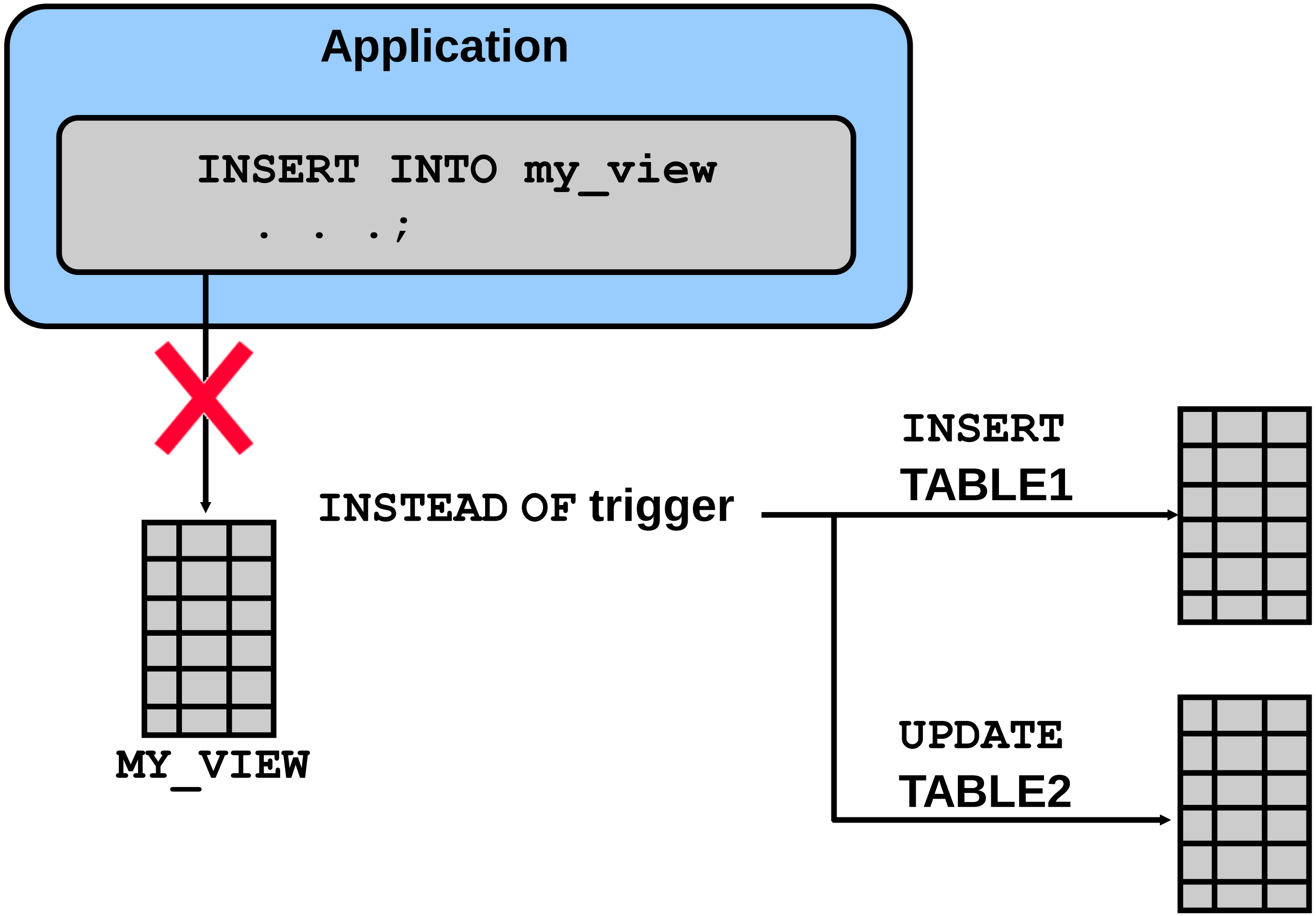
Implementing an Integrity Constraint with a Trigger

```
UPDATE employees SET department_id = 999
 WHERE employee_id = 170;
-- Integrity constraint violation error
```

```
CREATE OR REPLACE TRIGGER employee_dept_fk_trg
AFTER UPDATE OF department_id
ON employees FOR EACH ROW
BEGIN
 INSERT INTO departments VALUES(:new.department_id,
          'Dept '||:new.department_id, NULL, NULL);
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
    NULL; -- mask exception if department exists
END;
/
```

```
UPDATE employees SET department_id = 999
 WHERE employee_id = 170;
-- Successful after trigger is fired
```

INSTEAD OF Triggers



Creating an INSTEAD OF Trigger

Perform the INSERT into EMP_DETAILS that is based on EMPLOYEES and DEPARTMENTS tables:

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

1 INSTEAD OF INSERT into EMP_DETAILS

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90

2 INSERT into NEW_EMPS

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
100	King	24000	90
101	Kochhar	17000	90
102	De Haan	17000	90
...			
9001	ABBOTT	3000	10

3 UPDATE NEW_DEPTS

DEPARTMENT_ID	DEPARTMENT_NAME	DEPT_SA
10	Administration	940
20	Marketing	1900
30	Purchasing	3012
40	Human Resources	650
...		

