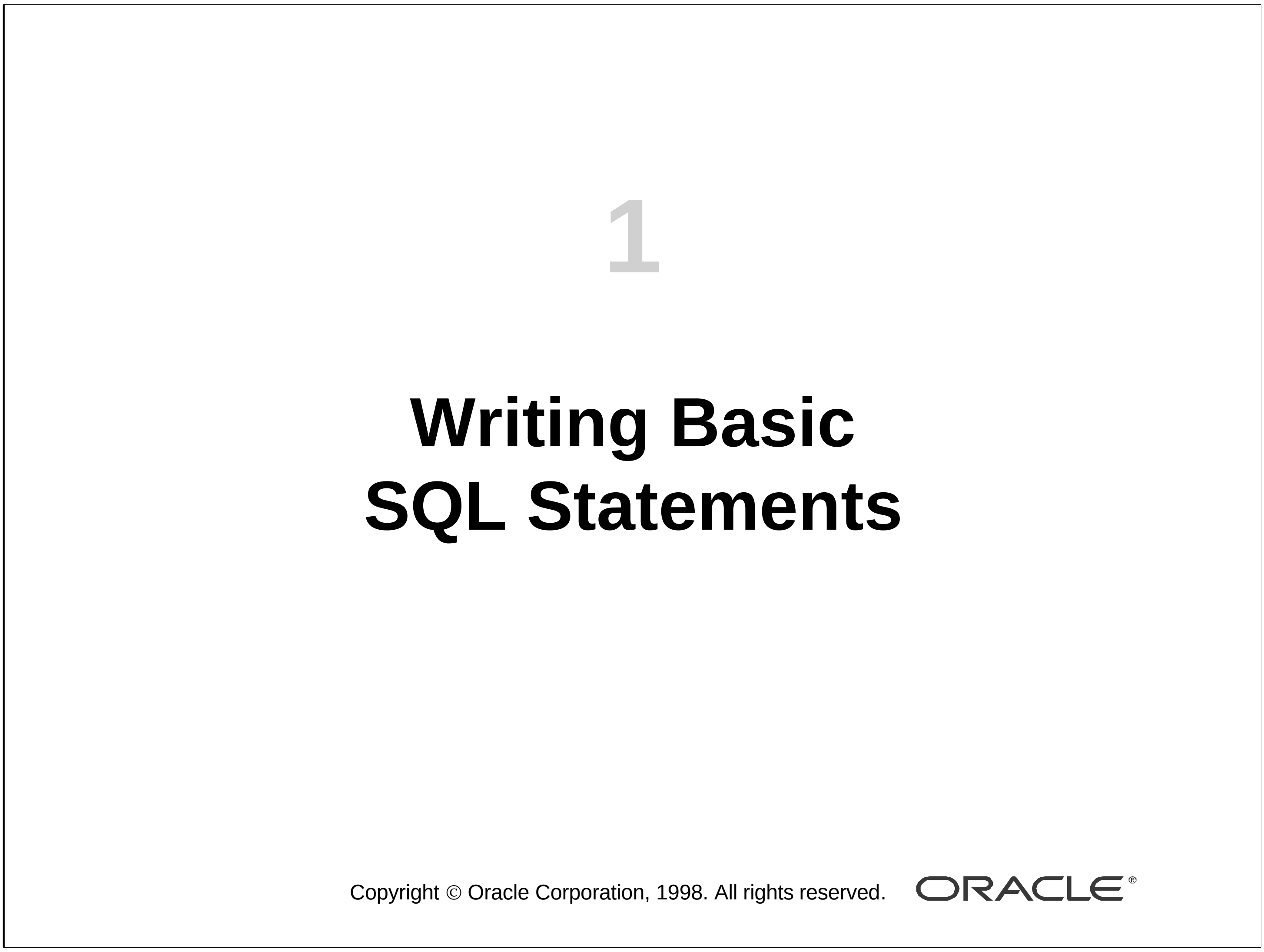


1

# Writing Basic SQL Statements

| Schedule: | Timing | Topic |
|---|---|---|
| | 40 minutes | Lecture |
| | 25 minutes | Practice |
| | 65 minutes | Total |

# Objectives

At the end of this lesson, you should be able to:

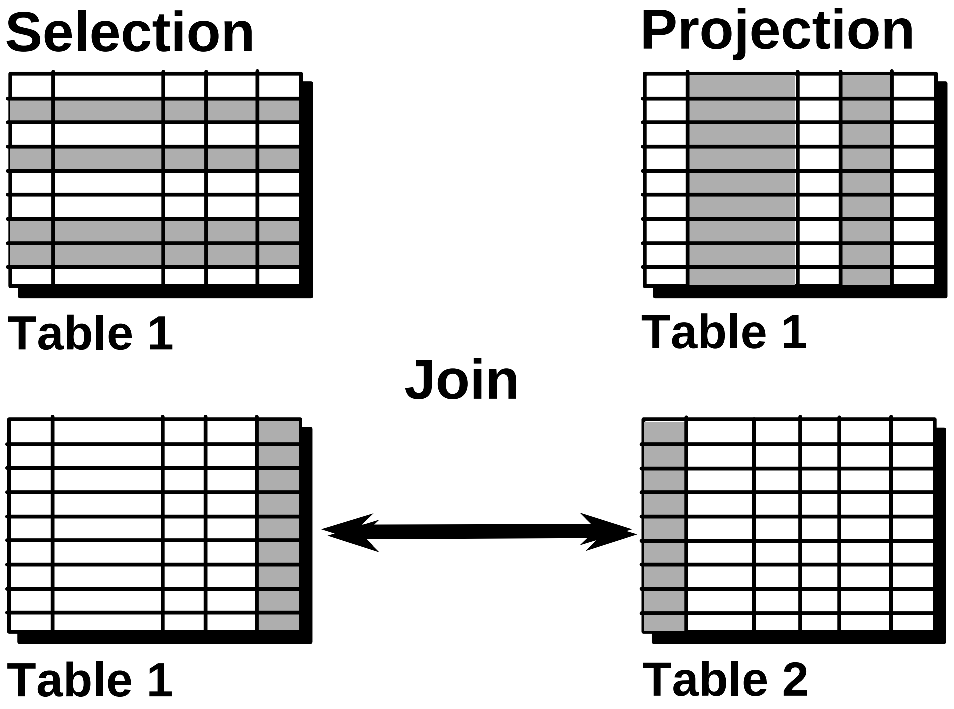
- List the capabilities of SQL SELECT statements
- Execute a basic SELECT statement
- Differentiate between SQL statements and SQL*Plus commands

**Lesson Aim**

To extract data from the database you need to use the structured query language (SQL) SELECT statement. You may need to restrict the columns that are displayed. This lesson describes all the SQL statements that you need to perform these actions.

You may want to create SELECT statements that can be used time and time again. This lesson also covers the use of SQL*Plus commands to execute SQL statements.

# Capabilities of SQL SELECT Statements



Selection

Projection

Table 1

Table 1

Join

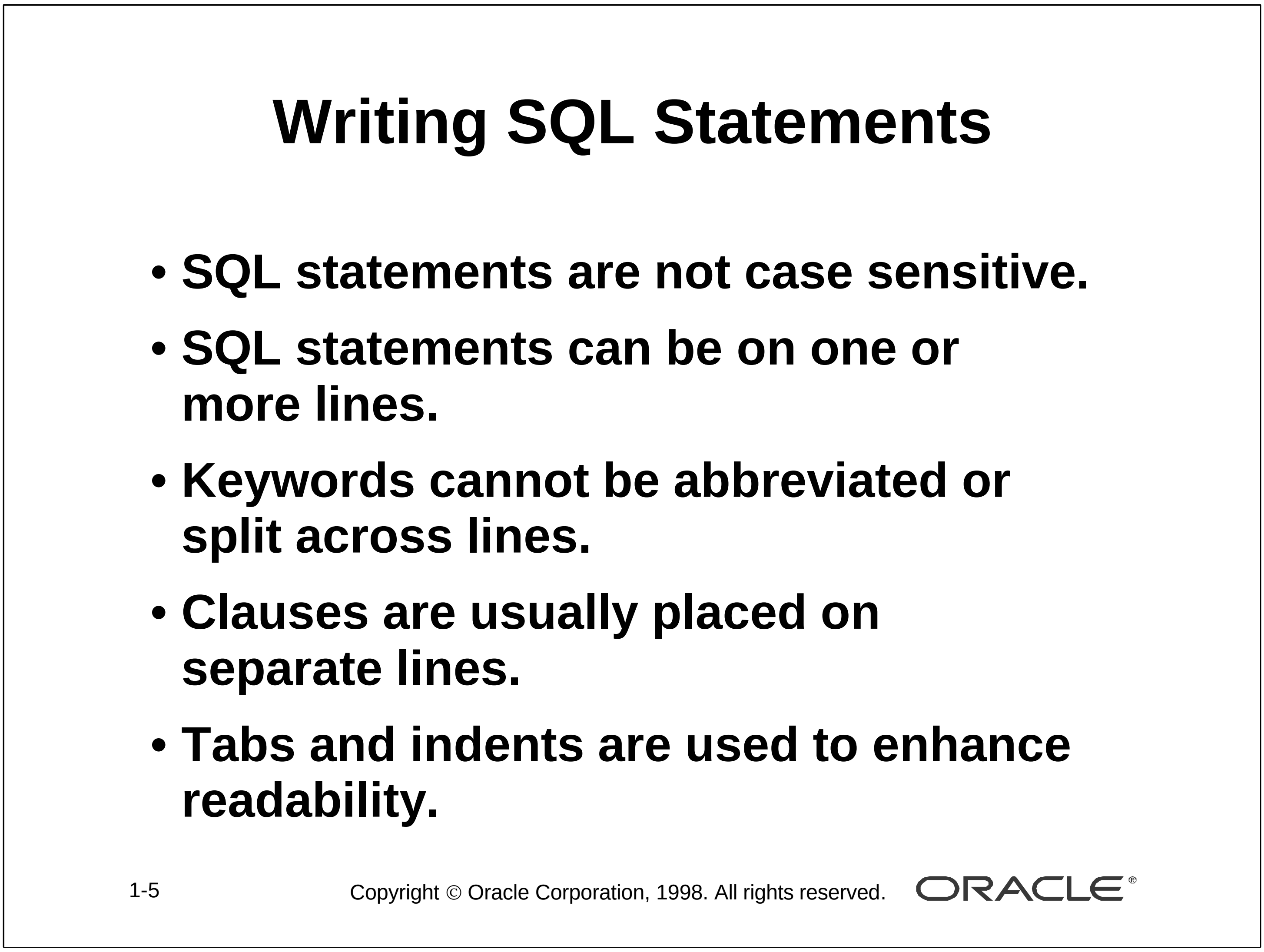
Table 1

Table 2

# Basic SELECT Statement

```
SELECT   [DISTINCT] {*, column [alias],...}
FROM     table;
```

- **SELECT identifies *what* columns**
- **FROM identifies *which* table**



# Writing SQL Statements

- SQL statements are not case sensitive.
- SQL statements can be on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Tabs and indents are used to enhance readability.

1-5

## Writing SQL Statements

By following simple rules and guidelines given below, you can construct valid statements that are both easy to read and easy to edit:

- SQL statements are not case sensitive, unless indicated.
- SQL statements can be entered on one or many lines.
- Keywords cannot be split across lines or abbreviated.
- Clauses are usually placed on separate lines for readability and ease of editing.
- Tabs and indents can be used to make code more readable.
- Keywords typically are entered in uppercase; all other words, such as table names and columns, are entered in lowercase.
- Within SQL*Plus, a SQL statement is entered at the SQL prompt, and the subsequent lines are numbered. This is called the *SQL buffer*. Only one statement can be current at any time within the buffer.

## Executing SQL Statements

- Place a semicolon (;) at the end of last clause.
- Place a slash on the last line in the buffer.
- Place a slash at the SQL prompt.
- Issue a SQL*Plus RUN command at the SQL prompt.

# Selecting All Columns

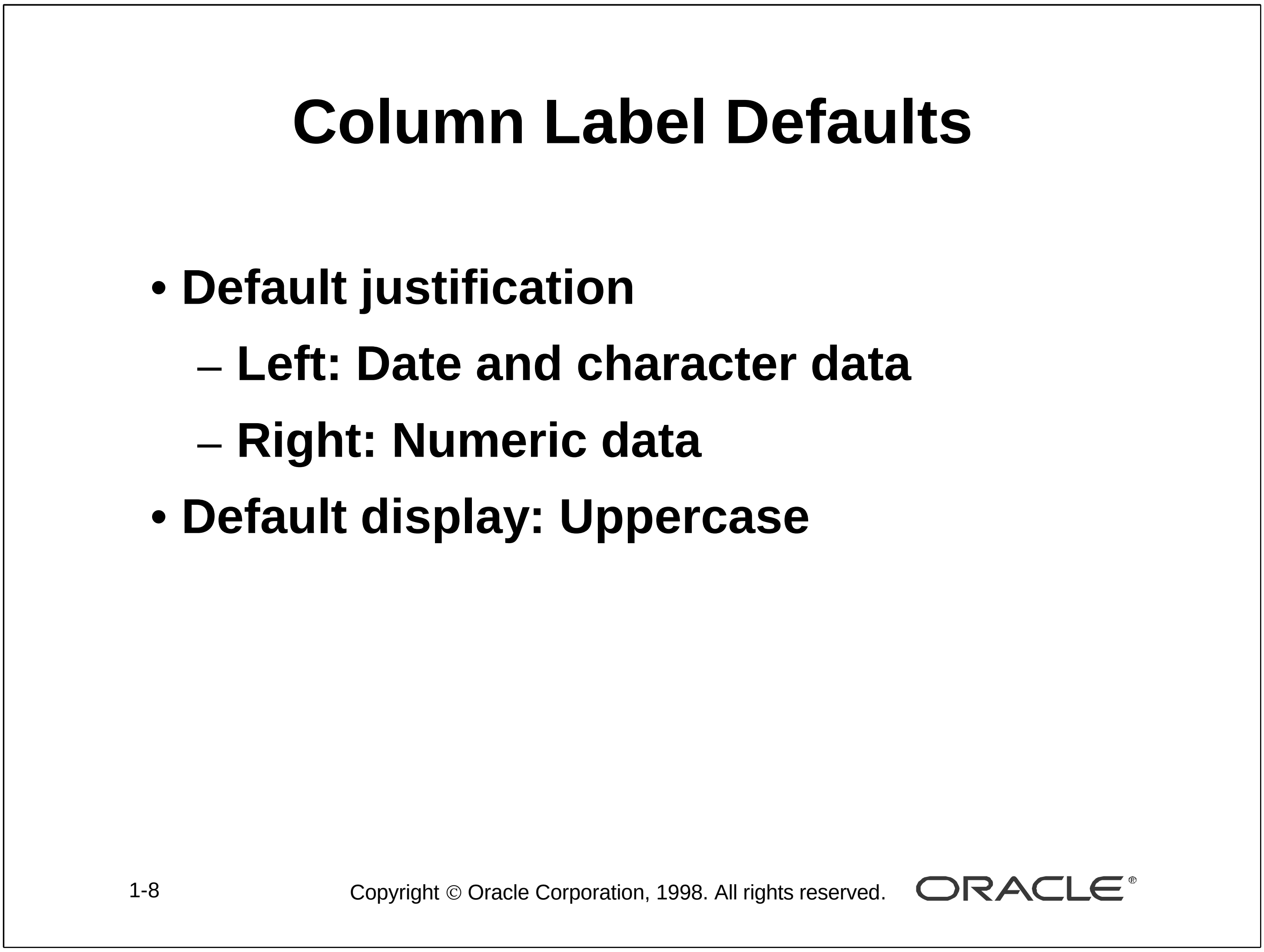
```
SQL> SELECT *
  2  FROM   dept;
```

```
   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
```

# Selecting Specific Columns

```
SQL> SELECT deptno, loc
  2  FROM   dept;
```

```
   DEPTNO LOC
--------- -------------
       10 NEW YORK
       20 DALLAS
       30 CHICAGO
       40 BOSTON
```



## Column Label Defaults

- Default justification
  - Left: Date and character data
  - Right: Numeric data
- Default display: Uppercase

1-8

ORACLE®

### Column Heading Defaults

Character column heading and data as well as date column heading and data are left-justified within a column width. Number headings and data are right-justified.

```
SQL> SELECT ename, hiredate, sal
  2  FROM   emp;
```

```
ENAME      HIREDATE        SAL
---------- --------- ---------
KING       17-NOV-81      5000
BLAKE      01-MAY-81      2850
CLARK      09-JUN-81      2450
JONES      02-APR-81      2975
MARTIN     28-SEP-81      1250
ALLEN      20-FEB-81      1600
...
14 rows selected.
```

Character and date column headings can be truncated, but number headings cannot be truncated. The column labels appear in uppercase by default. You can override the column label display with an alias. Column aliases are covered later in this lesson.

