Course Agenda

Lessons that are to be covered shortly:

1. Introduction to PL/SQL
2. Declaring PL/SQL Variables
3. Creating the Executable Section
4. Interacting with the Oracle Database Server
5. Writing Control Structures

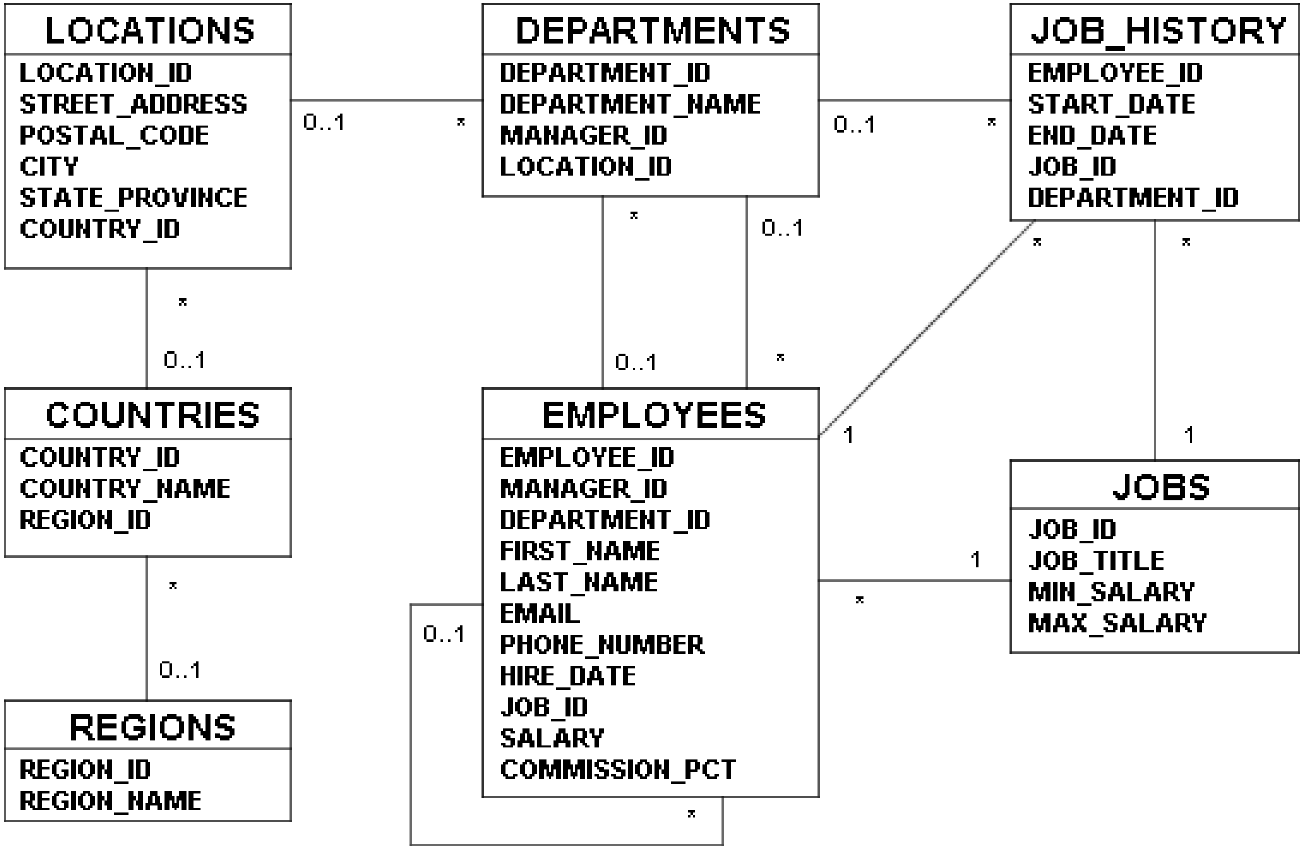
Course Agenda

Lessons that are to be covered in the afternoon:

7. **Using Explicit Cursors**

9. **Creating Stored Procedures and Functions**

The Human Resources (`hr`) Data Set



LOCATIONS
LOCATION_ID
STREET_ADDRESS
POSTAL_CODE
CITY
STATE_PROVINCE
COUNTRY_ID

DEPARTMENTS
DEPARTMENT_ID
DEPARTMENT_NAME
MANAGER_ID
LOCATION_ID

JOB_HISTORY
EMPLOYEE_ID
START_DATE
END_DATE
JOB_ID
DEPARTMENT_ID

COUNTRIES
COUNTRY_ID
COUNTRY_NAME
REGION_ID

EMPLOYEES
EMPLOYEE_ID
MANAGER_ID
DEPARTMENT_ID
FIRST_NAME
LAST_NAME
EMAIL
PHONE_NUMBER
HIRE_DATE
JOB_ID
SALARY
COMMISSION_PCT

JOBS
JOB_ID
JOB_TITLE
MIN_SALARY
MAX_SALARY

REGIONS
REGION_ID
REGION_NAME

PL/SQL Development Environments

The course setup provides the following tools for developing PL/SQL code:

- Oracle SQL Developer (used in this course)
- Oracle SQL*Plus

Oracle SQL Developer

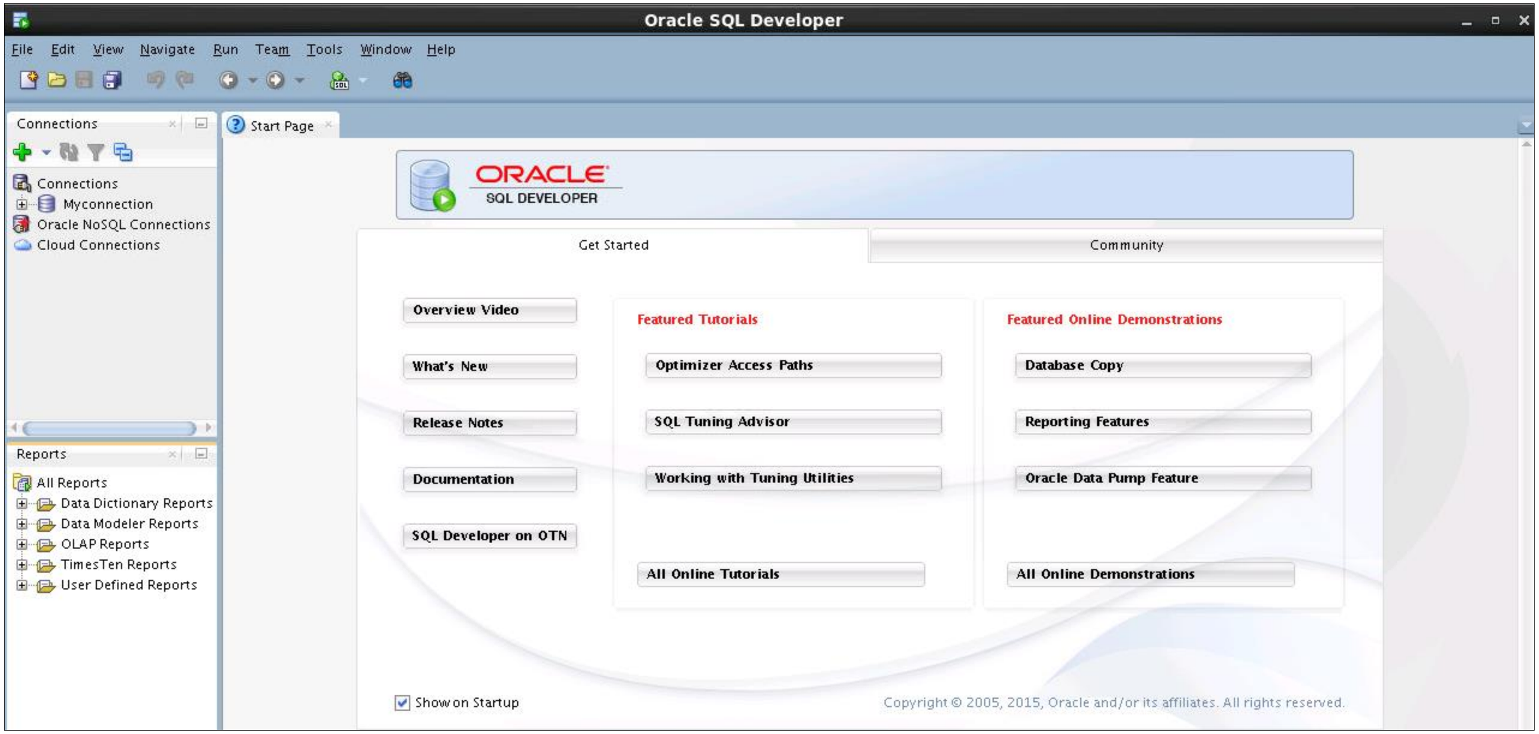
- Oracle SQL Developer is a free graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using standard Oracle database authentication.
- You use SQL Developer in this course.