ORACLE

Creating an INSTEAD OF Trigger

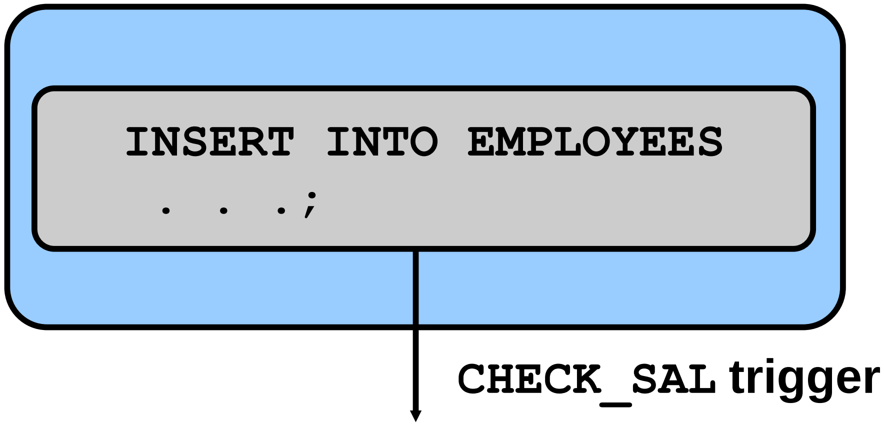
Use INSTEAD OF to perform DML on complex views:

```
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;

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
GROUP BY d.department_id,d.department_name;
```

Comparison of Database Triggers and Stored Procedures

Triggers	Procedures
Defined with `CREATE TRIGGER`	Defined with `CREATE PROCEDURE`
Data dictionary contains source code in `USER_TRIGGERS`.	Data dictionary contains source code in `USER_SOURCE`.
Implicitly invoked by DML	Explicitly invoked
`COMMIT`, `SAVEPOINT`, and `ROLLBACK` are not allowed.	`COMMIT`, `SAVEPOINT`, and `ROLLBACK` are allowed.

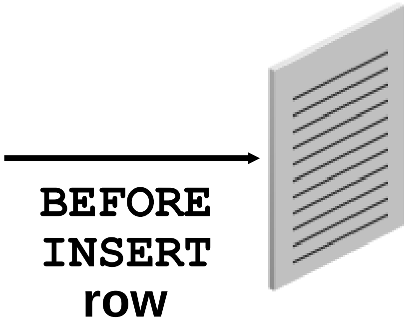
Comparison of Database Triggers and Oracle Forms Triggers



EMPLOYEES table

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
103	Hunold	IT_PROG	9000
104	Ernst	IT_PROG	6000

...



Managing Triggers

- Disable or reenable a database trigger:

```
ALTER TRIGGER trigger_name DISABLE | ENABLE
```

- Disable or reenable all triggers for a table:

```
ALTER TABLE table_name DISABLE | ENABLE
  ALL TRIGGERS
```

- Recompile a trigger for a table:

```
ALTER TRIGGER trigger_name COMPILE
```

Removing Triggers

To remove a trigger from the database, use the `DROP TRIGGER` statement:

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER secure_emp;
```

Note: All triggers on a table are removed when the table is removed.

Testing Triggers

- Test each triggering data operation, as well as nontriggering data operations.
- Test each case of the `WHEN` clause.
- Cause the trigger to fire directly from a basic data operation, as well as indirectly from a procedure.
- Test the effect of the trigger on other triggers.
- Test the effect of other triggers on the trigger.

Summary

In this lesson, you should have learned how to:

- Create database triggers that are invoked by DML operations
- Create statement and row trigger types
- Use database trigger-firing rules
- Enable, disable, and manage database triggers
- Develop a strategy for testing triggers
- Remove database triggers

Practice 10: Overview

This practice covers the following topics:

- **Creating row triggers**
- **Creating a statement trigger**
- **Calling procedures from a trigger**

1

Creating Packages

Objectives

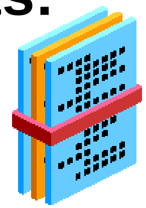
After completing this lesson, you should be able to do the following:

- Describe packages and list their components
- Create a package to group together related variables, cursors, constants, exceptions, procedures, and functions
- Designate a package construct as either public or private
- Invoke a package construct
- Describe the use of a bodiless package

PL/SQL Packages: Overview

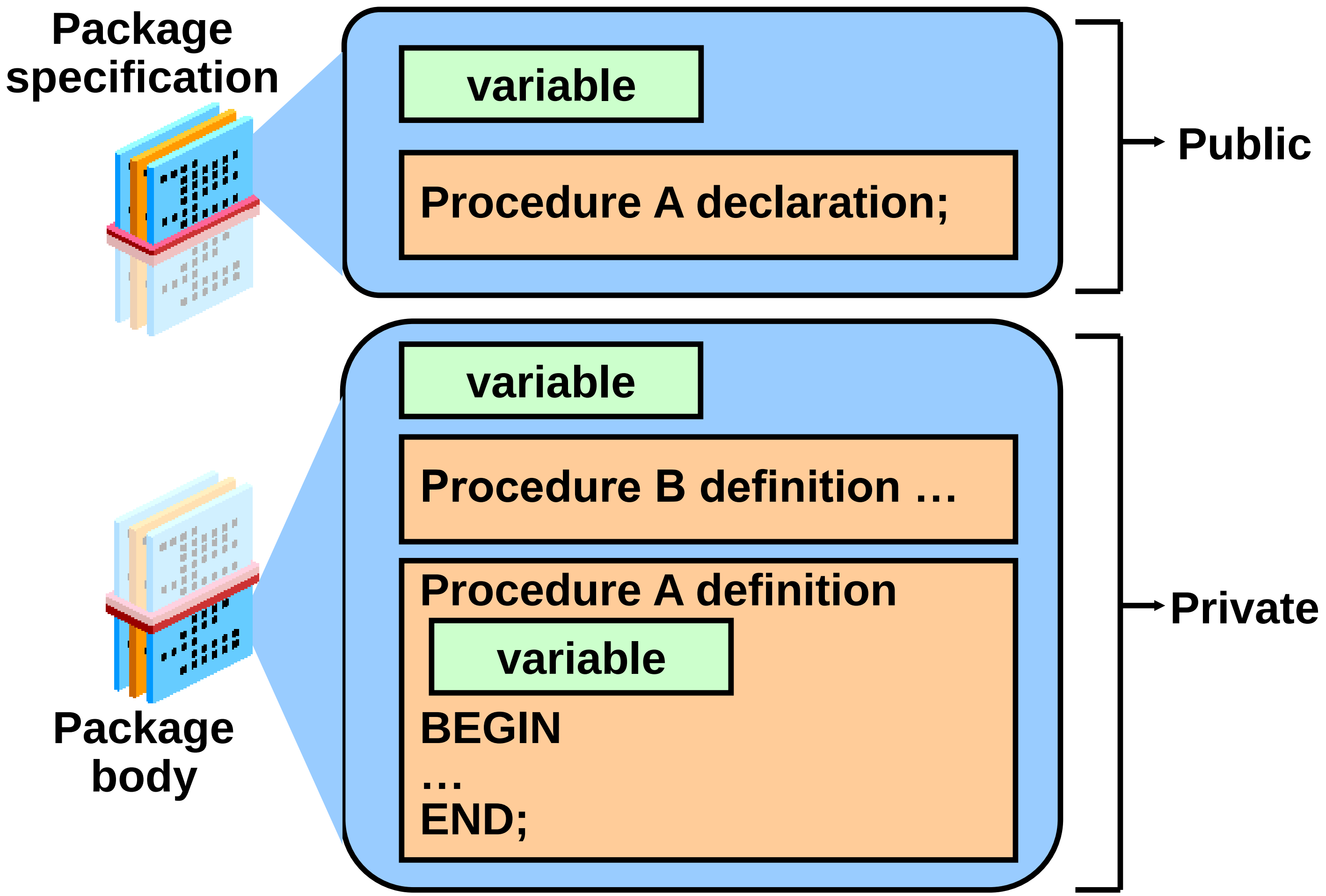
PL/SQL packages:

- Group logically related components:
  - PL/SQL types
  - Variables, data structures, and exceptions
  - Subprograms: procedures and functions
- Consist of two parts:
  - A specification
  - A body

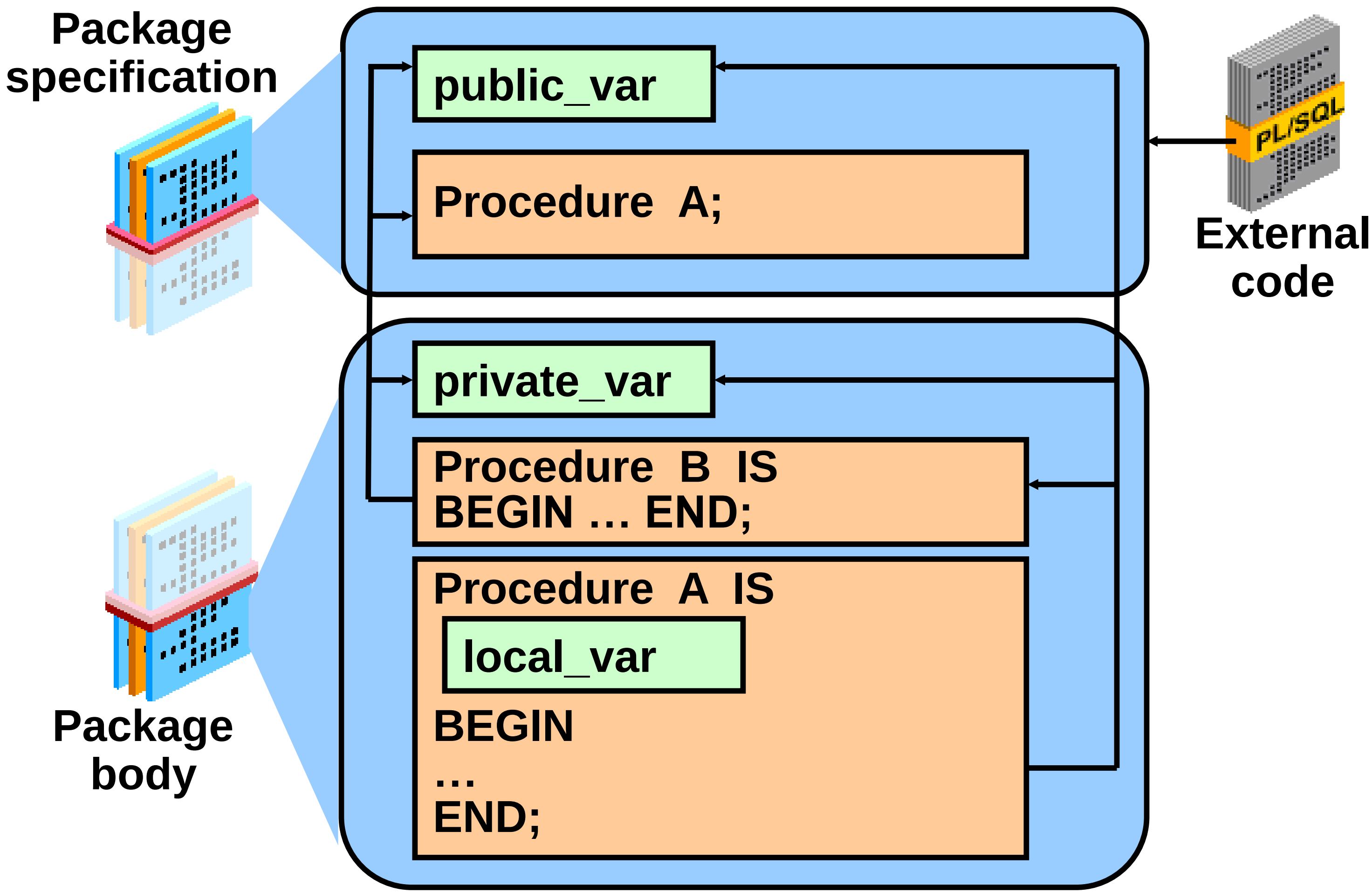


- Enable the Oracle server to read multiple objects into memory at once

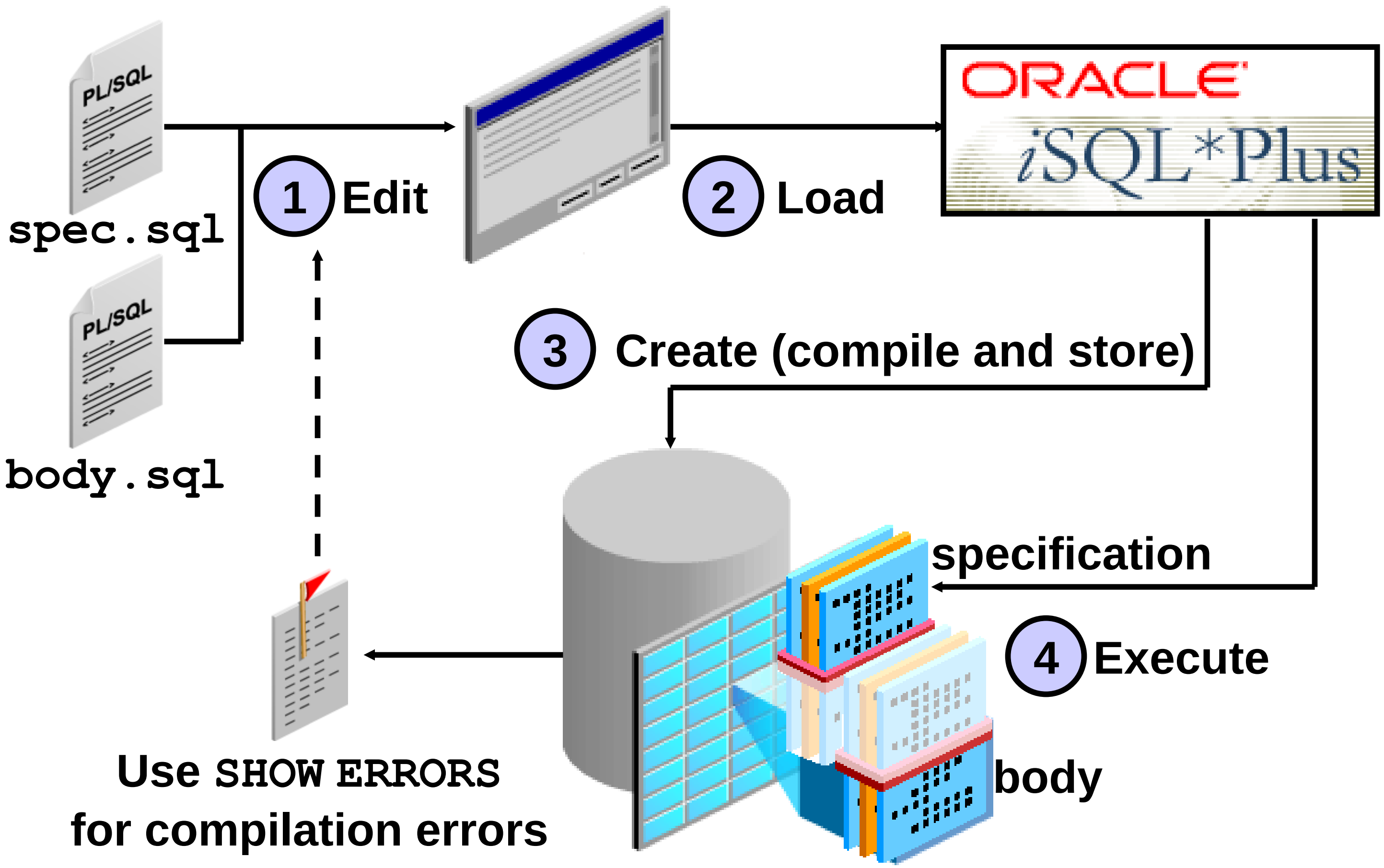
Components of a PL/SQL Package



Visibility of Package Components



Developing PL/SQL Packages



Creating the Package Specification

Syntax:

```
CREATE [OR REPLACE] PACKAGE package_name IS|AS
    public type and variable declarations
    subprogram specifications
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 privileges on the package.**

ORACLE

Example of Package Specification: `comm_pkg`

```
CREATE OR REPLACE PACKAGE comm_pkg IS
  std_comm NUMBER := 0.10;  --initialized to 0.10
  PROCEDURE reset_comm(new_comm NUMBER);
END comm_pkg;
/
```

- **`STD_COMM` is a global variable initialized to `0.10`.**
- **`RESET_COMM` is a public procedure used to reset the standard commission based on some business rules. It is implemented in the package body.**

Creating the Package Body

Syntax:

```
CREATE [OR REPLACE] PACKAGE BODY package_name IS|AS
    private type and 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.
- All private constructs must be declared before they are referenced.
- Public constructs are visible to the package body.

Example of Package Body: `comm_pkg`

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS
  FUNCTION validate(comm NUMBER) RETURN BOOLEAN IS
    max_comm employees.commission_pct%type;
  BEGIN
    SELECT MAX(commission_pct) INTO max_comm
    FROM   employees;
    RETURN (comm BETWEEN 0.0 AND max_comm);
  END validate;
  PROCEDURE reset_comm (new_comm NUMBER) IS BEGIN
    IF validate(new_comm) THEN
      std_comm := new_comm; -- reset public var
    ELSE  RAISE_APPLICATION_ERROR(
            -20210, 'Bad Commission');
    END IF;
  END reset_comm;
END comm_pkg;
```

Invoking Package Subprograms

- Invoke a function within the same package:

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS ...
  PROCEDURE reset_comm(new_comm NUMBER) IS
  BEGIN
    IF validate(new_comm) THEN
      std_comm := new_comm;
    ELSE ...
    END IF;
  END reset_comm;
END comm_pkg;
```

- Invoke a package procedure from *i*SQL*Plus:

```
EXECUTE comm_pkg.reset_comm(0.15)
```

- Invoke a package procedure in a different schema:

```
EXECUTE scott.comm_pkg.reset_comm(0.15)
```

Creating and Using Bodiless Packages