# Arithmetic Expressions

**Create expressions on NUMBER and DATE data types by using arithmetic operators.**

| Operator | Description |
|---|---|
| + | Add |
| - | Subtract |
| * | Multiply |
| / | Divide |

# Using Arithmetic Operators

```
SQL> SELECT ename, sal, sal+300
  2  FROM   emp;
```

```
ENAME            SAL   SAL+300
---------- --------- ---------
KING            5000      5300
BLAKE           2850      3150
CLARK           2450      2750
JONES           2975      3275
MARTIN          1250      1550
ALLEN           1600      1900
...
14 rows selected.
```

# Operator Precedence

*  /  +  –

- Multiplication and division take priority over addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to force prioritized evaluation and to clarify statements.

# Operator Precedence

```
SQL> SELECT ename, sal, 12*sal+100
  2  FROM   emp;
```

```
ENAME            SAL 12*SAL+100
---------- --------- ----------
KING            5000      60100
BLAKE           2850      34300
CLARK           2450      29500
JONES           2975      35800
MARTIN          1250      15100
ALLEN           1600      19300
...
14 rows selected.
```

# Using Parentheses

```
SQL> SELECT ename, sal, 12*(sal+100)
  2  FROM   emp;
```

```
ENAME            SAL 12*(SAL+100)
---------- --------- -----------
KING            5000       61200
BLAKE           2850       35400
CLARK           2450       30600
JONES           2975       36900
MARTIN          1250       16200
...
14 rows selected.
```

# Defining a Null Value

- **A null is a value that is unavailable, unassigned, unknown, or inapplicable.**
- **A null is not the same as zero or a blank space.**

```
SQL> SELECT   ename, job, comm
  2  FROM     emp;
```

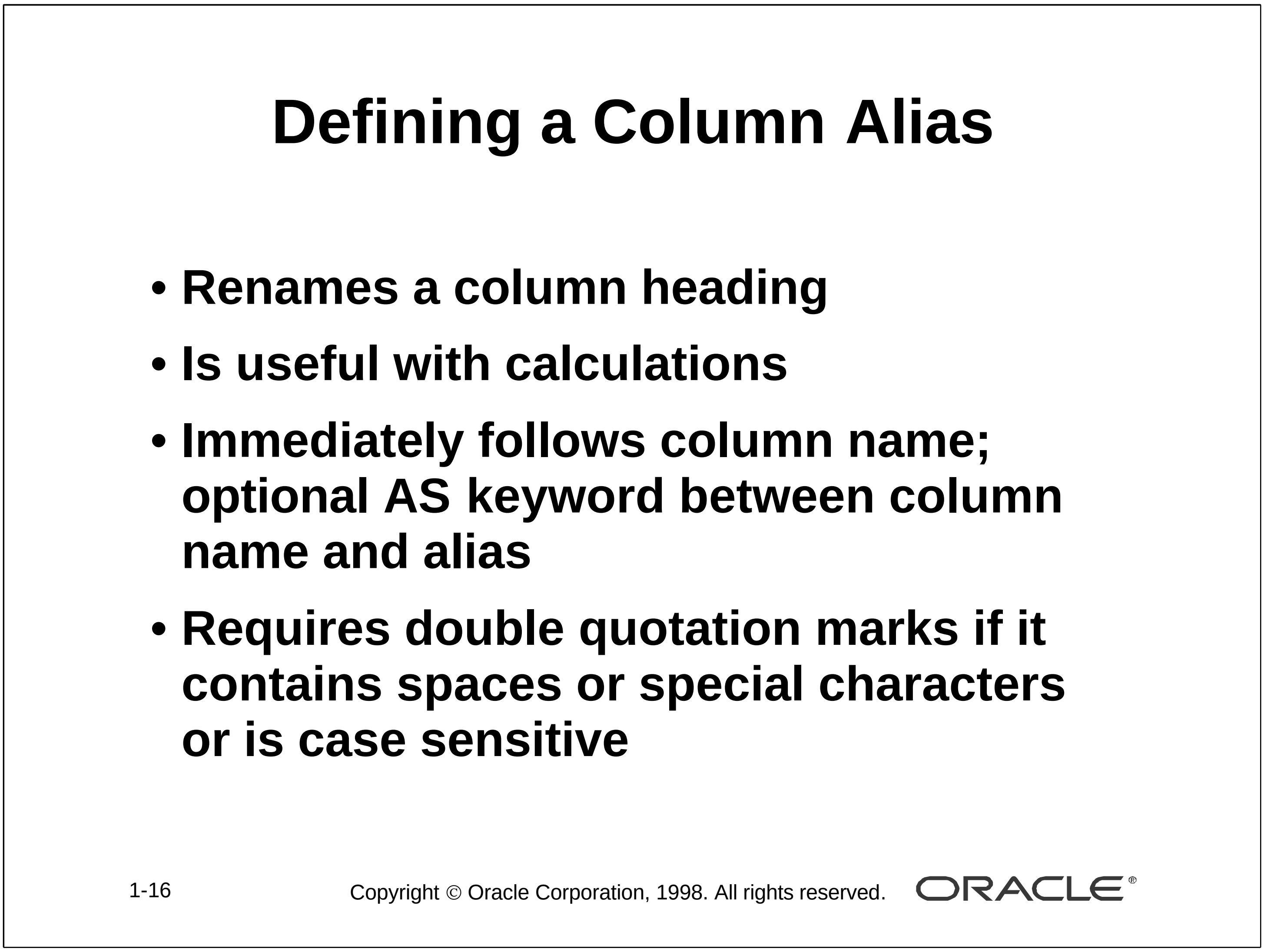
```
ENAME      JOB            COMM
---------- --------- ---------
KING       PRESIDENT
BLAKE      MANAGER
...
TURNER     SALESMAN          0
...
14 rows selected.
```

# Null Values in Arithmetic Expressions

**Arithmetic expressions containing a null value evaluate to null.**

```
SQL> select ename NAME, 12*sal+comm
  2  from   emp
  3  WHERE  ename='KING';
```

```
NAME       12*SAL+COMM
---------- -----------
KING
```



# Defining a Column Alias

- Renames a column heading
- Is useful with calculations
- Immediately follows column name; optional AS keyword between column name and alias
- Requires double quotation marks if it contains spaces or special characters or is case sensitive

1-16

### Column Aliases

When displaying the result of a query, SQL*Plus normally uses the name of the selected column as the column heading. In many cases, this heading may not be descriptive and hence is difficult to understand. You can change a column heading by using a column alias.

Specify the alias after the column in the SELECT list using a space as a separator. By default, alias headings appear in uppercase. If the alias contains spaces, special characters (such as # or $), or is case sensitive, enclose the alias in double quotation marks (“ ”).

### Class Management Note

Within a SQL statement, a column alias can be used in both the SELECT clause and the ORDER BY clause. You cannot use column aliases in the WHERE clause. Both alias features comply with the ANSI SQL 92 standard.

Demo: *l1alias.sql*

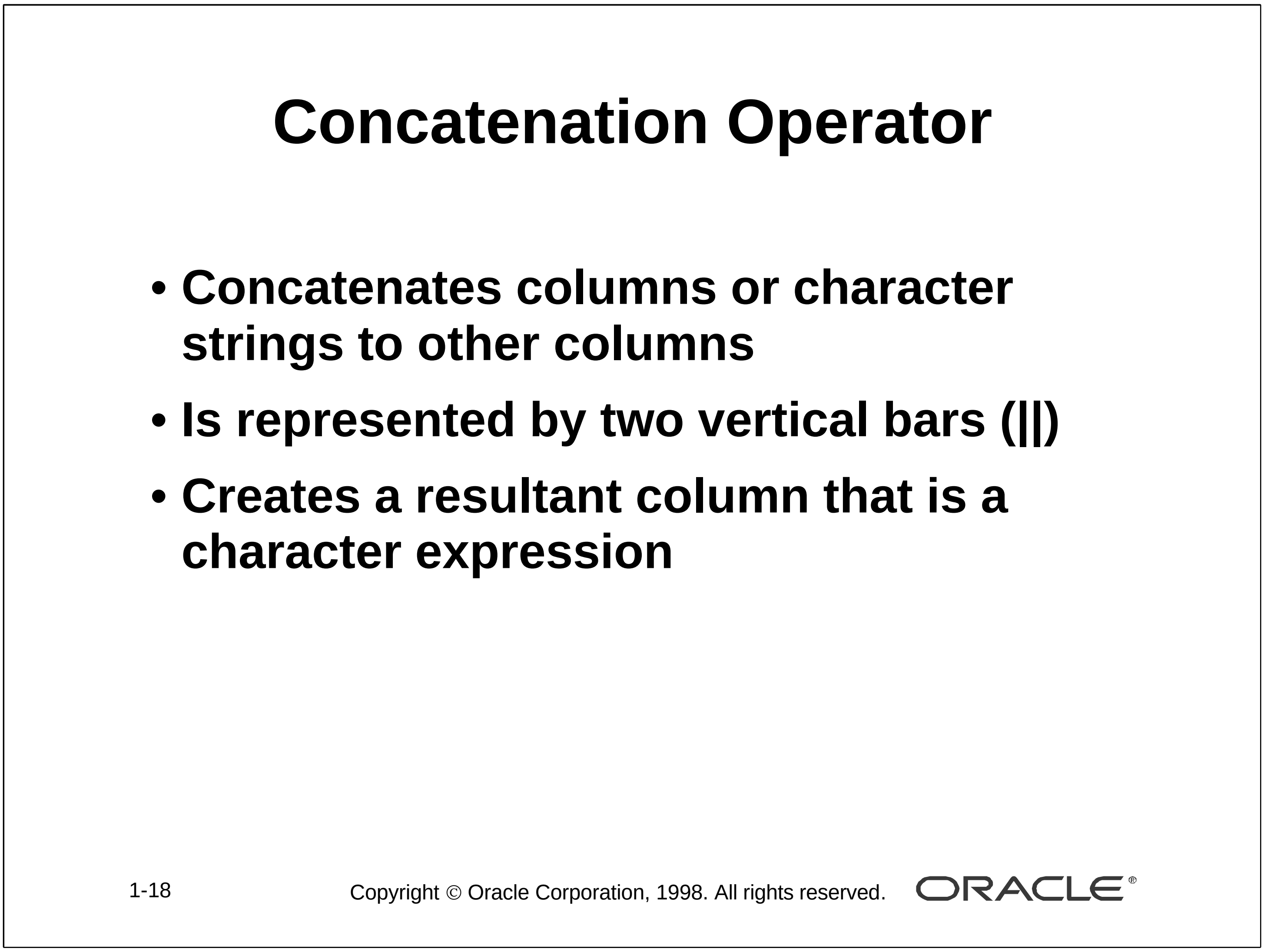
# Using Column Aliases

```
SQL> SELECT ename AS name, sal salary
  2  FROM   emp;
```

```
NAME             SALARY
------------- ---------
...
```

```
SQL> SELECT ename "Name",
  2         sal*12 "Annual Salary"
  3  FROM   emp;
```

```
Name          Annual Salary
------------- -------------
...
```



# Concatenation Operator

- Concatenates columns or character strings to other columns
- Is represented by two vertical bars (||)
- Creates a resultant column that is a character expression