SQL Developer

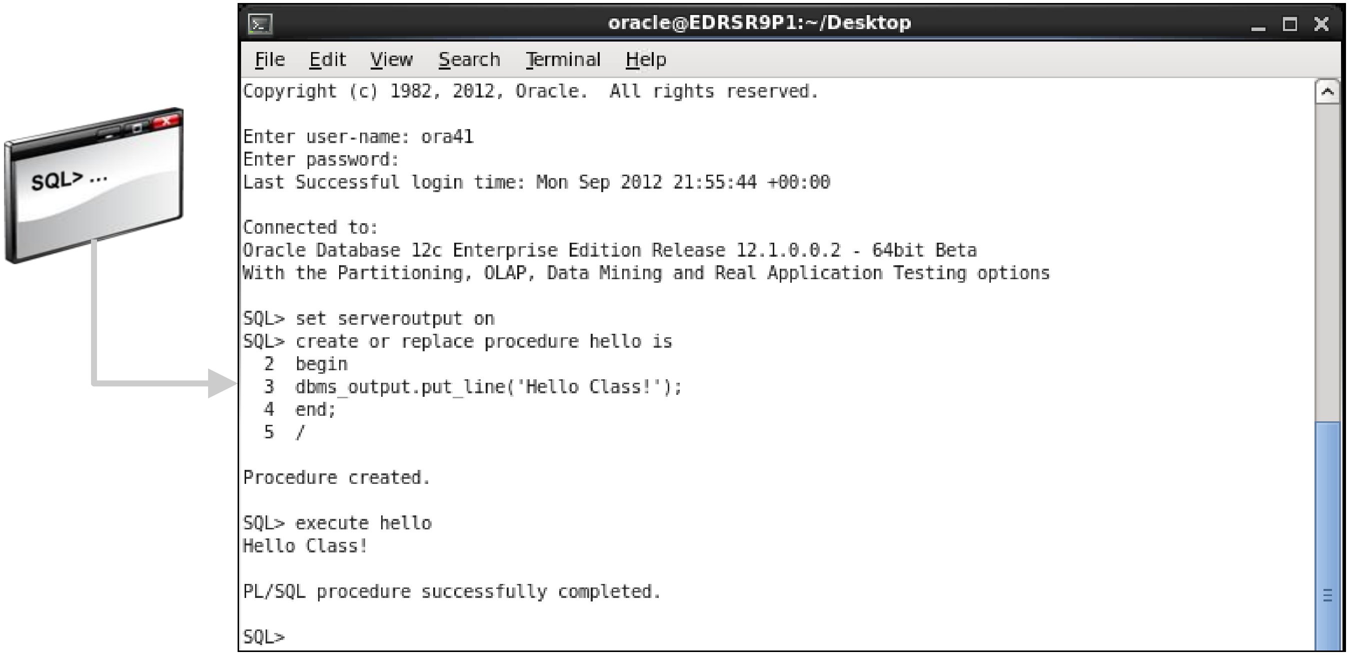
Specifications of SQL Developer

- Is developed in Java
- Supports the Windows, Linux, and Mac OS X
- Enables default connectivity by using the JD driver
- Connects to Oracle Database version 9.2.0.1
- Connects to Oracle Database on Cloud also

SQL Developer 4.1.3 Interface



Coding PL/SQL in SQL*Plus



```
Copyright (c) 1982, 2012, Oracle.  All rights reserved.

Enter user-name: ora41
Enter password:
Last Successful login time: Mon Sep 2012 21:55:44 +00:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.0.2 - 64bit Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> set serveroutput on
SQL> create or replace procedure hello is
  2  begin
  3  dbms_output.put_line('Hello Class!');
  4  end;
  5  /

Procedure created.

SQL> execute hello
Hello Class!

PL/SQL procedure successfully completed.

SQL>
```

Objectives

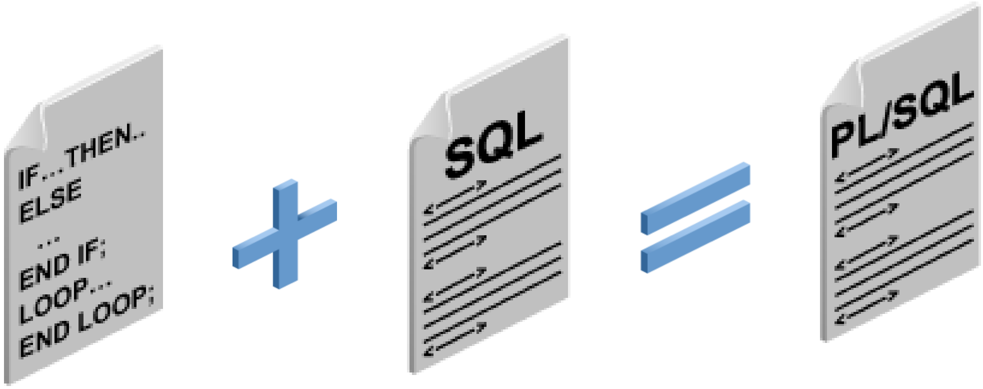
After completing this lesson, you should be able to do the following:

- Explain the need for PL/SQL
- Explain the benefits of PL/SQL
- Identify the different types of PL/SQL blocks
- 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

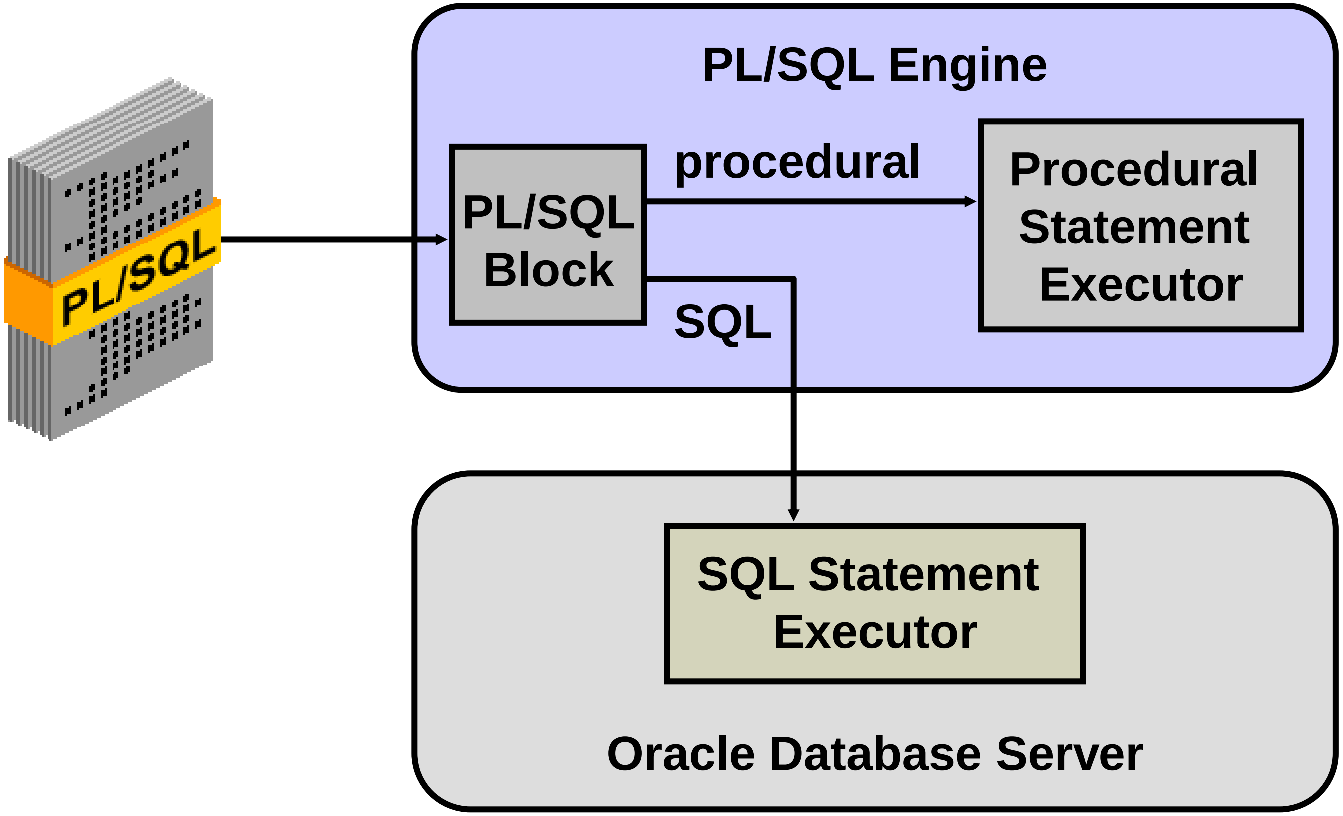


About PL/SQL

PL/SQL:

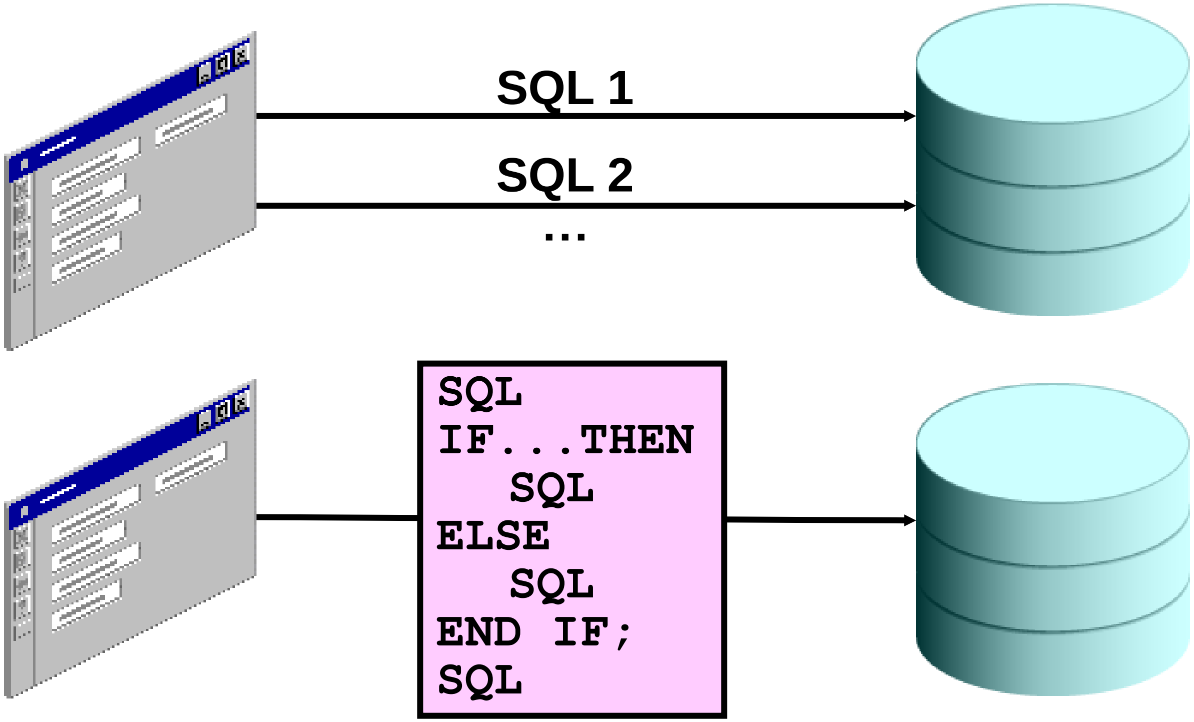
- Provides a block structure for executable units of code. Maintenance of code is made easier with such a well-defined structure.
- Provides procedural constructs such as:
  - Variables, constants, and types
  - Control structures such as conditional statements and loops
  - Reusable program units that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

PL/SQL Environment



Benefits of PL/SQL

- Integration of procedural constructs with SQL
- Improved performance



SQL 1

SQL 2

...

```
SQL
IF...THEN
   SQL
ELSE
   SQL
END IF;
SQL
```

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)**

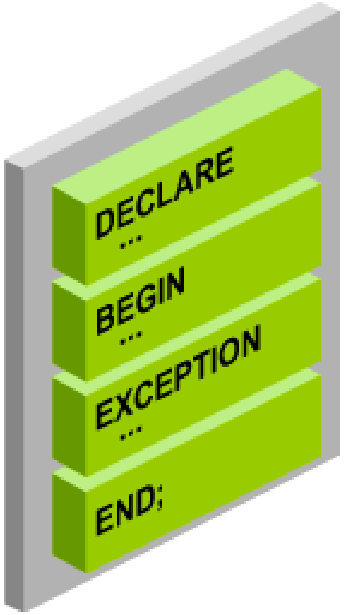
- SQL statements

- 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;
```

I

Declaring PL/SQL Variables

Objectives

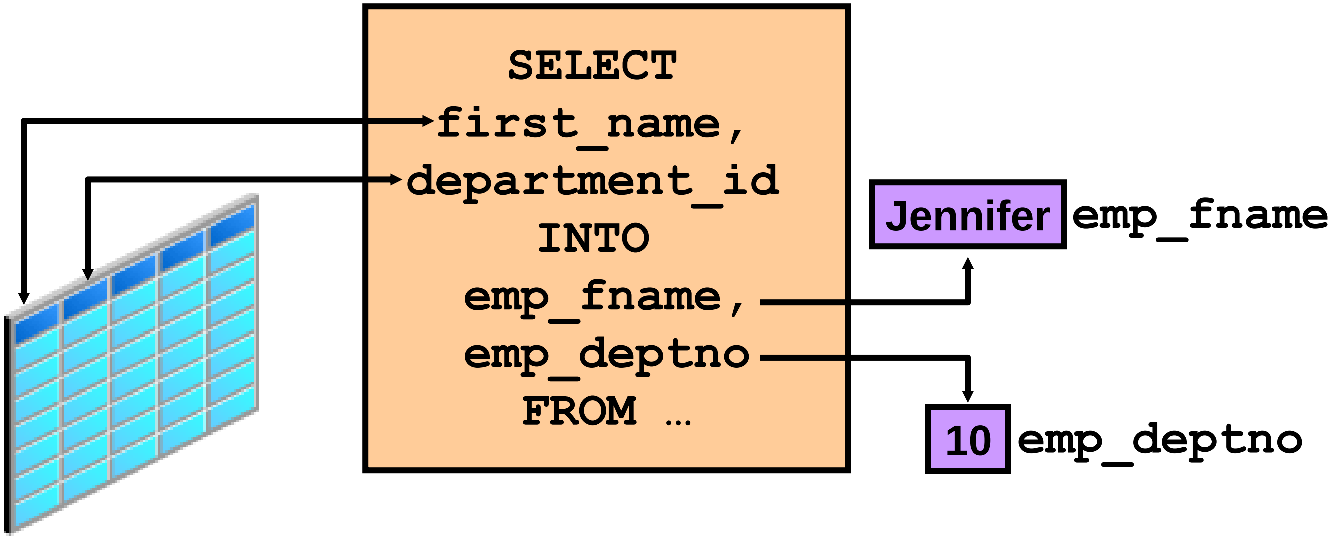
After completing this lesson, you should be able to do the following:

- 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



```
SELECT
first_name,
department_id
INTO
emp_fname,
emp_deptno
FROM ...
```

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;
/
```

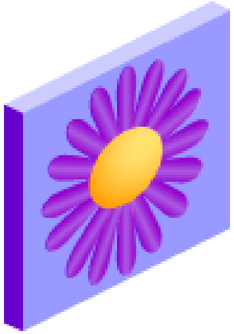
Types of Variables

- 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`

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;
```

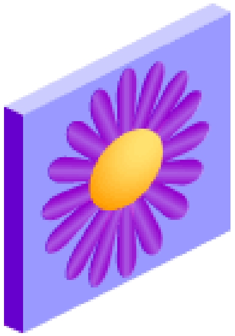
Examples:

```
...
  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.