```
CREATE OR REPLACE PACKAGE global_consts IS
  mile_2_kilo    CONSTANT  NUMBER  :=  1.6093;
  kilo_2_mile    CONSTANT  NUMBER  :=  0.6214;
  yard_2_meter   CONSTANT  NUMBER  :=  0.9144;
  meter_2_yard   CONSTANT  NUMBER  :=  1.0936;
END global_consts;
```

```
BEGIN  DBMS_OUTPUT.PUT_LINE('20 miles = ' ||
        20 * global_consts.mile_2_kilo || ' km');
END;
```

```
CREATE FUNCTION mtr2yrd(m NUMBER) RETURN NUMBER IS
BEGIN
  RETURN (m * global_consts.meter_2_yard);
END mtr2yrd;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(mtr2yrd(1))
```

Removing Packages

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

```
DROP PACKAGE package_name;
```

- To remove 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 maintained and is viewable through the `USER_SOURCE` and `ALL_SOURCE` tables in the data dictionary.

- To view the package specification, use:

```
SELECT text
FROM   user_source
WHERE  name = 'COMM_PKG' AND type = 'PACKAGE';
```

- To view the package body, use:

```
SELECT text
FROM   user_source
WHERE  name = 'COMM_PKG' AND type = 'PACKAGE BODY';
```

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.
- Changes to the package specification require recompilation of each referencing subprogram.
- 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.

ORACLE

Advantages of Using Packages

- 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

Summary

In this lesson, you should have learned how to:

- Improve code organization, management, security, and performance by using packages
- Create and remove package specifications and bodies
- Group related procedures and functions together in a package
- Encapsulate the code in a package body
- Define and use components in bodiless packages
- Change a package body without affecting a package specification

Summary

Command	Task
`CREATE [OR REPLACE] PACKAGE`	Create [or modify] an existing package specification
`CREATE [OR REPLACE] PACKAGE BODY`	Create [or modify] an existing package body
`DROP PACKAGE`	Remove both the package specification and the package body
`DROP PACKAGE BODY`	Remove the package body only

Practice 3: Overview

This practice covers the following topics:

- Creating packages
- Invoking package program units

1

Using More Package Concepts

Objectives

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

- Overload package procedures and functions
- Use forward declarations
- Create an initialization block in a package body
- Manage persistent package data states for the life of a session
- Use PL/SQL tables and records in packages
- Wrap source code stored in the data dictionary so that it is not readable

Overloading Subprograms

The overloading feature in PL/SQL:

- Enables you to create two or more subprograms with the same name
- Requires that the subprogram's formal parameters differ in number, order, or data type family
- Enables you to build flexible ways for invoking subprograms with different data
- Provides a way to extend functionality without loss of existing code

Note: Overloading can be done with local subprograms, package subprograms, and type methods, but not with stand-alone subprograms.

Overloading: Example

```
CREATE OR REPLACE PACKAGE dept_pkg IS
  PROCEDURE add_department(deptno NUMBER,
   name VARCHAR2 := 'unknown', loc NUMBER := 1700);
  PROCEDURE add_department(
   name VARCHAR2 := 'unknown', loc NUMBER := 1700);
END dept_pkg;
/
```

Overloading: Example

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

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

Overloading and the `STANDARD` Package

- A package named `STANDARD` defines the PL/SQL environment and built-in functions.
- Most built-in functions are overloaded. An example is the `TO_CHAR` function:

```
FUNCTION TO_CHAR (p1 DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (p2 NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 DATE, P2 VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 NUMBER, P2 VARCHAR2) RETURN
VARCHAR2;
```

- A PL/SQL subprogram with the same name as a built-in subprogram overrides the standard declaration in the local context, unless you qualify the built-in subprogram with its package name.

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;
/
```

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;
```

Package Initialization Block

The block at the end of the package body executes once and is used to initialize public and private package variables.

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

Using Package Functions in SQL and Restrictions

- 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

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

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

Persistent State of Packages

The collection of package variables and the 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 User Global Area (UGA)
  - Unique to each session
  - Subject to change when package subprograms are called or public variables are modified
- Not persistent for the session, but for the life of a subprogram call, when using `PRAGMA SERIALLY_REUSABLE` in the package specification

Persistent State of Package Variables: Example

Time	Events	State for: -Scott- STD	-Scott- MAX	-Jones- STD	-Jones- MAX
9:00	Scott> EXECUTE comm_pkg.reset_comm(0.25)	0.10 0.25	0.4	-	0.4
9:30	Jones> INSERT INTO employees(last_name,commission_pct) VALUES('Madonna', 0.8);	0.25	0.4		0.8
9:35	Jones> EXECUTE comm_pkg.reset_comm (0.5)	0.25	0.4	0.1 0.5	0.8
10:00	Scott> EXECUTE comm_pkg.reset_comm(0.6) Err -20210 'Bad Commission'	0.25	0.4	0.5	0.8
11:00	Jones> ROLLBACK;	0.25	0.4	0.5	0.4
11:01	EXIT ...	0.25	0.4	-	0.4
12:00	EXEC comm_pkg.reset_comm(0.2)	0.25	0.4	0.2	0.4

Persistent State of a Package Cursor

```
CREATE OR REPLACE PACKAGE BODY curs_pkg IS
  CURSOR c IS SELECT employee_id FROM employees;
  PROCEDURE open IS
  BEGIN
    IF NOT c%ISOPEN THEN   OPEN c;  END IF;
  END open;
  FUNCTION next(n NUMBER := 1) RETURN BOOLEAN IS
    emp_id employees.employee_id%TYPE;
  BEGIN
    FOR count IN 1 .. n LOOP
      FETCH c INTO emp_id;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Id: ' ||(emp_id));
    END LOOP;
    RETURN c%FOUND;
  END next;
  PROCEDURE close IS BEGIN
    IF c%ISOPEN THEN  CLOSE c;  END IF;
  END close;
END curs_pkg;
/
```

Executing CURS_PKG