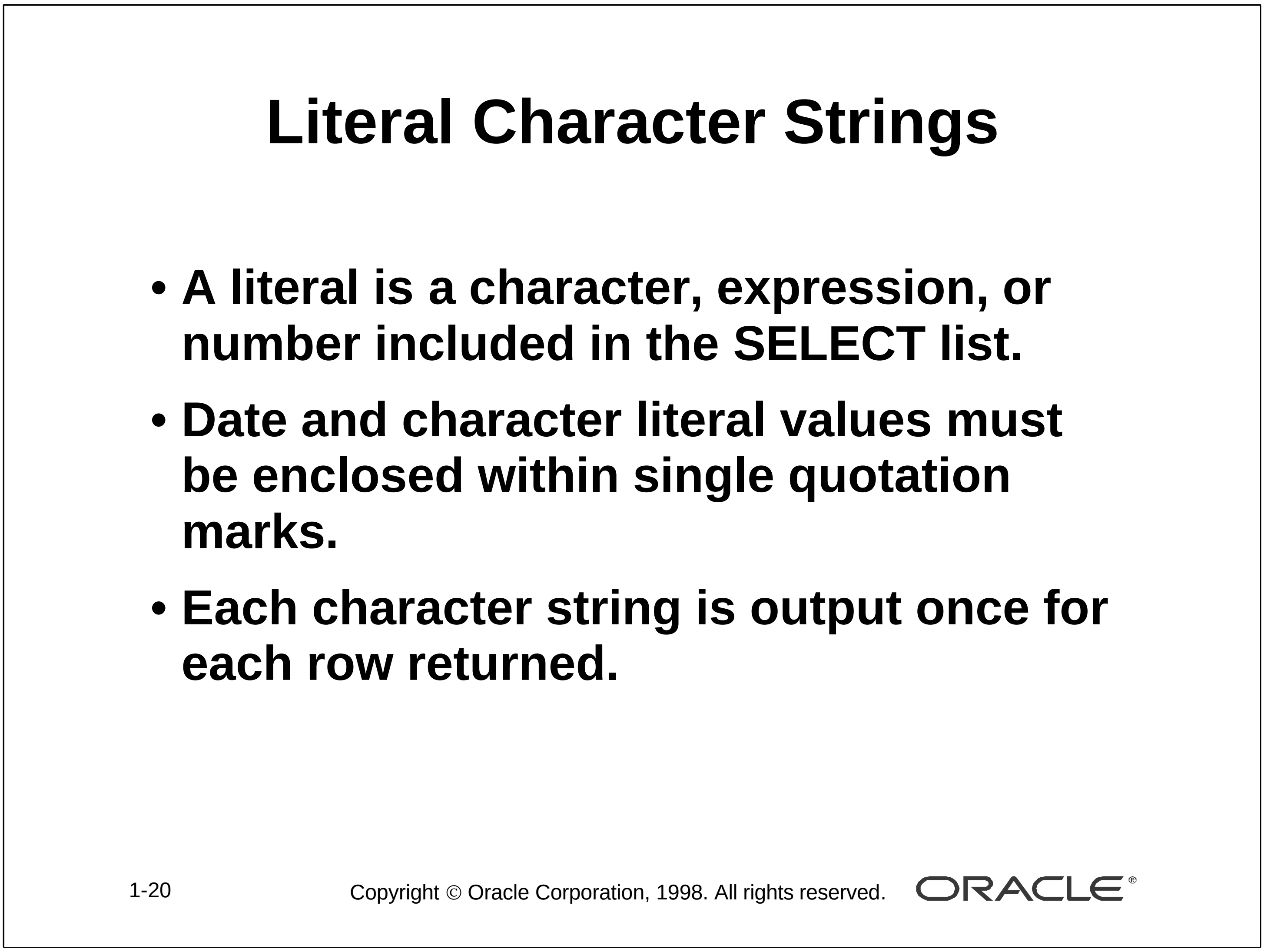
### Concatenation Operator

You can link columns to other columns, arithmetic expressions, or constant values to create a character expression by using the concatenation operator (||). Columns on either side of the operator are combined to make a single output column.

# Using the Concatenation Operator

```
SQL> SELECT   ename||job AS "Employees"
  2  FROM     emp;
```

```
Employees
-------------------
KINGPRESIDENT
BLAKEMANAGER
CLARKMANAGER
JONESMANAGER
MARTINSALESMAN
ALLENSALESMAN
...
14 rows selected.
```



# Literal Character Strings

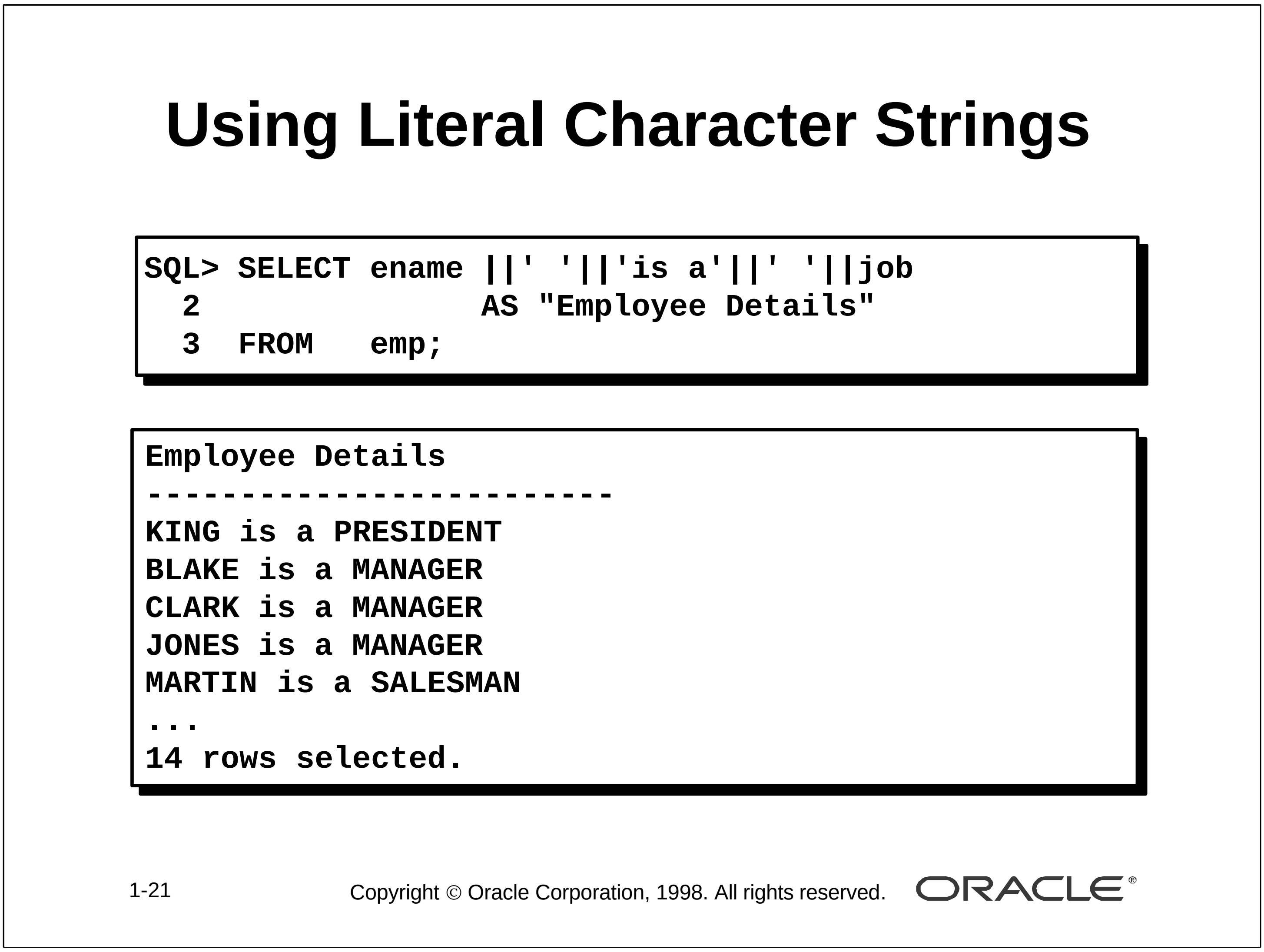
- A literal is a character, expression, or number included in the SELECT list.
- Date and character literal values must be enclosed within single quotation marks.
- Each character string is output once for each row returned.

1-20

## Literal Character Strings

A literal is any character, expression, or number included in the SELECT list that is not a column name or a column alias. It is printed for each row returned. Literal strings of free-format text can be included in the query result and are treated the same as a column in the SELECT list.

Date and character literals *must* be enclosed within single quotation marks (‘ ’); number literals must not.



# Using Literal Character Strings

```
SQL> SELECT ename ||' '||'is a'||' '||job
  2              AS "Employee Details"
  3  FROM   emp;
```

```
Employee Details
-------------------------
KING is a PRESIDENT
BLAKE is a MANAGER
CLARK is a MANAGER
JONES is a MANAGER
MARTIN is a SALESMAN
...
14 rows selected.
```

1-21

# Duplicate Rows

## The default display of queries is all rows, including duplicate rows.

```
SQL> SELECT deptno
  2  FROM   emp;
```

```
   DEPTNO
---------
       10
       30
       10
       20
...
14 rows selected.
```

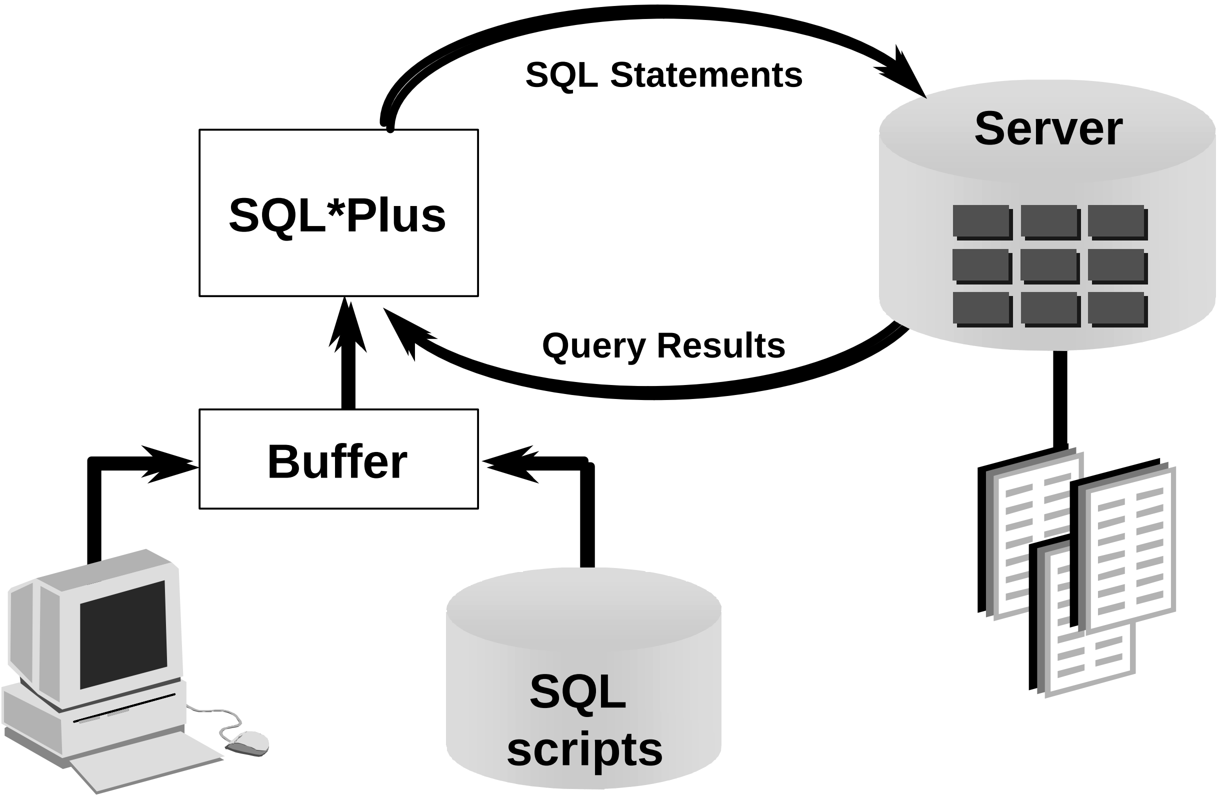
# Eliminating Duplicate Rows

**Eliminate duplicate rows by using the DISTINCT keyword in the SELECT clause.**

```
SQL> SELECT DISTINCT deptno
  2  FROM   emp;
```

```
   DEPTNO
---------
       10
       20
       30
```

# SQL and SQL*Plus Interaction



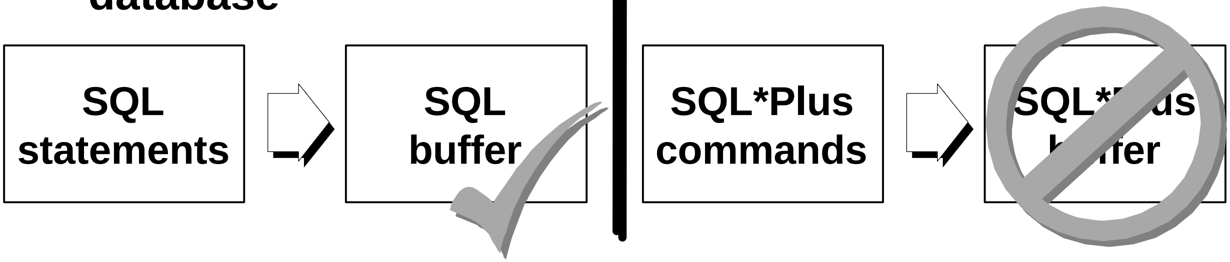
# SQL Statements Versus SQL*Plus Commands

**SQL**

- A language
- ANSI standard
- Keyword cannot be abbreviated
- Statements manipulate data and table definitions in the database

**SQL*Plus**

- An environment
- Oracle proprietary
- Keywords can be abbreviated
- Commands do not allow manipulation of values in the database



# Overview of SQL*Plus

- Log in to SQL*Plus.
- Describe the table structure.
- Edit your SQL statement.
- Execute SQL from SQL*Plus.
- Save SQL statements to files and append SQL statements to files.
- Execute saved files.
- Load commands from file to buffer to edit.

1-26

### SQL*Plus

SQL*Plus is an environment in which you can do the following:

- Execute SQL statements to retrieve, modify, add, and remove data from the database
- Format, perform calculations on, store, and print query results in the form of reports
- Create script files to store SQL statements for repetitive use in the future

SQL*Plus commands can be divided into the following main categories:

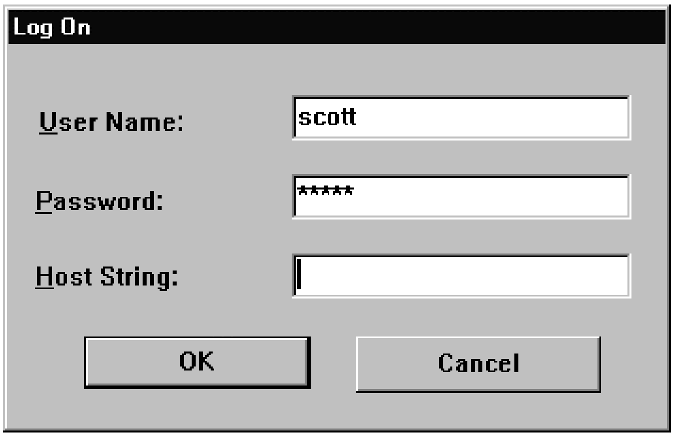
| Category | Purpose |
|---|---|
| Environment | Affects the general behavior of SQL statements for the session |
| Format | Formats query results |
| File manipulation | Saves, loads, and runs script files |
| Execution | Sends SQL statements from SQL buffer to Oracle8 Server |
| Edit | Modifies SQL statements in the buffer |
| Interaction | Allows you to create and pass variables to SQL statements, print variable values, and print messages to the screen |
| Miscellaneous | Has various commands to connect to the database, manipulate the SQL*Plus environment, and display column definitions |

**Class Management Note (For page 1-27)**

Snippet: Establishing a Database Session

# Logging In to SQL*Plus

- From Windows environment:



Log On

User Name: scott

Password: *****

Host String:

- From command line:

```
sqlplus [username[/password
          [@database]]]
```

# Displaying Table Structure

**Use the SQL*Plus DESCRIBE command to display the structure of a table.**

```
DESC[RIBE] tablename
```

# Displaying Table Structure

```
SQL> DESCRIBE dept
```

```
Name              Null?    Type
----------------- -------- ----
DEPTNO            NOT NULL NUMBER(2)
DNAME                      VARCHAR2(14)
LOC                        VARCHAR2(13)
```

# SQL*Plus Editing Commands

- **A[PPEND] *text***
- **C[HANGE] / *old* / *new***
- **C[HANGE] / *text* /**
- **CL[EAR] BUFF[ER]**
- **DEL**
- **DEL *n***
- **DEL *m n***

1-30

### SQL*Plus Editing Commands

SQL*Plus commands are entered one line at a time and are not stored in the SQL buffer.

| **Command** | **Description** |
|---|---|
| A[PPEND] *text* | Adds text to the end of the current line |
| C[HANGE] / *old* / *new* | Changes *old* text to *new* in the current line |
| C[HANGE] / *text* / | Deletes *text* from the current line |
| CL[EAR] BUFF[ER] | Deletes all lines from the SQL buffer |
| DEL | Deletes current line |

### Guidelines

- If you press [Return] before completing a command, SQL*Plus prompts you with a line number.
- You terminate the SQL buffer by either entering one of the terminator characters (semicolon or slash) or by pressing [Return] twice. You then see the SQL prompt.