Composite Data Types

TRUE	23-DEC-98	ATLANTA	

PL/SQL table structure

1	SMITH
2	JONES
3	NANCY
4	TIM

VARCHAR2

PLS_INTEGER

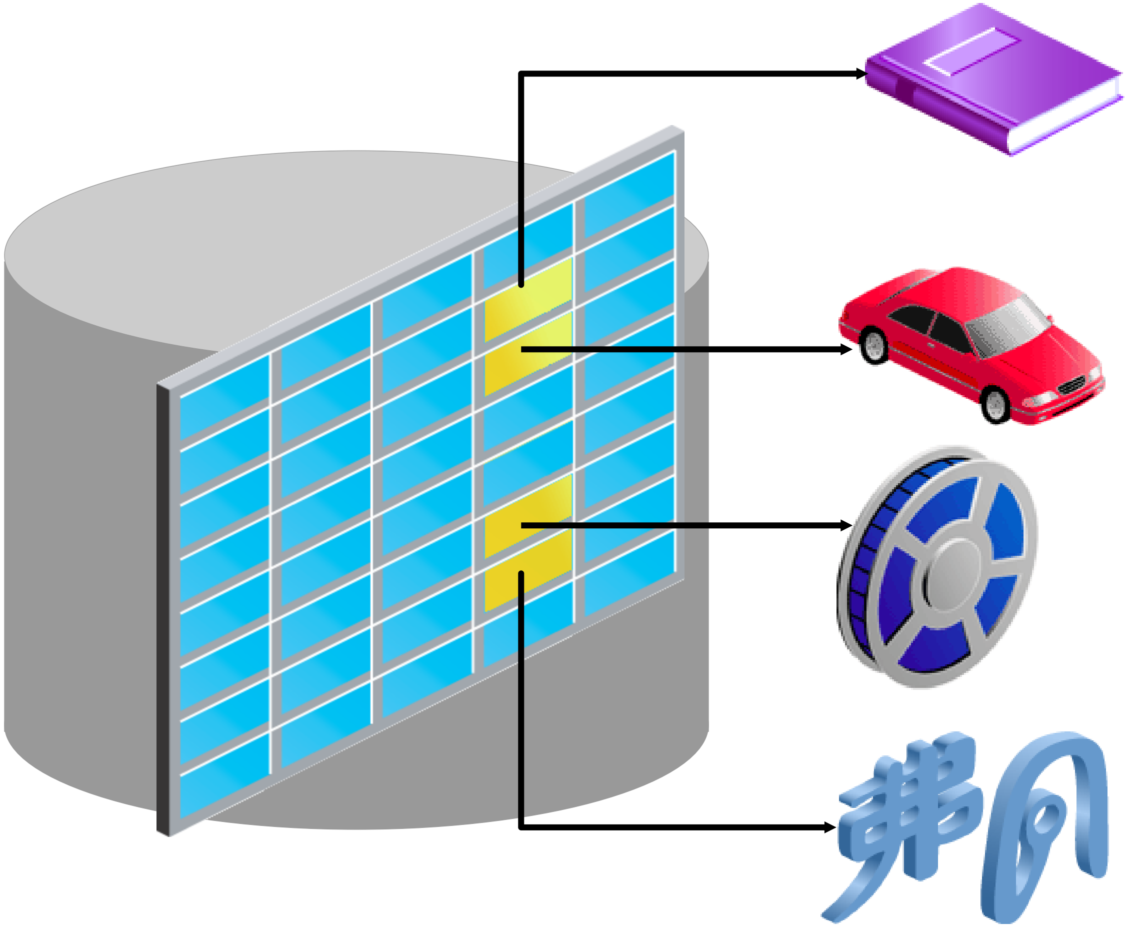
PL/SQL table structure

1	5000
2	2345
3	12
4	3456

NUMBER

PLS_INTEGER

LOB Data Type Variables



Book
(CLOB)

Photo
(BLOB)

Movie
(BFILE)

NCLOB

I

Writing Executable Statements

Objectives

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

- Identify lexical units in a PL/SQL block
- Use built-in SQL functions in PL/SQL
- Describe when implicit conversions take place and when explicit conversions have to be dealt with
- Write nested blocks and qualify variables with labels
- Write readable code with appropriate indentations

PL/SQL Block Syntax and Guidelines

- Literals:
  - Character and date literals must be enclosed in single quotation marks.

    ```
    name := 'Henderson';
    ```

  - Numbers can be simple values or scientific notation.
- Statements can continue over several lines.

Commenting Code

- Prefix single-line comments with two dashes (--).
- Place multiple-line comments between the symbols “/*” and “*/”.

Example:

```
DECLARE
...
annual_sal NUMBER (9,2);
BEGIN    -- Begin the executable section

/* Compute the annual salary based on the
   monthly salary input from the user */
annual_sal := monthly_sal * 12;
END;     -- This is the end of the block
/
```

SQL Functions in PL/SQL

- Available in procedural statements:
  - Single-row number
  - Single-row character
  - Data type conversion
  - Date
  - Timestamp
  - `GREATEST` and `LEAST`
  - Miscellaneous functions
- Not available in procedural statements:
  - `DECODE`
  - Group functions

SQL Functions in PL/SQL: Examples

- Get the length of a string:

```
desc_size INTEGER(5);
prod_description VARCHAR2(70):='You can use this
product with your radios for higher frequency';

-- get the length of the string in prod_description
desc_size:= LENGTH(prod_description);
```

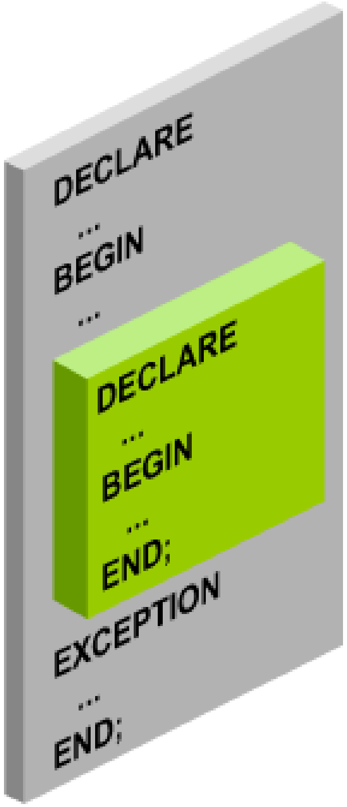
- Convert the employee name to lowercase:

```
emp_name:= LOWER(emp_name);
```

Nested Blocks

PL/SQL blocks can be nested.

- An executable section (`BEGIN` … `END`) can contain nested blocks.
- An exception section can contain nested blocks.



Nested Blocks

Example:

```
DECLARE
 outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
   inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
   DBMS_OUTPUT.PUT_LINE(inner_variable);
   DBMS_OUTPUT.PUT_LINE(outer_variable);
  END;
 DBMS_OUTPUT.PUT_LINE(outer_variable);
END;
/
```

Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations

} Same as in SQL

- Exponential operator (**)

Operators in PL/SQL

Examples:

- Increment the counter for a loop.

```
loop_count := loop_count + 1;
```

- Set the value of a Boolean flag.

```
good_sal := sal BETWEEN 50000 AND 150000;
```

- Validate whether an employee number contains a value.

```
valid  := (empno IS NOT NULL);
```

Programming Guidelines

Make code maintenance easier by:

- Documenting code with comments
- Developing a case convention for the code
- Developing naming conventions for identifiers and other objects
- Enhancing readability by indenting

Indenting Code

For clarity, indent each level of code.

Example:

```
BEGIN
  IF x=0 THEN
     y:=1;
  END IF;
END;
/
```

```
DECLARE
  deptno       NUMBER(4);
  location_id  NUMBER(4);
BEGIN
  SELECT  department_id,
          location_id
  INTO    deptno,
          location_id
  FROM    departments
  WHERE   department_name
          = 'Sales';
...
END;
/
```

Summary

In this lesson, you should have learned how to:

- Use built-in SQL functions in PL/SQL
- Write nested blocks to break logically related functionalities
- Decide when you should perform explicit conversions
- Qualify variables in nested blocks

I

Interacting with the Oracle Server

Objectives

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

- Decide which SQL statements can be directly included in a PL/SQL executable block
- Manipulate data with DML statements in PL/SQL
- Use transaction control statements in PL/SQL
- Make use of the INTO clause to hold the values returned by a SQL statement
- Differentiate between implicit cursors and explicit cursors
- Use SQL cursor attributes

SQL Statements in PL/SQL

- Retrieve a row from the database by using the `SELECT` command.
- Make changes to rows in the database by using DML commands.
- Control a transaction with the `COMMIT`, `ROLLBACK`, or `SAVEPOINT` command.

SELECT Statements in PL/SQL

Retrieve data from the database with a SELECT statement.