```
SET SERVEROUTPUT ON
EXECUTE curs_pkg.open
DECLARE
  more BOOLEAN := curs_pkg.next(3);
BEGIN
  IF NOT more THEN
    curs_pkg.close;
  END IF;
END;
/
RUN  -- repeats execution on the anonymous block
EXECUTE curs_pkg.close
```

Using PL/SQL Tables of Records in Packages

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE emp_table_type IS TABLE OF employees%ROWTYPE
     INDEX BY BINARY_INTEGER;
  PROCEDURE get_employees(emps OUT emp_table_type);
END emp_pkg;
/
```

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  PROCEDURE get_employees(emps OUT emp_table_type) IS
    i BINARY_INTEGER := 0;
  BEGIN
    FOR emp_record IN (SELECT * FROM employees)
    LOOP
      emps(i) := emp_record;
      i:= i+1;
    END LOOP;
  END get_employees;
END emp_pkg;
/
```

PL/SQL Wrapper

- The PL/SQL wrapper is a stand-alone utility that hides application internals by converting PL/SQL source code into portable object code.
- Wrapping has the following features:
  - Platform independence
  - Dynamic loading
  - Dynamic binding
  - Dependency checking
  - Normal importing and exporting when invoked

Running the Wrapper

The command-line syntax is:

```
WRAP INAME=input_file_name [ONAME=output_file_name]
```

- The `INAME` argument is required.
- The default extension for the input file is `.sql`, unless it is specified with the name.
- The `ONAME` argument is optional.
- The default extension for output file is `.plb`, unless specified with the `ONAME` argument.

Examples:

```
WRAP INAME=student.sql
WRAP INAME=student
WRAP INAME=student.sql ONAME=student.plb
```

Results of Wrapping

- Original PL/SQL source code in input file:

```
CREATE PACKAGE banking IS
  min_bal := 100;
  no_funds EXCEPTION;
...
END banking;
/
```

- Wrapped code in output file:

```
CREATE PACKAGE banking
wrapped

012abc463e ...