# SQL*Plus Editing Commands

- **I[NPUT]**
- **I[NPUT] *text***
- **L[IST]**
- **L[IST] *n***
- **L[IST] *m n***
- **R[UN]**
- ***n***
- ***n text***
- **0 *text***

**SQL*Plus Editing Commands (continued)**

| Command | Description |
|---|---|
| I[NPU T] | Inserts an indefinite number of lines |
| I[NPUT] *text* | Inserts a line consisting of *text* |
| L[IST] | Lists all lines in the SQL buffer |
| L[IST] *n* | Lists one line (specified by *n*) |
| L[IST] *m n* | Lists a range of lines (*m* to *n)* |
| R[UN] | Displays and runs the current SQL statement in the buffer |
| *n* | Specifies the line to make the current line |
| *n text* | Replaces line *n* with *text* |
| 0 *text* | Inserts a line before line 1 |

You can enter only one SQL*Plus command per SQL prompt. SQL*Plus commands are not stored in the buffer. To continue a SQL*Plus command on the next line, end the current line with a hyphen (-).

**Class Management Note**

Show students the use of the commonly used editing commands, like A[PPEND], C[HANGE], DEL, L[IST], and R[UN].

# SQL*Plus File Commands

- **SAVE *filename***
- **GET *filename***
- **START *filename***
- **@ *filename***
- **EDIT *filename***
- **SPOOL *filename***
- **EXIT**

### SQL*Plus File Commands

SQL statements communicate with the Oracle Server. SQL*Plus commands control the environment, format query results, and manage files. You can use the commands identified in the following table:

| **Command** | **Description** |
|---|---|
| SAV[E] *filename* [.ext] [REP[LACE]APP[END]] | Saves current contents of SQL buffer to a file. Use APPEND to add to an existing file; use REPLACE to overwrite an existing file. The default extension is .sql. |
| GET *filename* [.ext] | Writes the contents of a previously saved file to the SQL buffer. The default extension for the filename is .sql. |
| STA[RT] *filename* [.ext] | Runs a previously saved command file. |
| @ *filename* | Runs a previously saved command file (same as START). |
| ED[IT] | Invokes the editor and saves the buffer contents to a file named *afiedt.buf*. |
| ED[IT] [*filename*[.ext]] | Invokes editor to edit contents of a saved file. |
| SPO[OL] [*filename*[.ext]\| OFF\|OUT] | Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the system printer. |
| EXIT | Leaves SQL*Plus. |

# Summary

```
SELECT   [DISTINCT] {*,column[alias],...}
FROM     table;
```

Use SQL*Plus as an environment to:

- Execute SQL statements
- Edit SQL statements

# Practice Overview

- Selecting all data from different tables.
- Describing the structure of tables.
- Performing arithmetic calculations and specifying column names.
- Using SQL*Plus editor.

### Practice Overview

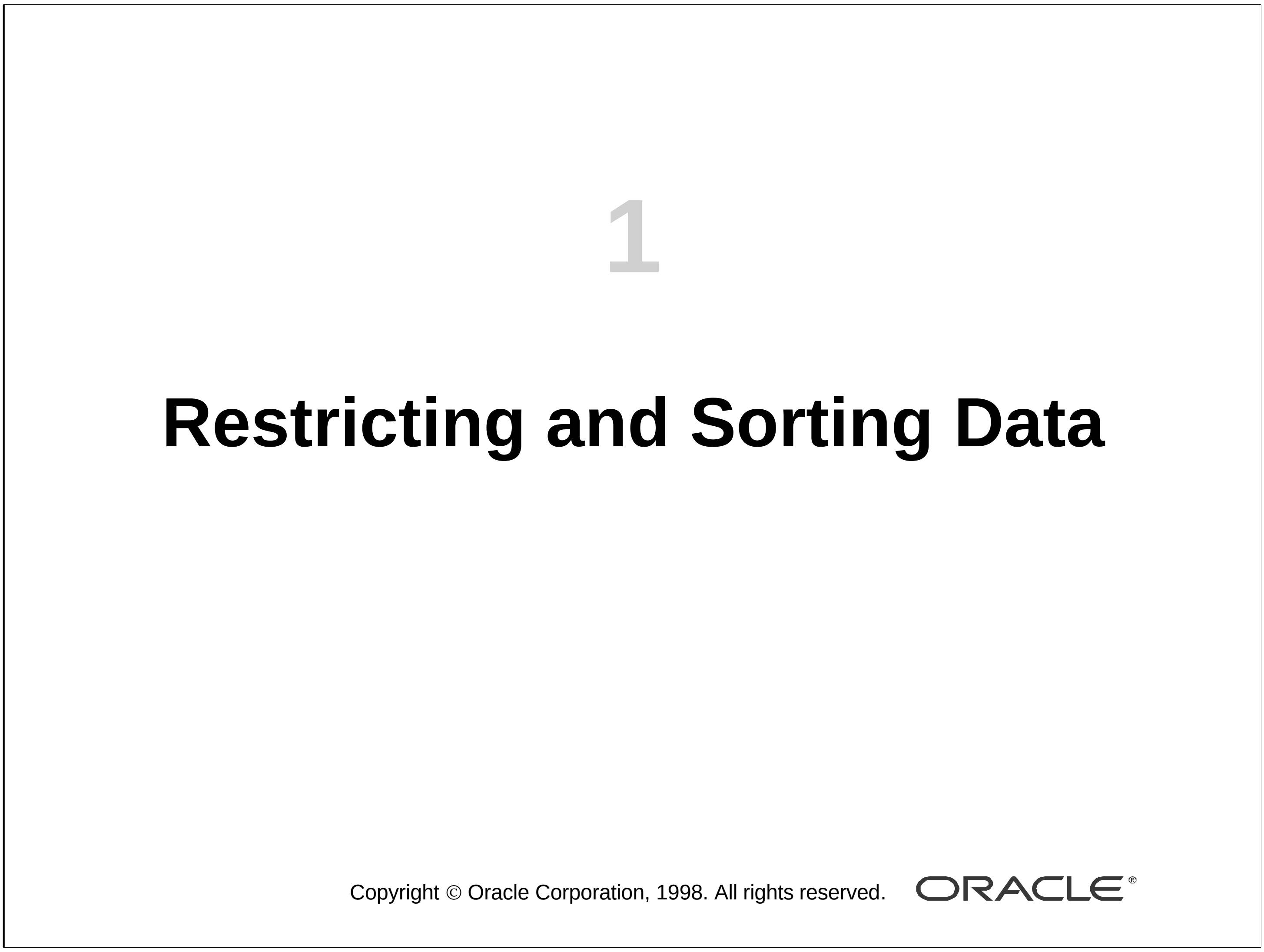
This is the first of many practices. The solutions (if you require them) can be found in Appendix A. Practices are intended to introduce all topics covered in the lesson. Questions 2-4 are paper-based.

In any practice, there may be "if you have time" or "if you want extra challenge" questions. Do these only if you have completed all other questions within the allocated time and would like a further challenge to your skills.

Take the practice slowly and precisely. You can experiment with saving and running command files. If you have any questions at any time, attract the instructor's attention.

### Paper-Based Questions

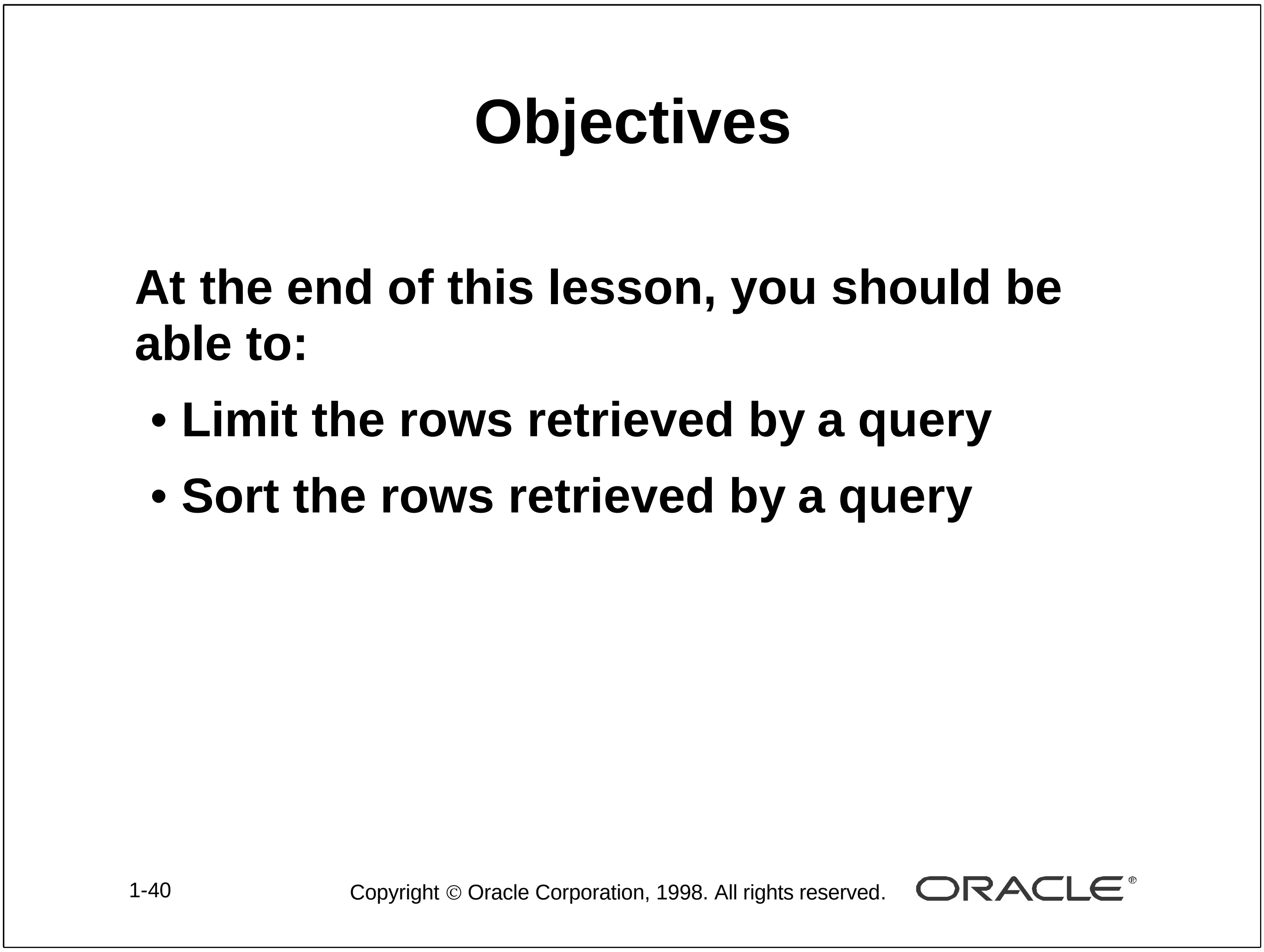
For questions 2-4 circle either True or False.



1

# Restricting and Sorting Data

| Schedule: | Timing | Topic |
|---|---|---|
| | 45 minutes | Lecture |
| | 30 minutes | Practice |
| | 75 minutes | Total |



# Objectives

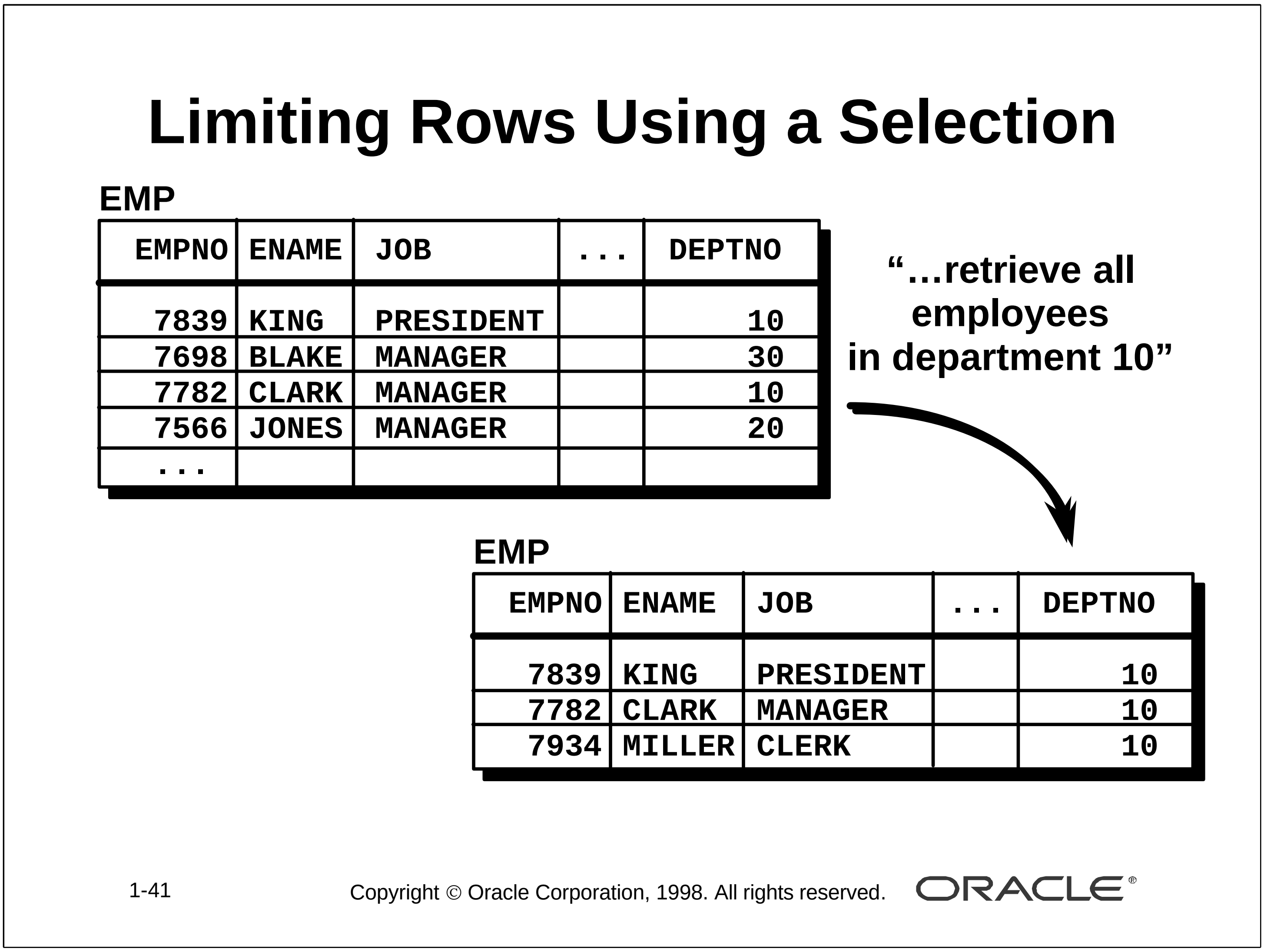
At the end of this lesson, you should be able to:

- Limit the rows retrieved by a query
- Sort the rows retrieved by a query

1-40

**Lesson Aim**

While retrieving data from the database, you may need to restrict the rows of data that are displayed or specify the order in which the rows are displayed. This lesson explains the SQL statements that you will use to perform these actions.