Syntax:

```
SELECT   select_list
INTO     {variable_name[, variable_name]...
          | record_name}
FROM     table
[WHERE   condition];
```

SELECT Statements in PL/SQL

- The INTO clause is required.
- Queries must return only one row.

Example:

```
SET SERVEROUTPUT ON
DECLARE
 fname VARCHAR2(25);
BEGIN
 SELECT first_name INTO fname
 FROM employees WHERE employee_id=200;
 DBMS_OUTPUT.PUT_LINE(' First Name is : '||fname);
END;
/
```

Retrieving Data in PL/SQL

Retrieve the `hire_date` and the `salary` for the specified employee.

Example:

```
DECLARE
 emp_hiredate   employees.hire_date%TYPE;
 emp_salary     employees.salary%TYPE;
BEGIN
  SELECT   hire_date, salary
  INTO     emp_hiredate, emp_salary
  FROM     employees
  WHERE    employee_id = 100;
END;
/
```

Retrieving Data in PL/SQL

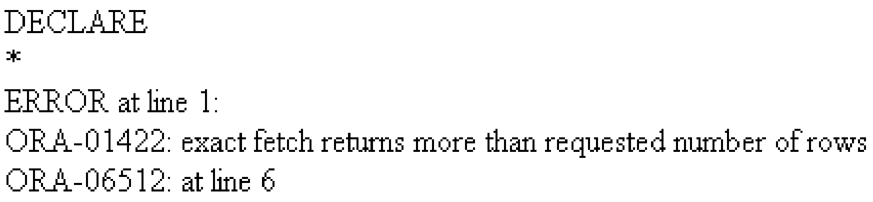
Return the sum of the salaries for all the employees in the specified department.

Example:

```
SET SERVEROUTPUT ON
DECLARE
   sum_sal  NUMBER(10,2);
   deptno   NUMBER NOT NULL := 60;
BEGIN
   SELECT  SUM(salary)  -- group function
   INTO sum_sal FROM employees
   WHERE  department_id = deptno;
   DBMS_OUTPUT.PUT_LINE ('The sum of salary is '
   || sum_sal);
END;
/
```

Naming Conventions

```
DECLARE
  hire_date       employees.hire_date%TYPE;
  sysdate         hire_date%TYPE;
  employee_id     employees.employee_id%TYPE := 176;
BEGIN
  SELECT      hire_date, sysdate
  INTO        hire_date, sysdate
  FROM        employees
  WHERE       employee_id = employee_id;
END;
/
```



```
DECLARE
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
```

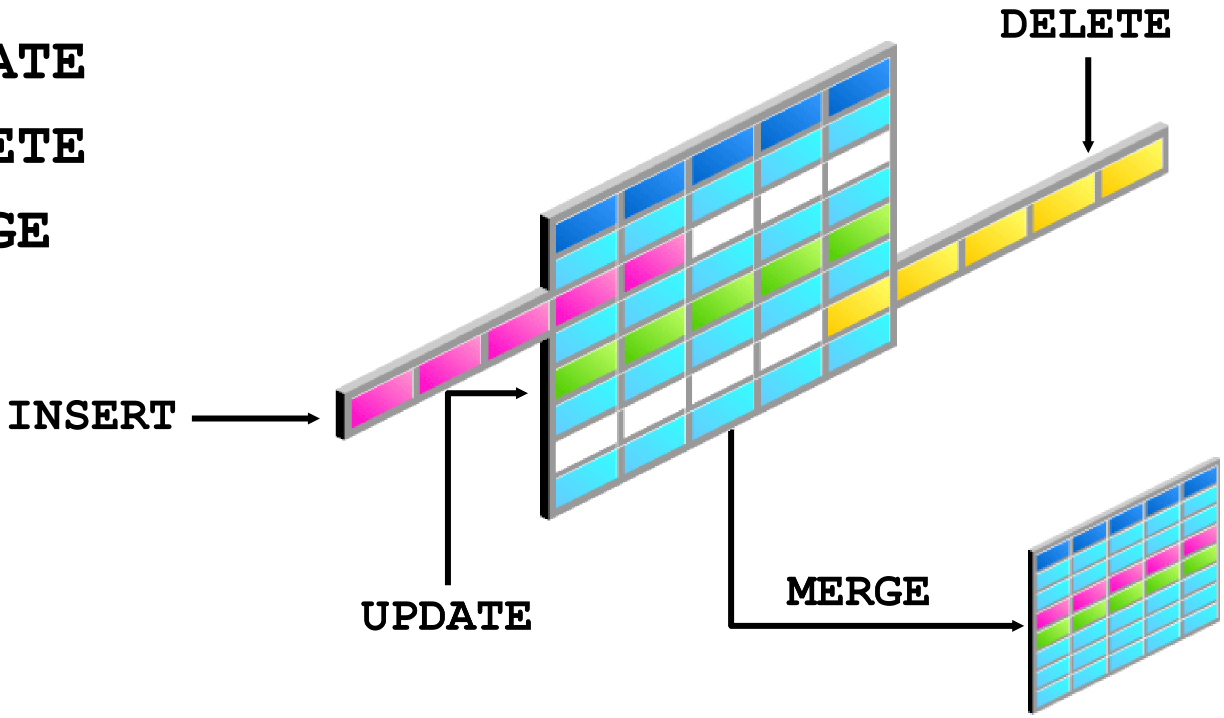
Naming Conventions

- Use a naming convention to avoid ambiguity in the `WHERE` clause.
- Avoid using database column names as identifiers.
- Syntax errors can arise because PL/SQL checks the database first for a column in the table.
- The names of local variables and formal parameters take precedence over the names of database *tables*.
- The names of database table *columns* take precedence over the names of local variables.

Manipulating Data Using PL/SQL

Make changes to database tables by using DML commands:

- `INSERT`
- `UPDATE`
- `DELETE`
- `MERGE`



Inserting Data

Add new employee information to the `EMPLOYEES` table.

Example:

```
BEGIN
 INSERT INTO employees
  (employee_id, first_name, last_name, email,
   hire_date, job_id, salary)
   VALUES(employees_seq.NEXTVAL, 'Ruth', 'Cores',
   'RCORES',sysdate, 'AD_ASST', 4000);
END;
/
```

Updating Data

Increase the salary of all employees who are stock clerks.

Example:

```
DECLARE
  sal_increase   employees.salary%TYPE := 800;
BEGIN
  UPDATE     employees
  SET        salary = salary + sal_increase
  WHERE      job_id = 'ST_CLERK';
END;
/
```

Deleting Data

Delete rows that belong to department 10 from the `employees` table.

Example:

```
DECLARE
  deptno   employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM   employees
  WHERE  department_id = deptno;
END;
/
```

SQL Cursor

- A cursor is a pointer to the private memory area allocated by the Oracle server.
- There are two types of cursors:
  - Implicit cursors: Created and managed internally by the Oracle server to process SQL statements
  - Explicit cursors: Explicitly declared by the programmer

SQL Cursor Attributes for Implicit Cursors

Using SQL cursor attributes, you can test the outcome of your SQL statements.

`SQL%FOUND`	Boolean attribute that evaluates to `TRUE` if the most recent SQL statement returned at least one row.
`SQL%NOTFOUND`	Boolean attribute that evaluates to `TRUE` if the most recent SQL statement did not return even one row.
`SQL%ROWCOUNT`	An integer value that represents number of rows affected by the most recent SQL statement.

SQL Cursor Attributes for Implicit Cursors

Delete rows that have the specified employee ID from the `employees` table. Print the number of rows deleted.