/
```

Guidelines for Wrapping

- You must wrap only the package body, not the package specification.
- The wrapper can detect syntactic errors but cannot detect semantic errors.
- The output file should not be edited. You maintain the original source code and wrap again as required.

Summary

In this lesson, you should have learned how to:

- Create and call overloaded subprograms
- Use forward declarations for subprograms
- Write package initialization blocks
- Maintain persistent package state
- Use the PL/SQL wrapper to wrap code

Practice 4: Overview

This practice covers the following topics:

- Using overloaded subprograms
- Creating a package initialization block
- Using a forward declaration
- Using the `WRAP` utility to prevent the source code from being deciphered by humans

Utilizing Oracle-Supplied Packages in Application Development

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

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.

List of Some Oracle-Supplied Packages

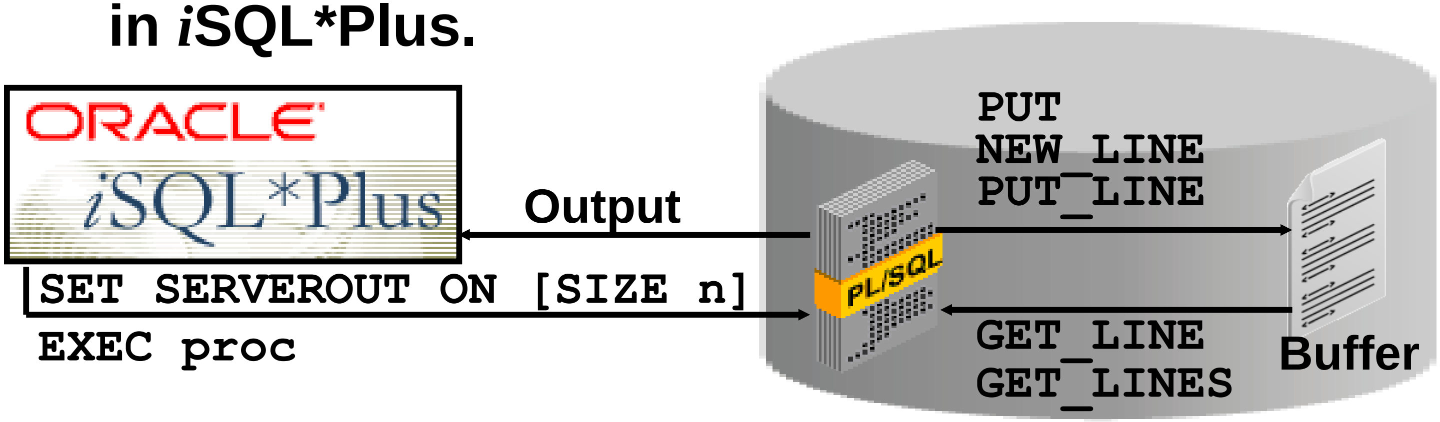
Here is an abbreviated list of some Oracle-supplied packages:

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

How the DBMS_OUTPUT Package Works

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

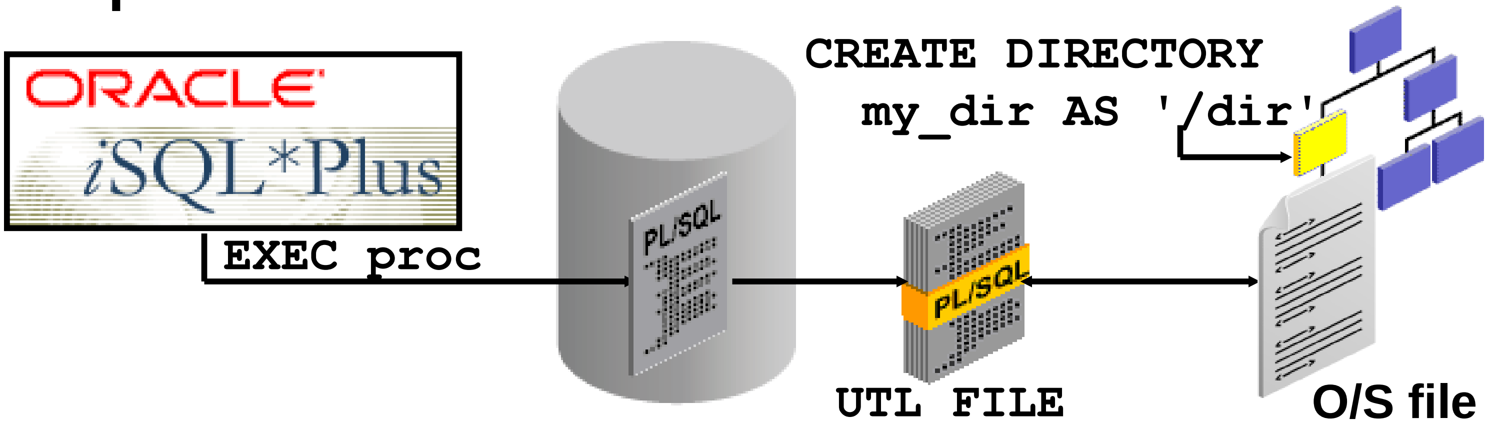
- PUT and PUT_LINE place text in the buffer.
- GET_LINE and GET_LINES read the buffer.
- Messages are not sent until the sender completes.
- Use SET SERVEROUTPUT ON to display messages in *i*SQL*Plus.



Interacting with Operating System Files

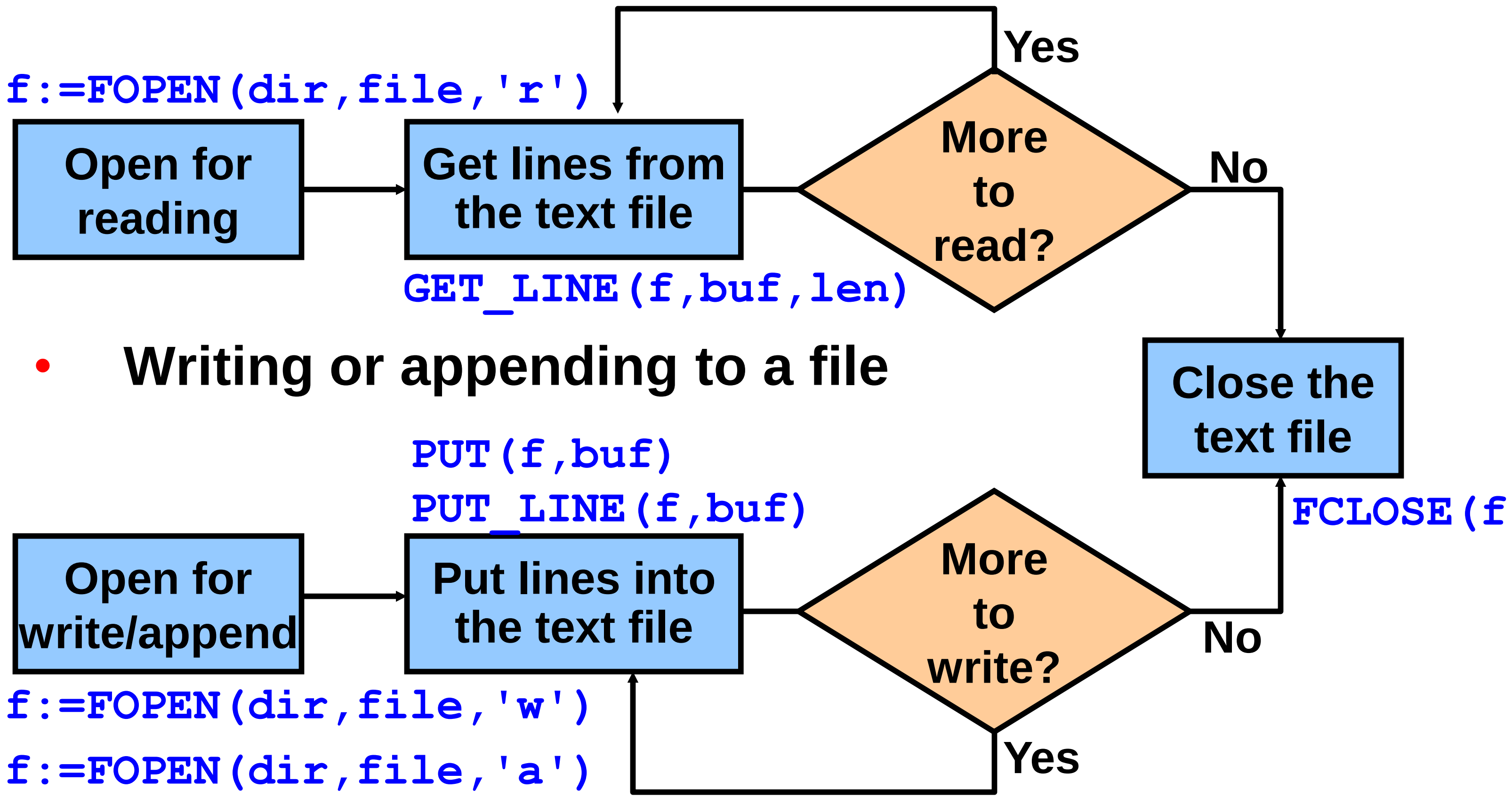
The `UTL_FILE` package extends PL/SQL programs to read and write operating system text files. `UTL_FILE`:

- 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.



File Processing Using the UTL_FILE Package

- Reading a file
- Writing or appending to a file



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`

The other exception not in the `UTL_FILE` package is:

- `NO_DATA_FOUND` and `VALUE_ERROR`

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:

```
PROCEDURE read(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;
```

Using `UTL_FILE`: Example

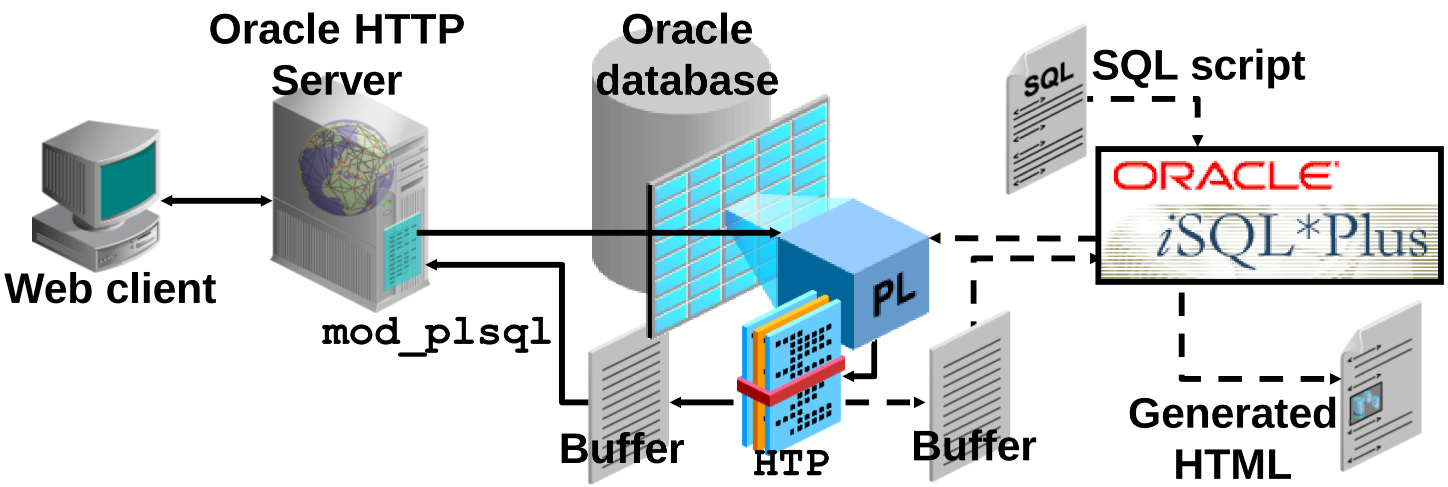
```
CREATE OR REPLACE PROCEDURE sal_status(
  dir IN VARCHAR2, filename IN VARCHAR2) IS
  file UTL_FILE.FILE_TYPE;
 CURSOR empc IS
  SELECT last_name, salary, department_id
  FROM employees ORDER BY department_id;
 newdeptno employees.department_id%TYPE;
 olddeptno employees.department_id%TYPE := 0;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

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);
    END IF;
    UTL_FILE.PUT_LINE (file,
       '  EMPLOYEE: ' || emp_rec.last_name ||
       ' earns: ' || emp_rec.salary);
    olddeptno := emp_rec.department_id;
  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;
/
```

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



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>
```

- Used to create a well-formed HTML document:

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

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.

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

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

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;
/
```

Sending E-Mail with a Text Attachment

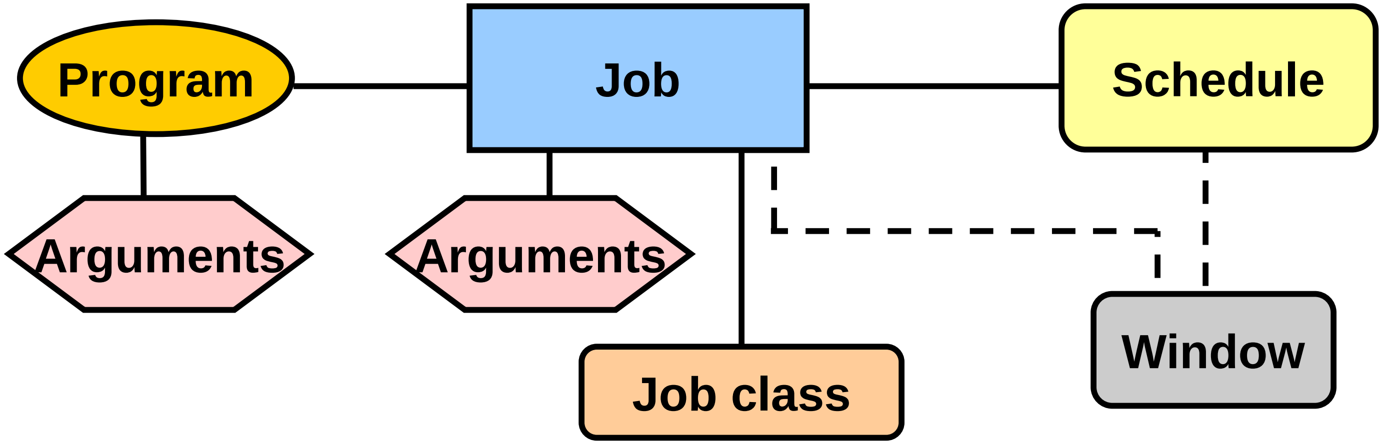
Use the **`UTL_MAIL.SEND_ATTACH_VARCHAR2`** procedure:

```
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;
/
```

`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)



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.

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;
/
```

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 for a Program with Arguments

- Create a program:

```
DBMS_SCHEDULER.CREATE_PROGRAM(
  program_name => 'PROG_NAME',
  program_type => 'STORED_PROCEDURE',
  program_action => 'EMP_REPORT');
```

- Define an argument:

```
DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
  program_name => 'PROG_NAME',
  argument_name => 'DEPT_ID',
  argument_position=> 1, argument_type=> 'NUMBER',
  default_value => '50');
```

- Create a job specifying the number of arguments:

```
DBMS_SCHEDULER.CREATE_JOB('JOB_NAME', program_name
  => 'PROG_NAME', start_date => SYSTIMESTAMP,
  repeat_interval => 'FREQ=DAILY',
  number_of_arguments => 1, enabled => TRUE);
```

ORACLE

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` 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;
```

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)'
```

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;
/
```

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);
```

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**

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

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