# Limiting Rows Using a Selection

EMP

| EMPNO | ENAME | JOB | ... | DEPTNO |
|---|---|---|---|---|
| 7839 | KING | PRESIDENT | | 10 |
| 7698 | BLAKE | MANAGER | | 30 |
| 7782 | CLARK | MANAGER | | 10 |
| 7566 | JONES | MANAGER | | 20 |
| ... | | | | |

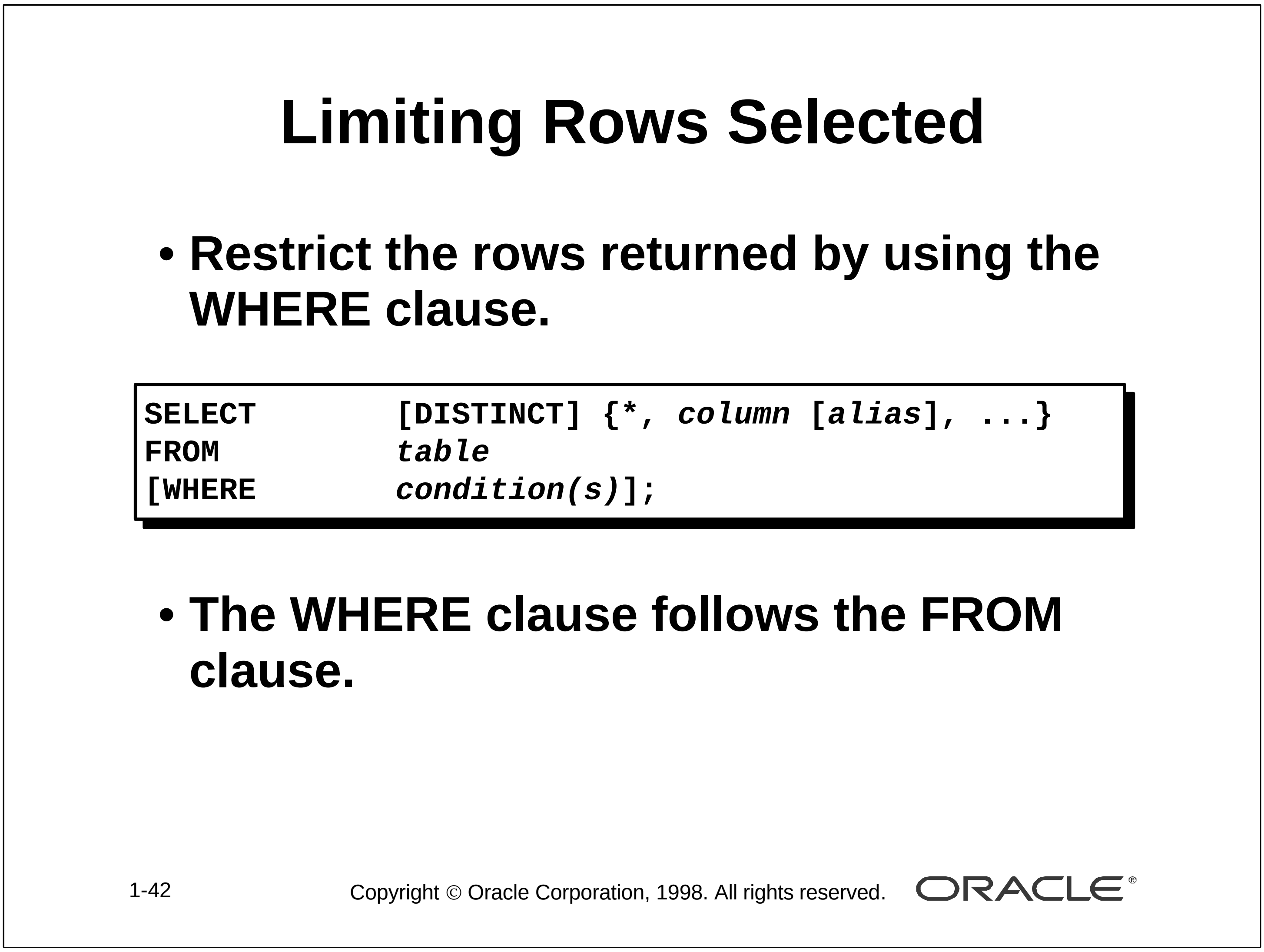
"…retrieve all employees in department 10"

EMP

| EMPNO | ENAME | JOB | ... | DEPTNO |
|---|---|---|---|---|
| 7839 | KING | PRESIDENT | | 10 |
| 7782 | CLARK | MANAGER | | 10 |
| 7934 | MILLER | CLERK | | 10 |

### Limiting Rows Using a Selection

In the above example, assume that you want to display all the employees in department 10. The highlighted set of rows with a value of 10 in DEPTNO column are the only ones returned. This method of restriction is the basis of the WHERE clause in SQL.



# Limiting Rows Selected

- **Restrict the rows returned by using the WHERE clause.**

```
SELECT       [DISTINCT] {*, column [alias], ...}
FROM         table
[WHERE       condition(s)];
```

- **The WHERE clause follows the FROM clause.**

1-42 ORACLE®

### Limiting Rows Selected

You can restrict the rows returned from the query by using the WHERE clause. A WHERE clause contains a condition that must be met, and it directly follows the FROM clause.

In the syntax:

WHERE restricts the query to rows that meet a condition

*condition* is composed of column names, expressions, constants, and comparison operator

The WHERE clause can compare values in columns, literal values, arithmetic expressions, or functions. The WHERE clause consists of three elements:

- Column name
- Comparison operator
- Column name, constant, or list of values

# Using the WHERE Clause

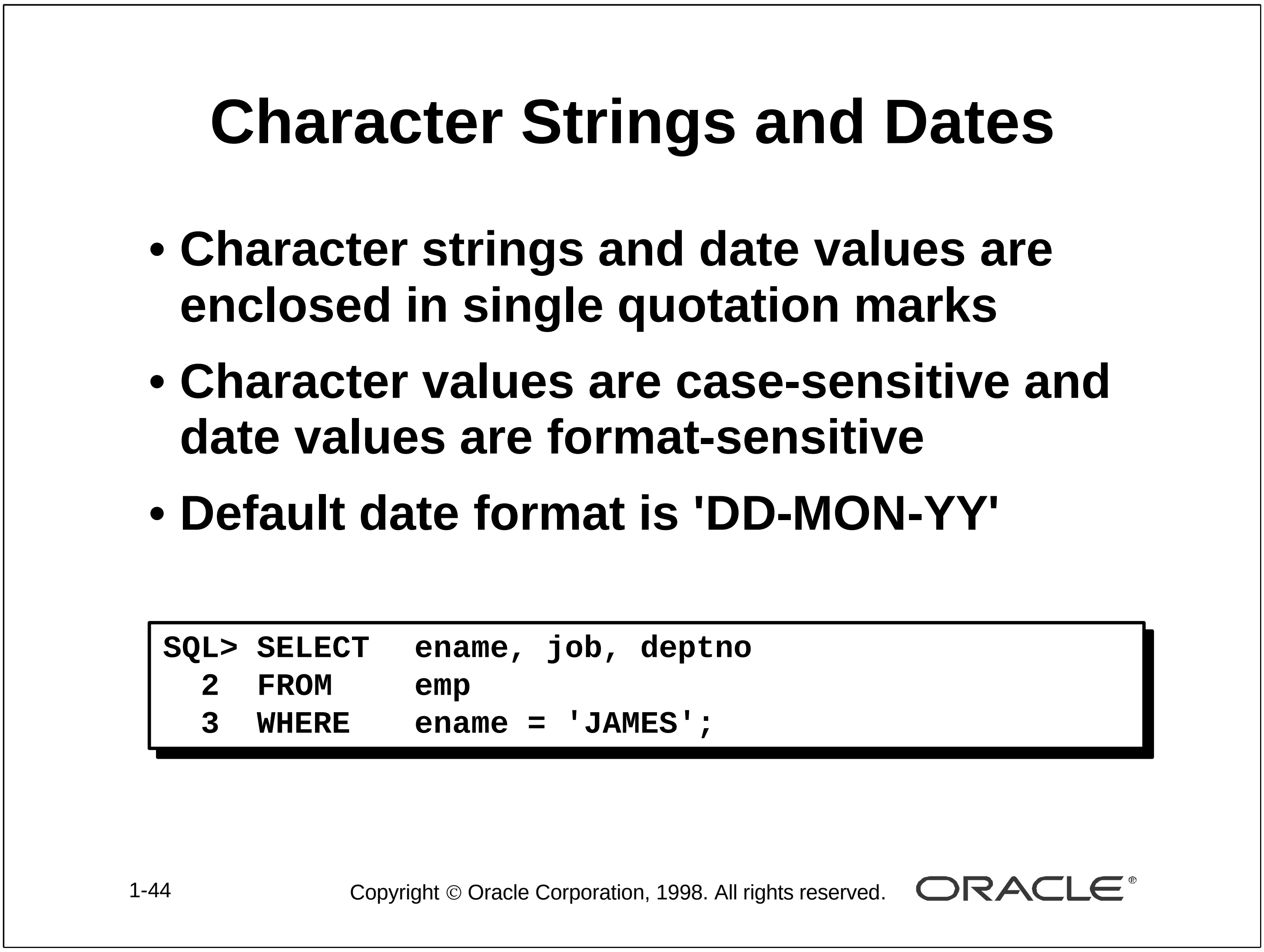
```
SQL> SELECT ename, job, deptno
  2  FROM   emp
  3  WHERE  job='CLERK';
```

```
ENAME      JOB          DEPTNO
---------- --------- ---------
JAMES      CLERK            30
SMITH      CLERK            20
ADAMS      CLERK            20
MILLER     CLERK            10
```

### Using the WHERE clause

In the example, the SELECT statement retrieves the name, job title, and the department number of all employees whose job title is CLERK.

Note that the job title CLERK has been specified in uppercase to ensure that the match is made with the job column in the EMP table. Character strings are case sensitive.



# Character Strings and Dates

- Character strings and date values are enclosed in single quotation marks
- Character values are case-sensitive and date values are format-sensitive
- Default date format is 'DD-MON-YY'

```
SQL> SELECT   ename, job, deptno
  2  FROM     emp
  3  WHERE    ename = 'JAMES';
```

### Character Strings and Dates

Character strings and dates in the WHERE clause must be enclosed in single quotation marks (’’). Number constants, however, must not.

All character searches are case sensitive. In the following example, no rows are returned because the EMP table stores all the data in uppercase.

```
SQL> SELECT ename, empno, job, deptno
  2  FROM   emp
  3  WHERE  job=’clerk’;
```

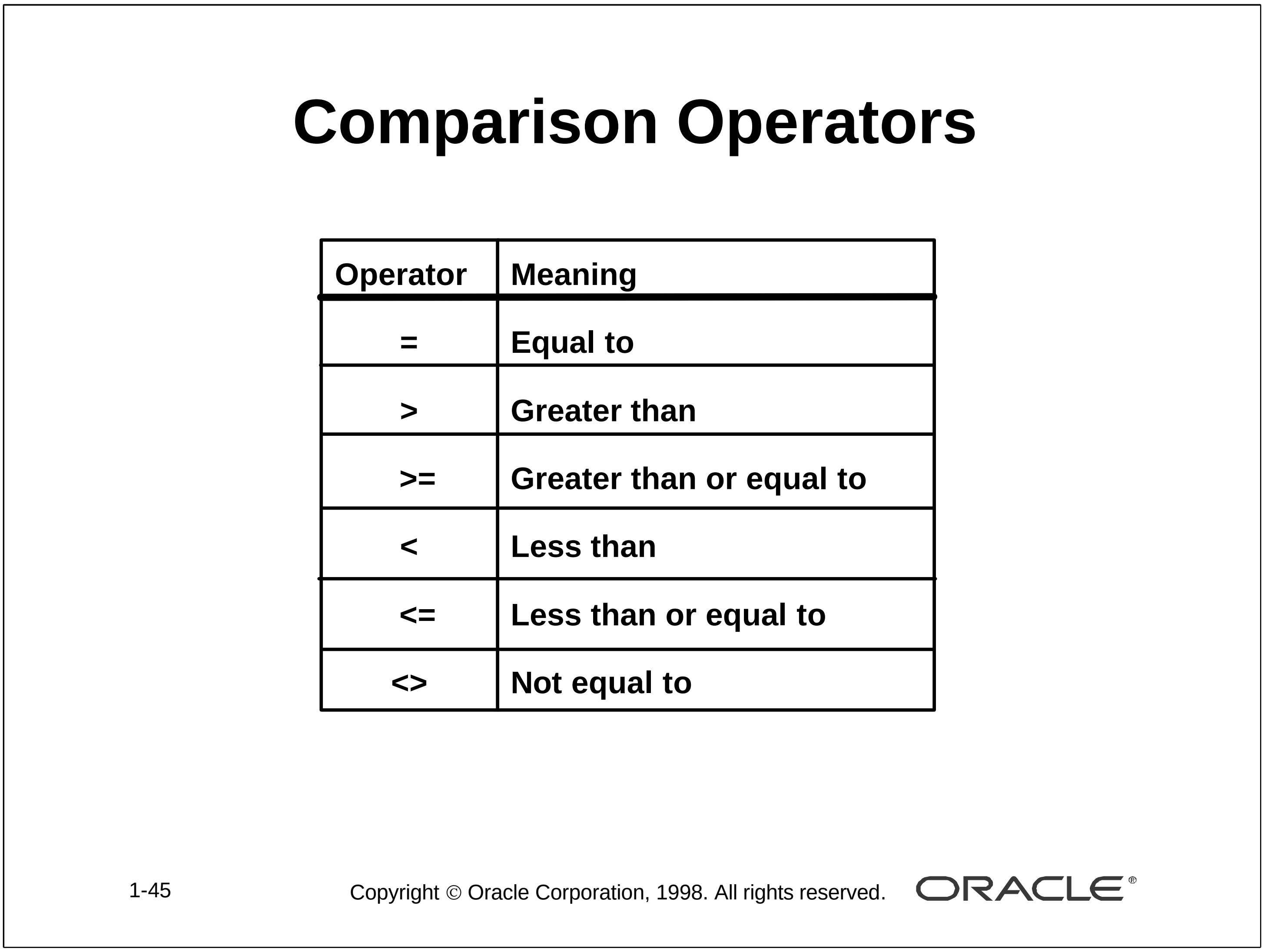
Oracle stores dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds. The default date display is DD-MON-YY.