Example:

```
VARIABLE rows_deleted VARCHAR2(30)
DECLARE
  empno employees.employee_id%TYPE := 176;
BEGIN
  DELETE FROM  employees
  WHERE employee_id = empno;
  :rows_deleted := (SQL%ROWCOUNT ||
                       ' row deleted.');
END;
/
PRINT rows_deleted
```

Summary

In this lesson, you should have learned how to:

- Embed DML statements, transaction control statements, and DDL statements in PL/SQL
- Use the `INTO` clause, which is mandatory for all `SELECT` statements in PL/SQL
- Differentiate between implicit cursors and explicit cursors
- Use SQL cursor attributes to determine the outcome of SQL statements

I

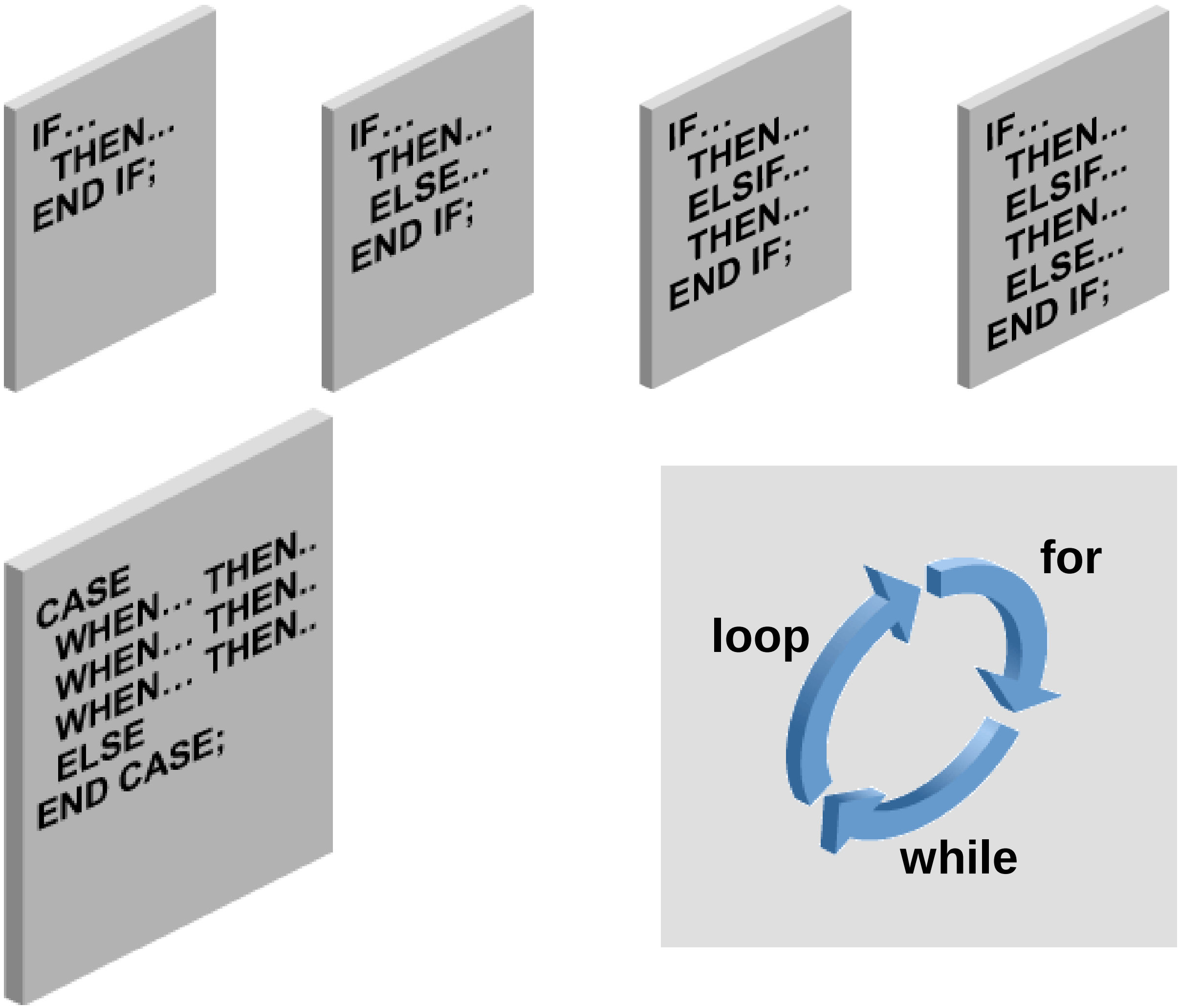
Writing Control Structures

Objectives

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

- Identify the uses and types of control structures
- Construct an `IF` statement
- Use `CASE` statements and `CASE` expressions
- Construct and identify different loop statements
- Make use of guidelines while using the conditional control structures

Controlling Flow of Execution



```
IF...
  THEN...
END IF;
```

```
IF...
  THEN...
  ELSE...
END IF;
```

```
IF...
  THEN...
  ELSIF...
  THEN...
END IF;
```

```
IF...
  THEN...
  ELSIF...
  THEN...
  ELSE...
END IF;
```

```
CASE
  WHEN... THEN..
  WHEN... THEN..
  WHEN... THEN..
  ELSE
END CASE;
```

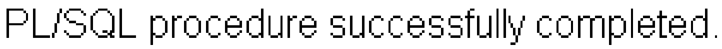
IF Statements

Syntax:

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Simple `IF` Statement

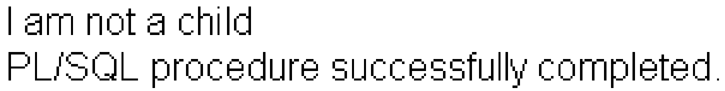
```
DECLARE
  myage number:=31;
BEGIN
  IF myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  END IF;
END;
/
```



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

IF THEN ELSE Statement

```
SET SERVEROUTPUT ON
DECLARE
myage number:=31;
BEGIN
IF myage < 11
 THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
 ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
END IF;
END;
/
```



```
I am not a child
PL/SQL procedure successfully completed.
```

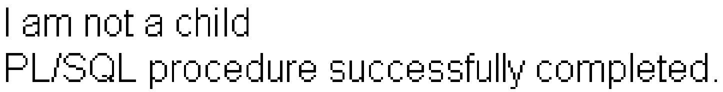
IF ELSIF ELSE Clause

```
DECLARE
myage number:=31;
BEGIN
IF myage < 11
 THEN
       DBMS_OUTPUT.PUT_LINE(' I am a child ');
   ELSIF myage < 20
     THEN
       DBMS_OUTPUT.PUT_LINE(' I am young ');
   ELSIF myage < 30
     THEN
       DBMS_OUTPUT.PUT_LINE(' I am in my twenties');
   ELSIF myage < 40
     THEN
       DBMS_OUTPUT.PUT_LINE(' I am in my thirties');
 ELSE
    DBMS_OUTPUT.PUT_LINE(' I am always young ');
END IF;
END;
/
```

```
I am in my thirties
PL/SQL procedure successfully completed.
```

NULL Values in IF Statements

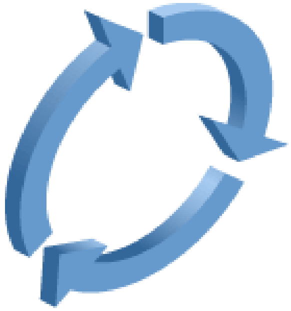
```
DECLARE
myage number;
BEGIN
IF myage < 11
 THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
 ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
END IF;
END;
/
```



```
I am not a child
PL/SQL procedure successfully completed.
```

Iterative Control: `LOOP` Statements

- Loops repeat a statement or sequence of statements multiple times.
- There are three loop types:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop



Basic Loops

Syntax:

```
LOOP
  statement1;
  . . .
  EXIT [WHEN condition];
END LOOP;
```

Basic Loops

Example:

```
DECLARE
  countryid     locations.country_id%TYPE := 'CA';
  loc_id        locations.location_id%TYPE;
  counter        NUMBER(2) := 1;
  new_city      locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO loc_id FROM locations
  WHERE country_id = countryid;
  LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((loc_id + counter), new_city, countryid);
    counter := counter + 1;
    EXIT WHEN counter > 3;
  END LOOP;
END;
/
```

WHILE Loops

Syntax:

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Use the WHILE loop to repeat statements while a condition is TRUE.

WHILE Loops

Example:

```
DECLARE
  countryid   locations.country_id%TYPE := 'CA';
  loc_id      locations.location_id%TYPE;
  new_city    locations.city%TYPE := 'Montreal';
  counter     NUMBER := 1;
BEGIN
  SELECT MAX(location_id) INTO loc_id FROM locations
  WHERE country_id = countryid;
  WHILE counter <= 3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((loc_id + counter), new_city, countryid);
    counter := counter + 1;
  END LOOP;
END;
/
```

FOR Loops

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared implicitly.
- 'lower_bound .. upper_bound' is required syntax.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

FOR Loops

Example:

```
DECLARE
  countryid   locations.country_id%TYPE := 'CA';
  loc_id      locations.location_id%TYPE;
  new_city    locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO loc_id
    FROM locations
    WHERE country_id = countryid;
  FOR i IN 1..3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((loc_id + i), new_city, countryid );
  END LOOP;
END;
/
```

`FOR` Loops

Guidelines

- **Reference the counter within the loop only; it is undefined outside the loop.**
- **Do not reference the counter as the target of an assignment.**
- **Neither loop bound should be `NULL`.**

Guidelines While Using Loops

- Use the basic loop when the statements inside the loop must execute at least once.
- Use the `WHILE` loop if the condition has to be evaluated at the start of each iteration.
- Use a `FOR` loop if the number of iterations is known.