**Note:** Changing default date format will be covered in lesson 3.

Number values are not enclosed within quotation marks.

### Class Management Note

Some students may ask how to override the case sensitivity. Later in the course, we will cover the use of single-row functions such as UPPER and LOWER to override the case sensitivity.



# Comparison Operators

| Operator | Meaning |
|---|---|
| = | Equal to |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |
| <> | Not equal to |

1-45

### Comparison Operators

Comparison operators are used in conditions that compare one expression to another. They are used in the WHERE clause in the following format.

### Syntax

```
… WHERE expr operator value
```

### Examples

```
… WHERE hiredate='01-JAN-95'
… WHERE sal>=1500
… WHERE ename='SMITH'
```

### Class Management Note

Remind students that the *expr* cannot be an alias.

# Using the Comparison Operators

```
SQL> SELECT ename, sal, comm
  2  FROM   emp
  3  WHERE  sal<=comm;
```

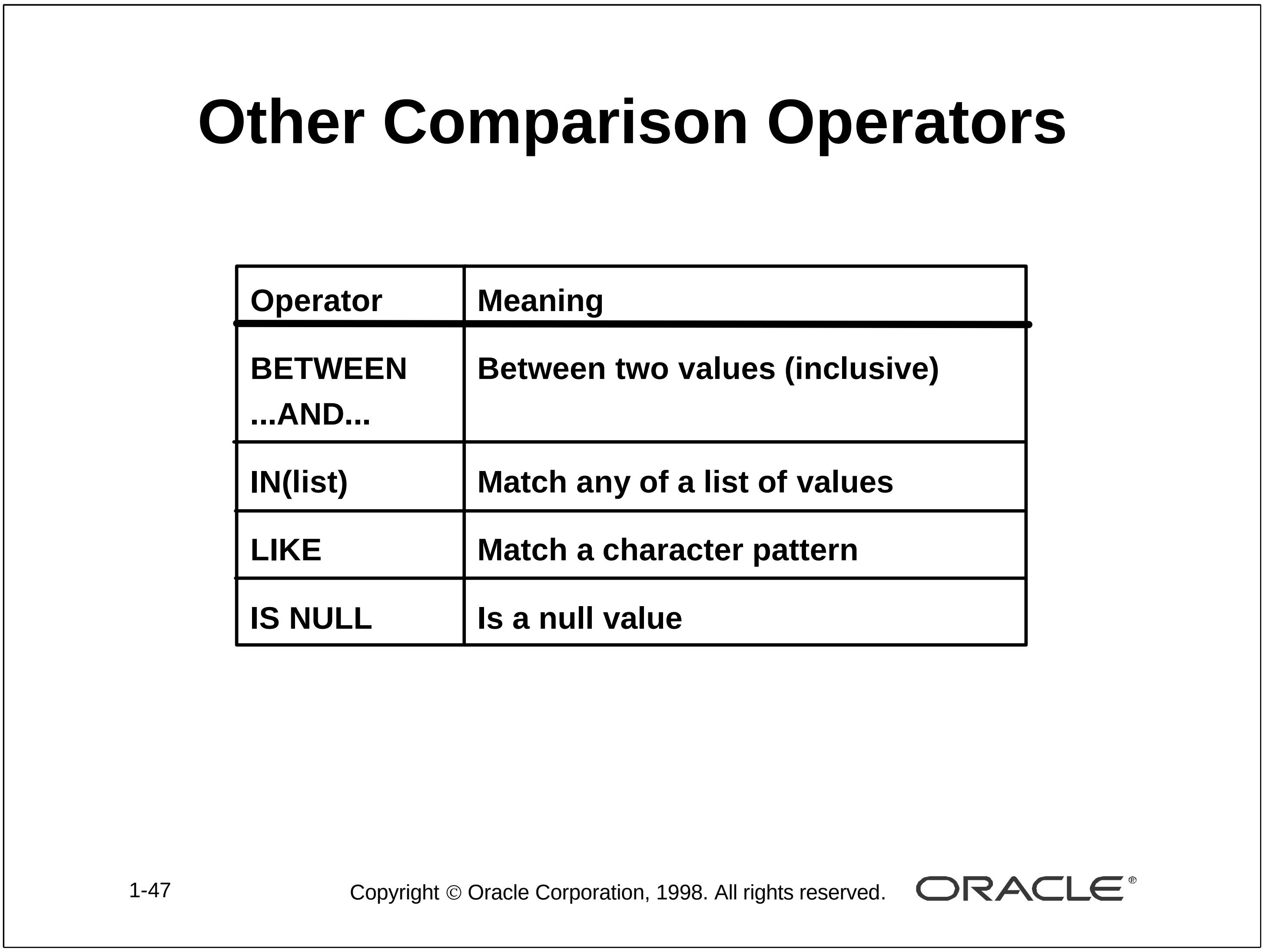
```
ENAME            SAL      COMM
---------- --------- ---------
MARTIN          1250  ↔   1400
```

### Using the Comparison Operators

In the example, the SELECT statement retrieves name, salary, and commission from the EMP table, where the employee salary is less than or equal to their commission amount. Note that there is no explicit value supplied to the WHERE clause. The two values being compared are taken from the SAL and COMM columns in the EMP table.

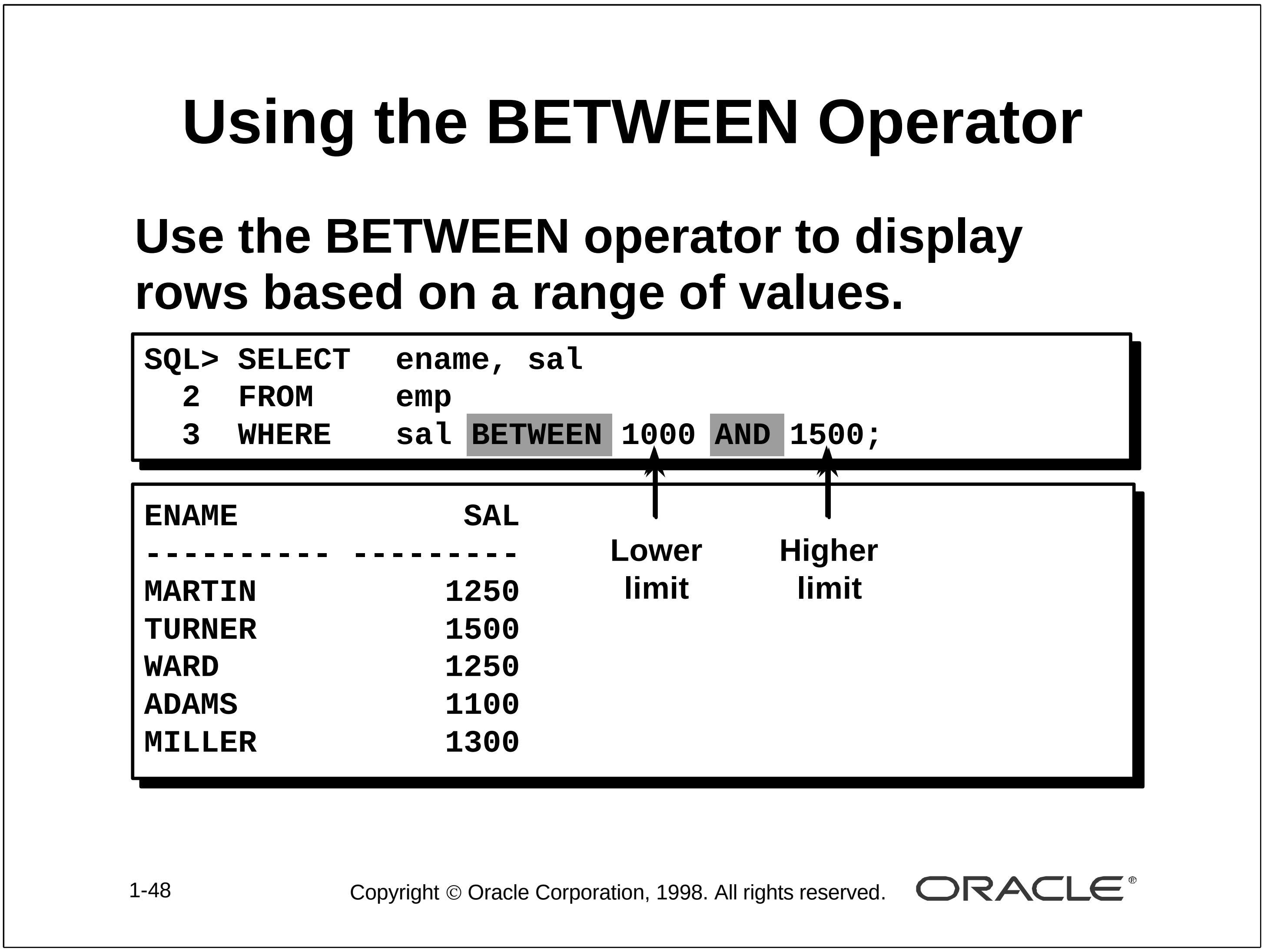
### Class Management Note

Rows that have a null value in the COMM column result in a null value for the comparison expression and are effectively not part of the result.



# Other Comparison Operators

| Operator | Meaning |
|---|---|
| BETWEEN<br>...AND... | Between two values (inclusive) |
| IN(list) | Match any of a list of values |
| LIKE | Match a character pattern |
| IS NULL | Is a null value |



# Using the BETWEEN Operator

## Use the BETWEEN operator to display rows based on a range of values.

```
SQL> SELECT   ename, sal
  2  FROM     emp
  3  WHERE    sal BETWEEN 1000 AND 1500;
```

```
ENAME            SAL
---------- ---------
MARTIN          1250
TURNER          1500
WARD            1250
ADAMS           1100
MILLER          1300
```

Lower limit

Higher limit

1-48 Copyright © Oracle Corporation, 1998. All rights reserved.

### The BETWEEN Operator

You can display rows based on a range of values using the BETWEEN operator. The range that you specify contains a lower range and an upper range.

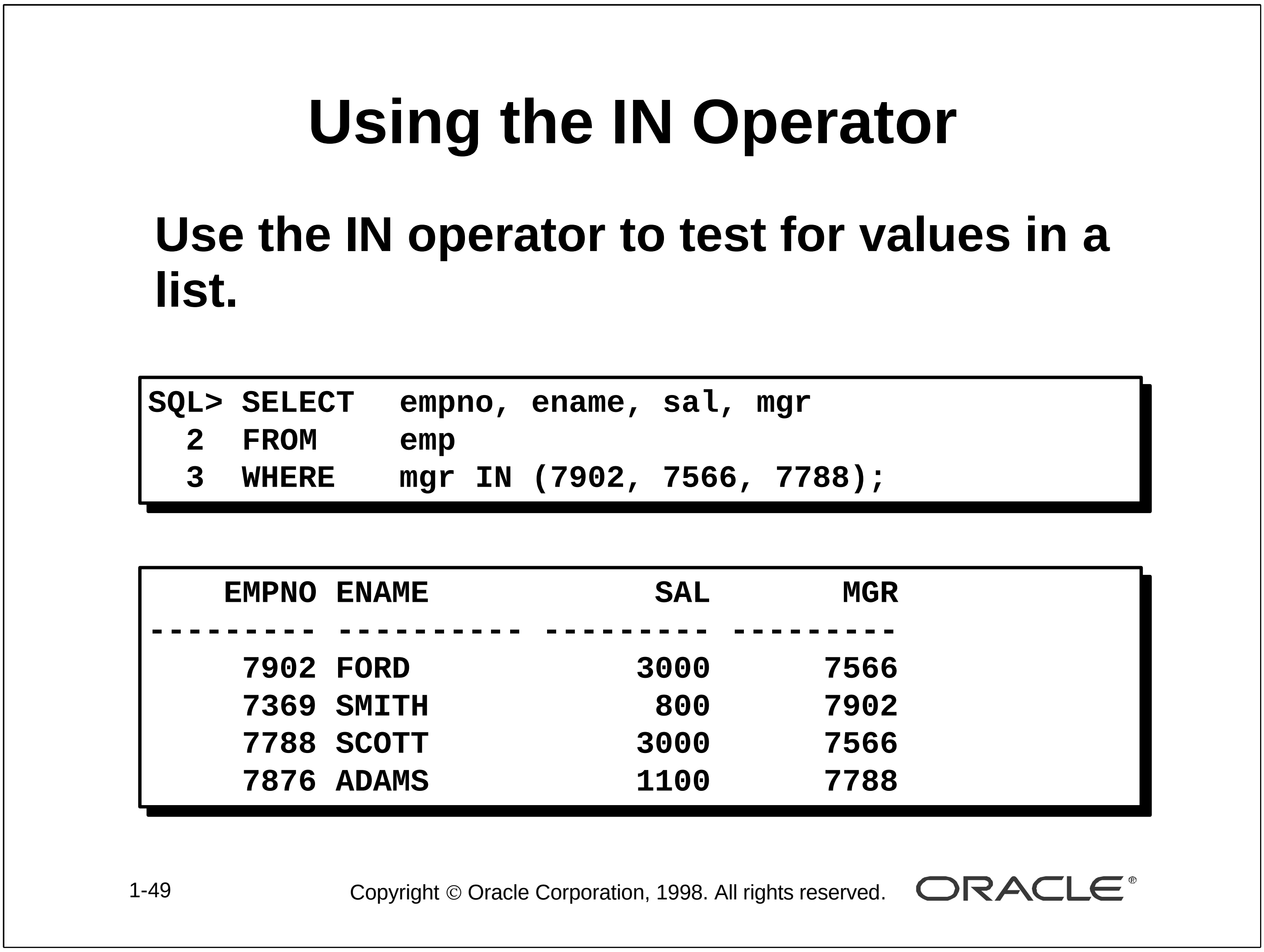
The SELECT statement above returns rows from the EMP table for any employee whose salary is between $1000 and $1500.

Values specified with the BETWEEN operator are inclusive. You must specify the lower limit first.

### Class Management Note

Emphasize that the values specified with the BETWEEN operator in the example are inclusive. Point out that Turner who earns $1500 (higher limit) is included in the output.

Demo: *l2betw.sql*



# Using the IN Operator

## Use the IN operator to test for values in a list.

```
SQL> SELECT   empno, ename, sal, mgr
  2  FROM     emp
  3  WHERE    mgr IN (7902, 7566, 7788);
```

```
    EMPNO ENAME            SAL       MGR
--------- ---------- --------- ---------
     7902 FORD            3000      7566
     7369 SMITH            800      7902
     7788 SCOTT           3000      7566
     7876 ADAMS           1100      7788
```

1-49

### The IN Operator

To test for values in a specified list, use the IN operator.

The above example displays employee number, name, salary, and manager’s employee number of all the employees whose manager’s employee number is 7902, 7566, or 7788.

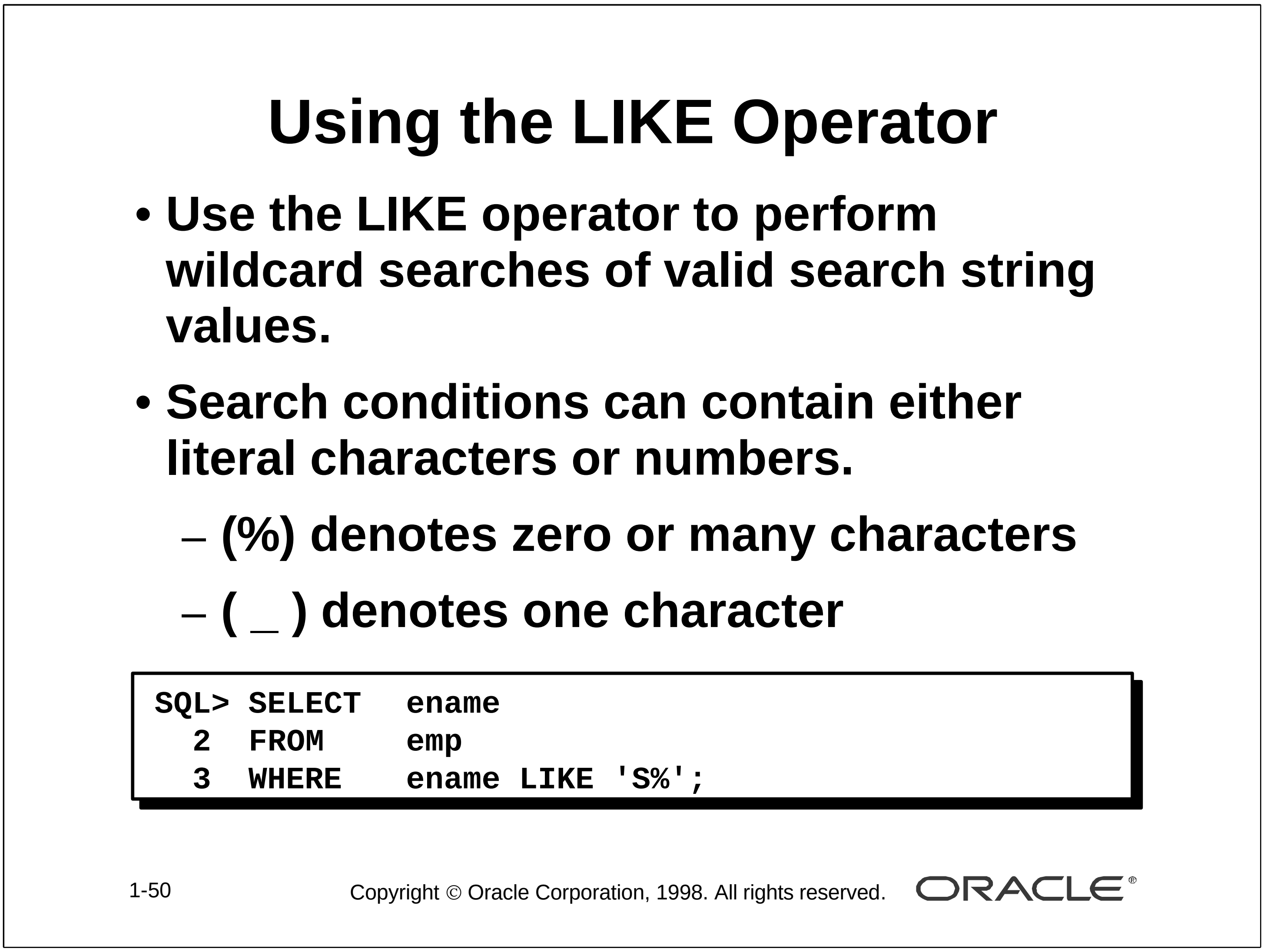
The IN operator can be used with any datatype. The following example returns a row from the EMP table for any employee whose name is included in the list of names in the WHERE clause.

```
SQL> SELECT   empno,  ename,  mgr, deptno
  2   FROM     emp
  3   WHERE    ename IN (’FORD’ , ’ALLEN’);
```

If characters or dates are used in the list, they must be enclosed in single quotation marks (‘ ’).

### Class Management Note

Demo: *l2in.sql*



# Using the LIKE Operator

- Use the LIKE operator to perform wildcard searches of valid search string values.
- Search conditions can contain either literal characters or numbers.
  - (%) denotes zero or many characters
  - ( _ ) denotes one character

```
SQL> SELECT   ename
  2  FROM     emp
  3  WHERE    ename LIKE 'S%';
```

### The LIKE Operator

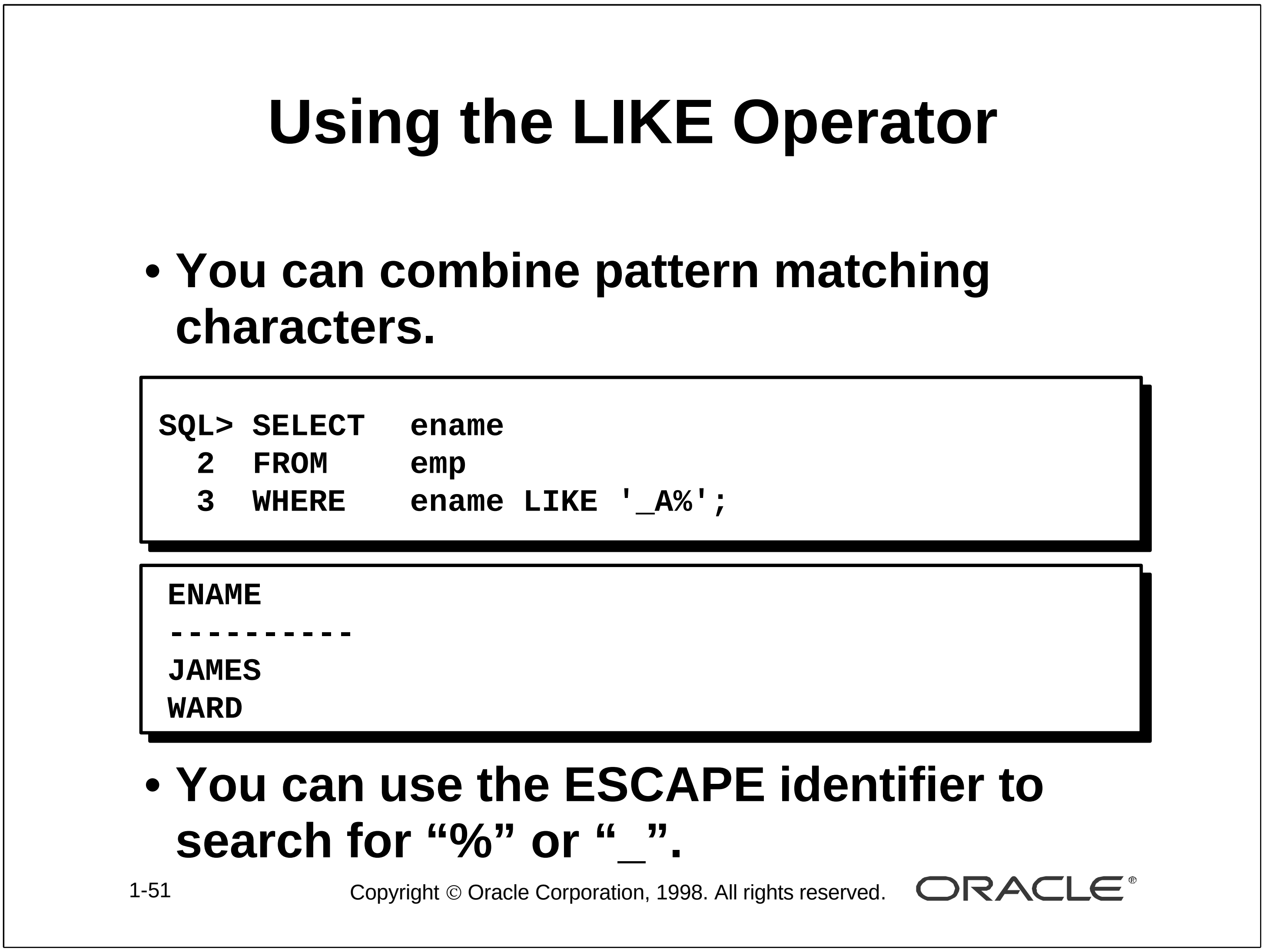
You may not always know the exact value to search for. You can select rows that match a character pattern by using the LIKE operator. The character pattern matching operation is referred to as a ***wildcard*** search. Two symbols can be used to construct the search string.

| Symbol | Description |
|---|---|
| % | Represents any sequence of zero or more characters |
| _ | Represents any single character |

The SELECT statement above returns the employee name from the EMP table for any employee whose name begins with an “S”. Note the uppercase “S.” Names beginning with an “s” will not be returned.

The LIKE operator can be used as a shortcut for some BETWEEN comparisons. The following example displays names and hiredates of all employees who joined in 81.

```
SQL>  SELECT    ename, hiredate
  2   FROM      emp
  3   WHERE     hiredate LIKE ’%81’;
```



# Using the LIKE Operator

- You can combine pattern matching characters.

```
SQL> SELECT   ename
  2  FROM     emp
  3  WHERE    ename LIKE '_A%';
```

```
ENAME
----------
JAMES
WARD
```

- You can use the ESCAPE identifier to search for “%” or “_”.

1-51

### Combining Wildcard Characters

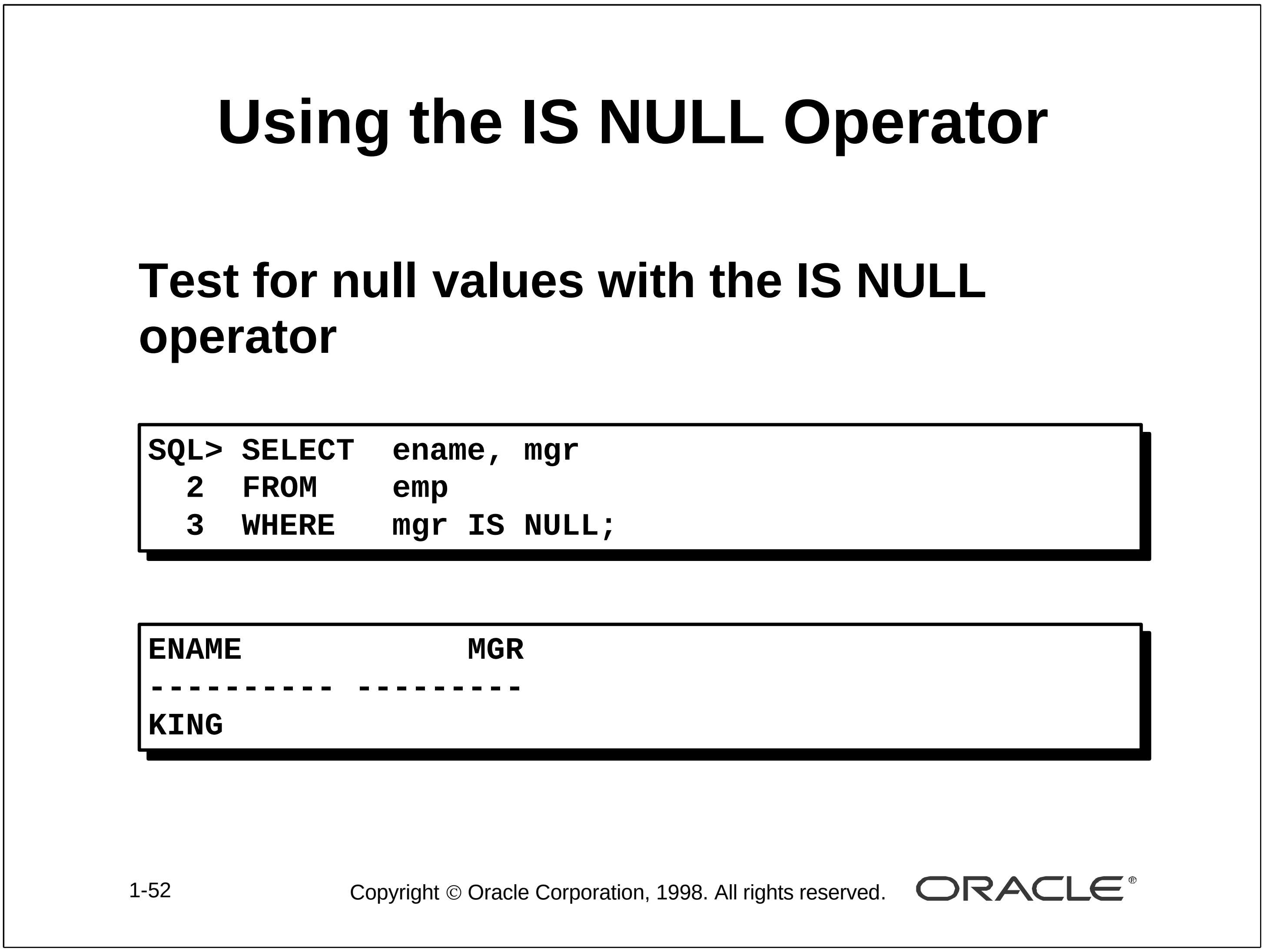
The % and _ symbols can be used in any combination with literal characters. The example on the slide displays the names of all employees whose name has an “A” as the second character.

### The ESCAPE Option

When you need to have an exact match for the actual “%” and “_” characters, use the ESCAPE option. This option specifies what the ESCAPE character is. To display the names of employees whose name contains “A_B”, use the following SQL statement:

```
SQL> SELECT    name
  2  FROM      emp
  3  WHERE     name  LIKE  '%A\_%B'  ESCAPE  '\';
```

The ESCAPE option identifies the backslash (\) as the escape character. In the pattern, the escape character precedes the underscore (_). This causes the Oracle8 Server to interpret the underscore literally.