Summary

In this lesson, you should have learned how to:

Change the logical flow of statements by using the following control structures.

- Conditional (`IF` statement)
- `CASE` expressions and `CASE` statements
- Loops:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop
- `EXIT` statements

I

Using Explicit Cursors

Objectives

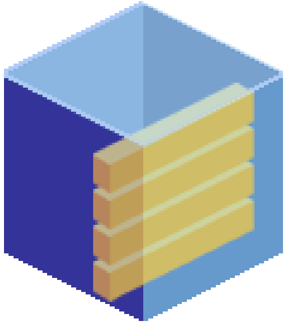
After completing this lesson, you should be able to do the following:

- Distinguish between an implicit and an explicit cursor
- Discuss when and why to use an explicit cursor
- Declare and control explicit cursors
- Use simple loop and cursor `FOR` loop to fetch data
- Declare and use cursors with parameters
- Lock rows using the `FOR UPDATE` clause
- Reference the current row with the `WHERE CURRENT` clause

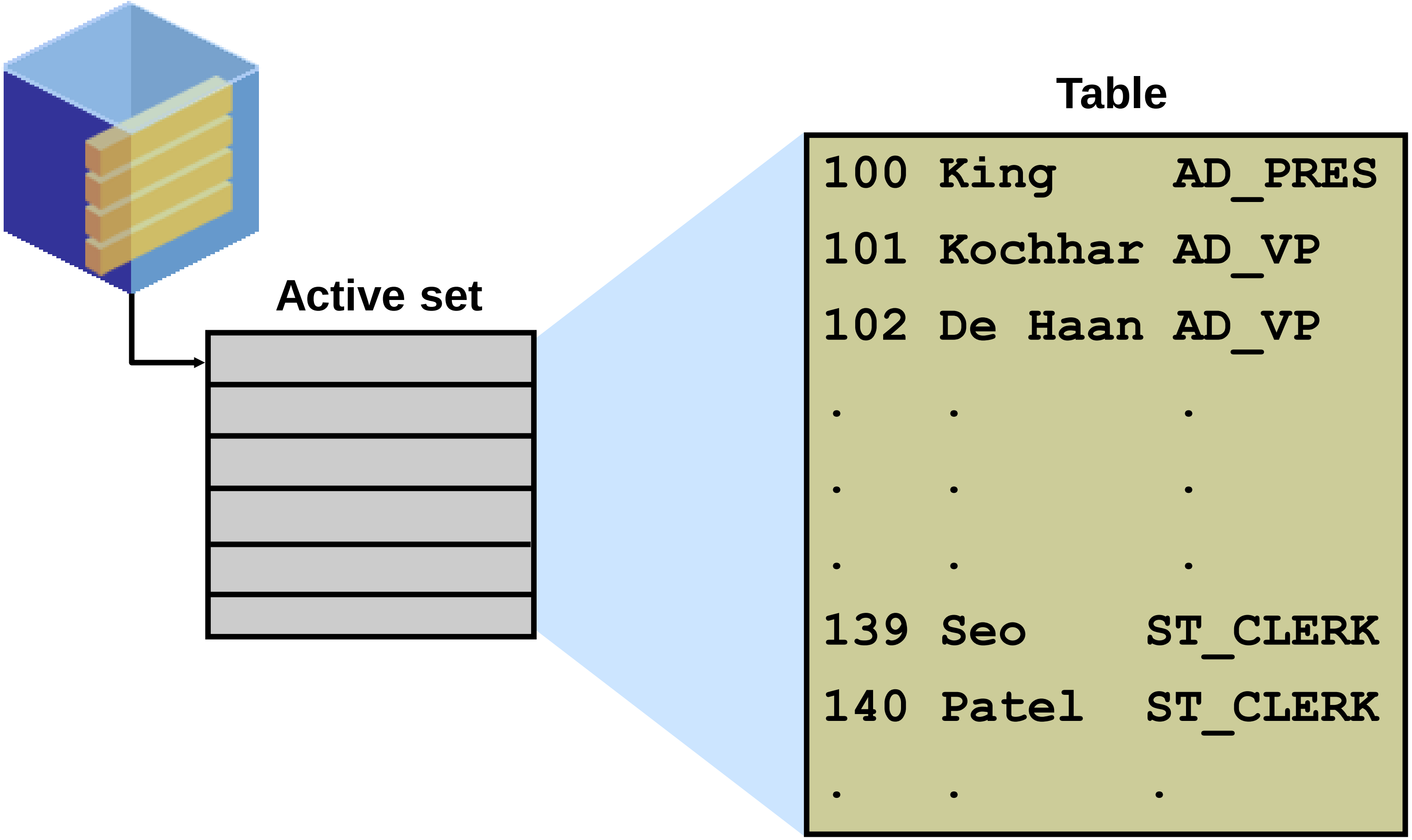
About Cursors

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

- Implicit cursors: Declared and managed by PL/SQL for all DML and PL/SQL `SELECT` statements
- Explicit cursors: Declared and managed by the programmer

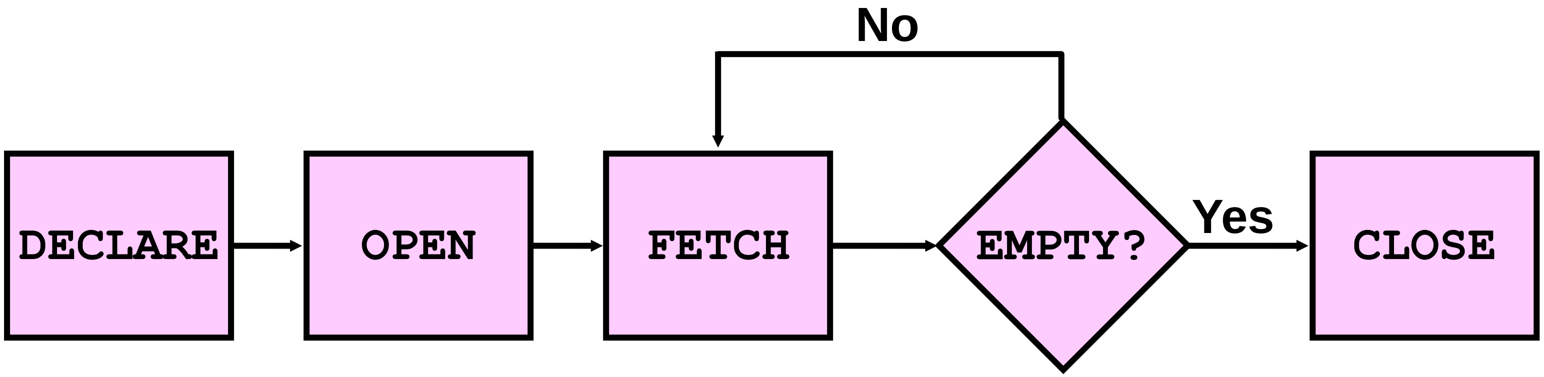


Explicit Cursor Operations



100	King	AD_PRES
101	Kochhar	AD_VP
102	De Haan	AD_VP
.	.	.
.	.	.
.	.	.
139	Seo	ST_CLERK
140	Patel	ST_CLERK
.	.	.

Controlling Explicit Cursors



No

`DECLARE` `OPEN` `FETCH` `EMPTY?` Yes `CLOSE`

- Create a named SQL area

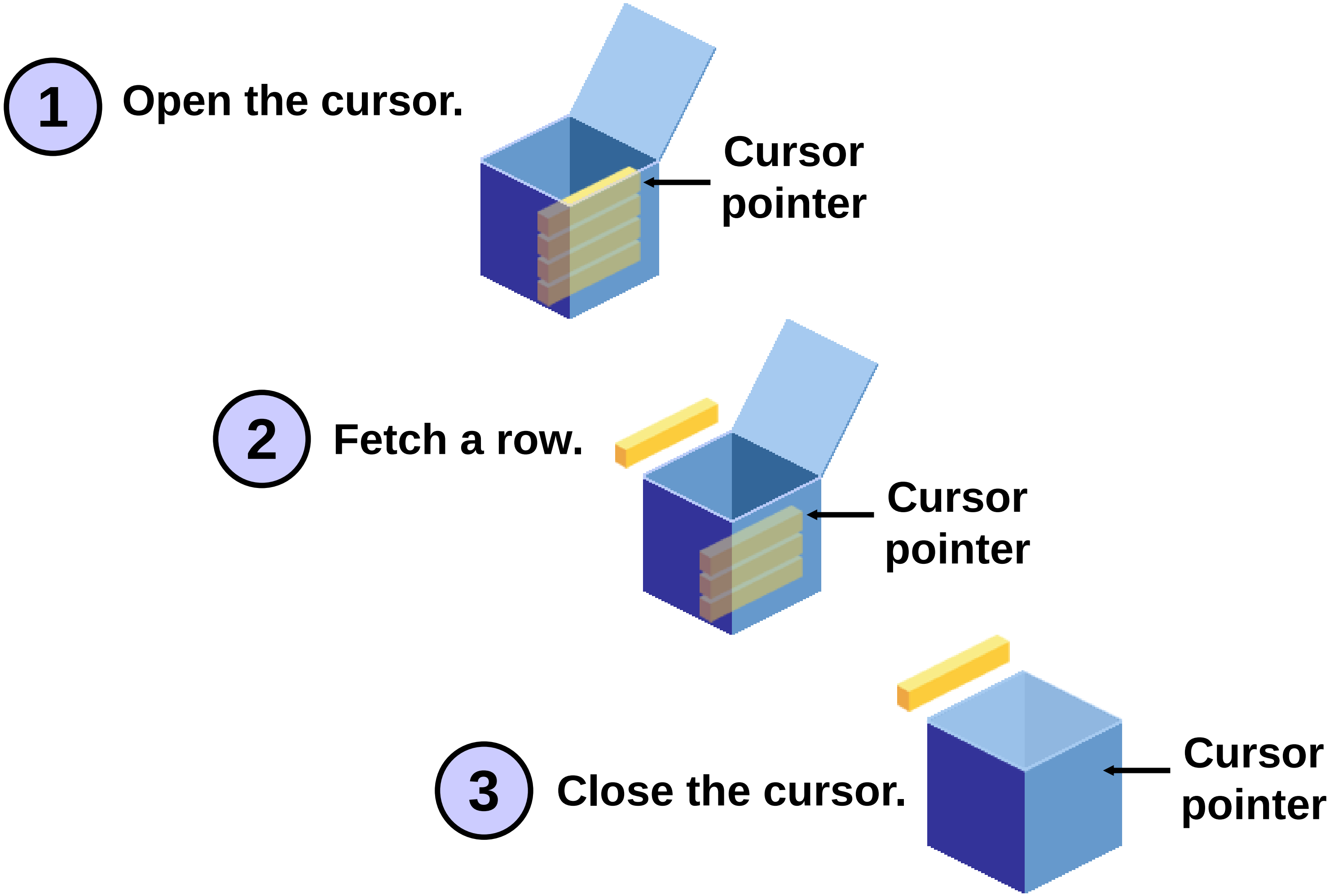
- Identify the active set

- 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.

Declaring the Cursor

Syntax:

```
CURSOR cursor_name IS
     select_statement;
```

Examples:

```
DECLARE
  CURSOR emp_cursor IS
  SELECT employee_id, last_name FROM employees
  WHERE department_id =30;
```

```
DECLARE
  locid NUMBER:= 1700;
  CURSOR dept_cursor IS
  SELECT * FROM departments
  WHERE location_id = locid;
...
```

Opening the Cursor

```
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
...
BEGIN
  OPEN emp_cursor;
```

Fetching Data from the Cursor

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
  empno employees.employee_id%TYPE;
  lname employees.last_name%TYPE;
BEGIN
  OPEN emp_cursor;
  FETCH emp_cursor INTO empno, lname;
  DBMS_OUTPUT.PUT_LINE( empno ||' '||lname);
  ...
END;
/
```

Fetching Data from the Cursor

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE  department_id =30;
  empno employees.employee_id%TYPE;
  lname employees.last_name%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO empno, lname;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( empno ||' '||lname);
  END LOOP;
  ...
END;
/
```

Closing the Cursor

```
...
  LOOP
    FETCH emp_cursor INTO empno, lname;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( empno ||' '||lname);
  END LOOP;
 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

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
BEGIN
   FOR emp_record IN emp_cursor
    LOOP
     DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
     ||' ' ||emp_record.last_name);
    END LOOP;
END;
/
```

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

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...
```

Example of %ROWCOUNT and %NOTFOUND

```
SET SERVEROUTPUT ON
DECLARE
  empno  employees.employee_id%TYPE;
  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 empno, ename;
   EXIT WHEN emp_cursor%ROWCOUNT > 10 OR
                     emp_cursor%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(empno)
                       ||' '|| ename);
  END LOOP;
  CLOSE emp_cursor;
END ;
/
```

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.

```
UPDATE employees
   SET     salary = ...
   WHERE CURRENT OF emp_cursor;
```

Summary

In this lesson, you should have learned how 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
- Create and handle explicit cursors
- Use simple loops and cursor `FOR` loops to handle multiple rows in the cursors
- Evaluate the cursor status by using the cursor attributes
- Use the `FOR UPDATE` and `WHERE CURRENT OF` clauses to update or delete the current fetched row

I

Creating Stored Procedures and Functions

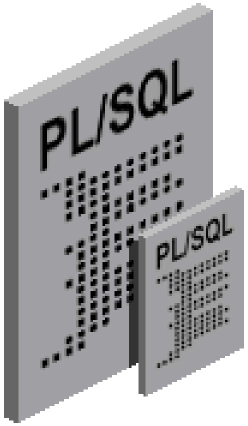
Objectives

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

- Differentiate between anonymous blocks and subprograms
- Create a simple procedure and invoke it from an anonymous block
- Create a simple function
- Create a simple function that accepts a parameter
- Differentiate between 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	They are 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

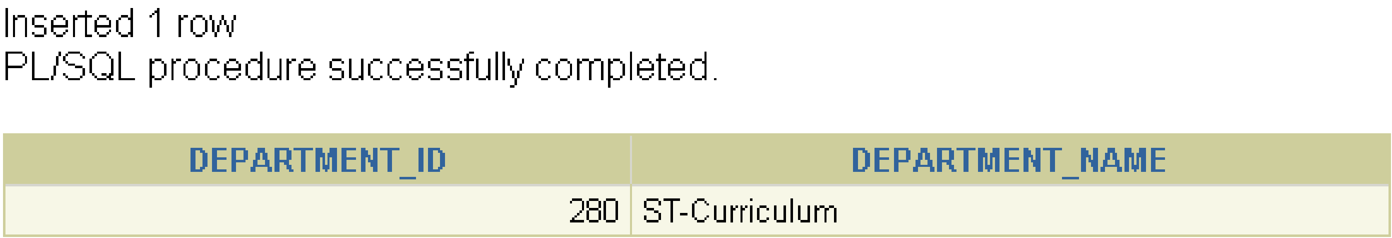
```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
IS|AS
procedure_body;
```

Procedure: Example

```
...
CREATE TABLE dept AS SELECT * FROM departments;
CREATE PROCEDURE add_dept IS
 dept_id dept.department_id%TYPE;
 dept_name dept.department_name%TYPE;
BEGIN
 dept_id:=280;
 dept_name:='ST-Curriculum';
 INSERT INTO dept(department_id,department_name)
 VALUES(dept_id,dept_name);
 DBMS_OUTPUT.PUT_LINE(' Inserted '||
  SQL%ROWCOUNT ||' row ');
END;
/
```

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

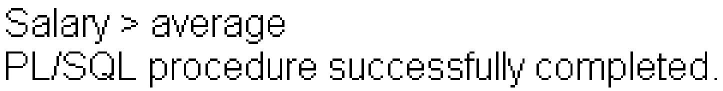
```
CREATE [OR REPLACE] FUNCTION function_name
 [(argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
function_body;
```

Function: Example

```
CREATE FUNCTION check_sal RETURN Boolean IS
 dept_id employees.department_id%TYPE;
 empno   employees.employee_id%TYPE;
 sal     employees.salary%TYPE;
 avg_sal employees.salary%TYPE;
BEGIN
 empno:=205;
 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
  WHEN NO_DATA_FOUND THEN
   RETURN NULL;
END;
/
```

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 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;
/
```

Summary

In this lesson, you should have learned how to:

- Create a simple procedure
- Invoke the procedure from an anonymous block
- Create a simple function
- Create a simple function that accepts parameters
- Invoke the function from an anonymous block