# Using the IS NULL Operator

## Test for null values with the IS NULL operator

```
SQL> SELECT  ename, mgr
  2  FROM    emp
  3  WHERE   mgr IS NULL;
```

```
ENAME            MGR
---------- ---------
KING
```

1-52

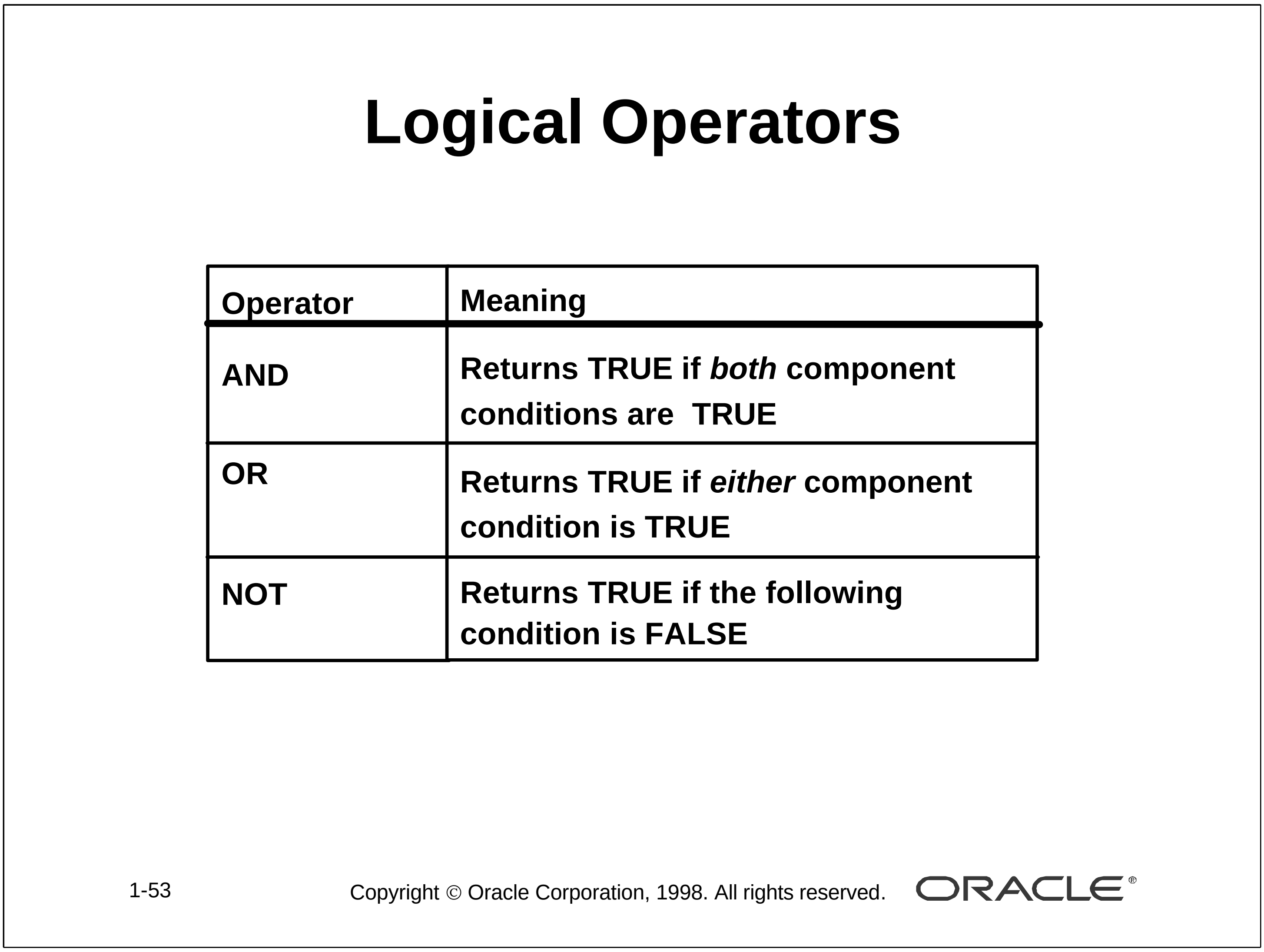
### The IS NULL Operator

The IS NULL operator tests for values that are null. A null value means the value is unavailable, unassigned, unknown, or inapplicable. Therefore, you cannot test with (=) because a null value cannot be equal or unequal to any value. The example above retrieves the name and manager of all employees who do not have a manager.

For example, to display name, job title, and commission for all employees who are not entitled to get a commission, use the following SQL statement:

```
SQL> SELECT    ename,  job, comm
  2  FROM      emp
  3  WHERE     comm  IS  NULL;
```

```
ENAME    JOB           COMM
-------- ---------- ------
KING     PRESIDENT
BLAKE    MANAGER
CLARK    MANAGER
...
```



# Logical Operators

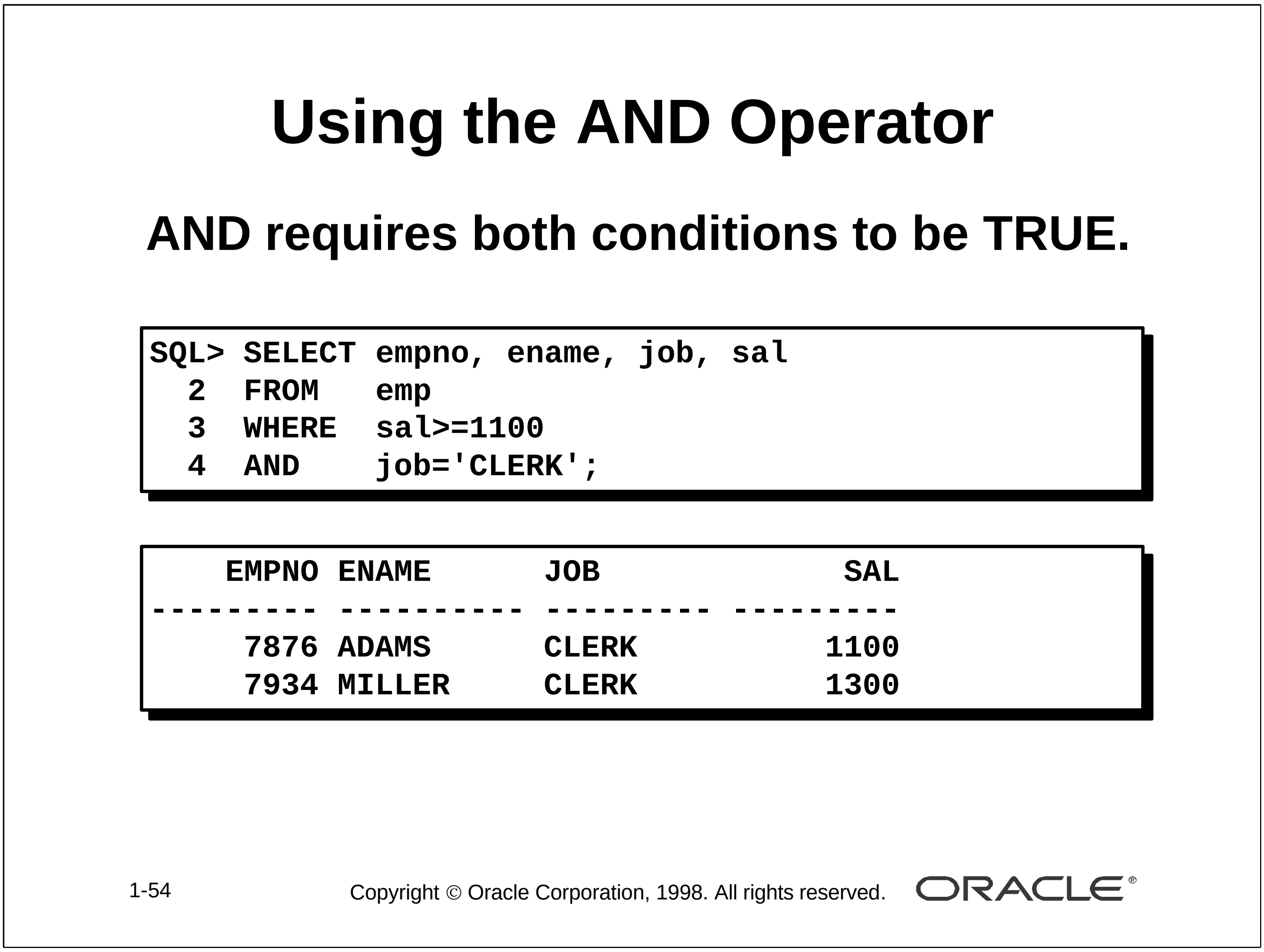
| Operator | Meaning |
|---|---|
| AND | Returns TRUE if *both* component conditions are TRUE |
| OR | Returns TRUE if *either* component condition is TRUE |
| NOT | Returns TRUE if the following condition is FALSE |

### Logical Operators

A logical operator combines the result of two component conditions to produce a single result based on them or to invert the result of a single condition. Three logical operators are available in SQL:

- AND
- OR
- NOT

All the examples so far have specified only one condition in the WHERE clause. You can use several conditions in one WHERE clause using the AND and OR operators.



# Using the AND Operator

## AND requires both conditions to be TRUE.

```
SQL> SELECT empno, ename, job, sal
  2  FROM   emp
  3  WHERE  sal>=1100
  4  AND    job='CLERK';
```

```
    EMPNO ENAME      JOB             SAL
--------- ---------- --------- ---------
     7876 ADAMS      CLERK          1100
     7934 MILLER     CLERK          1300
```

1-54

ORACLE®

### The AND Operator

In the example, both conditions must be true for any record to be selected. Therefore, an employee who has a job title of CLERK *and* earns more than $1100 will be selected.

All character searches are case sensitive. No rows are returned if CLERK is not in uppercase. Character strings must be enclosed in quotation marks.

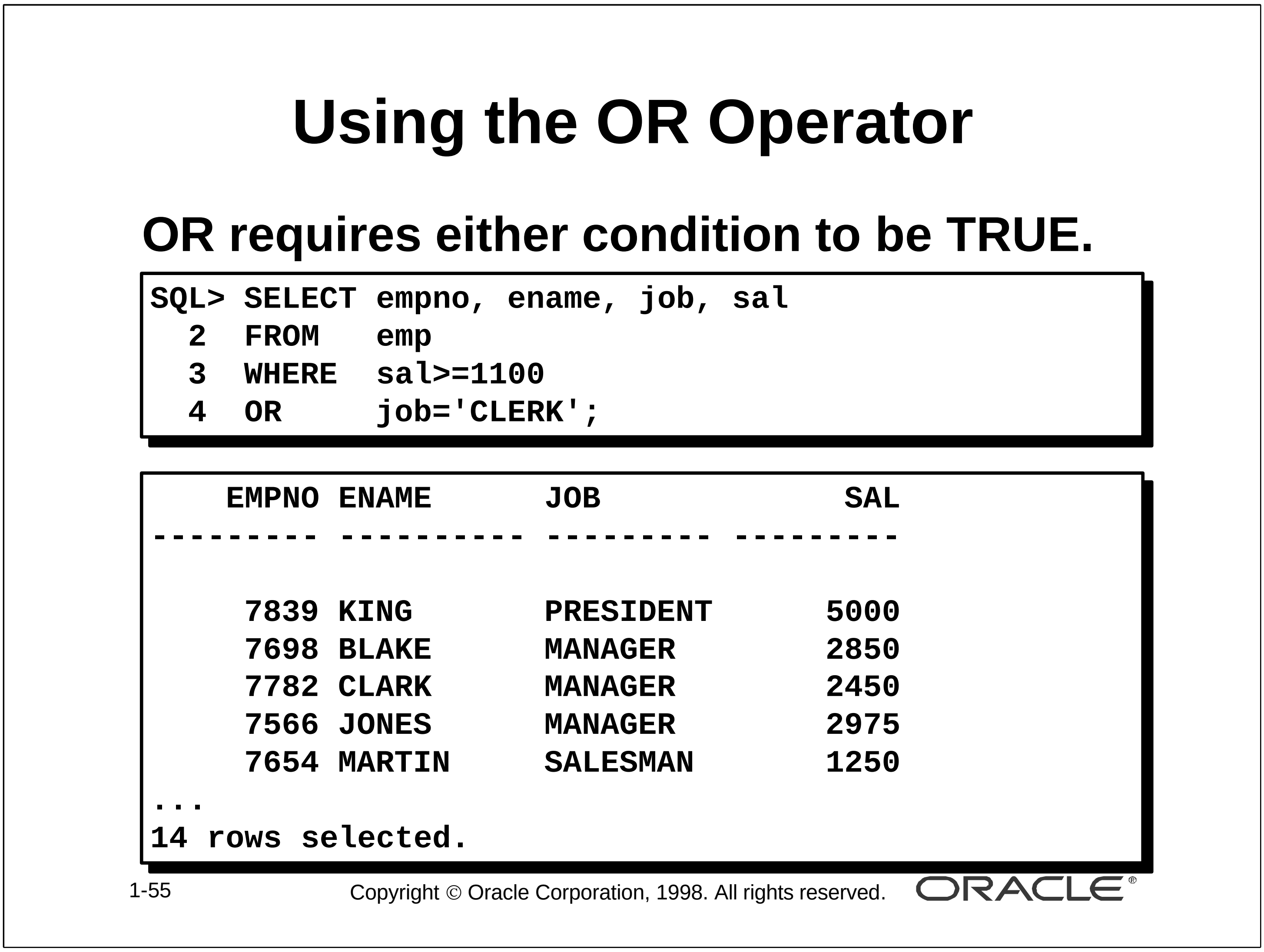
### AND Truth Table

The following table shows the results of combining two expressions with AND:

| **AND** | **TRUE** | **FALSE** | **UNKNOWN** |
|---|---|---|---|
| **TRUE** | TRUE | FALSE | UNKNOWN |
| **FALSE** | FALSE | FALSE | FALSE |
| **UNKNOWN** | UNKNOWN | FALSE | UNKNOWN |

### Class Management Note

Demo: *l2and.sql*



# Using the OR Operator

## OR requires either condition to be TRUE.

```
SQL> SELECT empno, ename, job, sal
  2  FROM   emp
  3  WHERE  sal>=1100
  4  OR     job='CLERK';
```

```
    EMPNO ENAME      JOB             SAL
--------- ---------- --------- ---------

     7839 KING       PRESIDENT      5000
     7698 BLAKE      MANAGER        2850
     7782 CLARK      MANAGER        2450
     7566 JONES      MANAGER        2975
     7654 MARTIN     SALESMAN       1250
...
14 rows selected.
```

### The OR Operator

In the example, either condition can be true for any record to be selected. Therefore, an employee who has a job title of CLERK *or* earns more than $1100 will be selected.

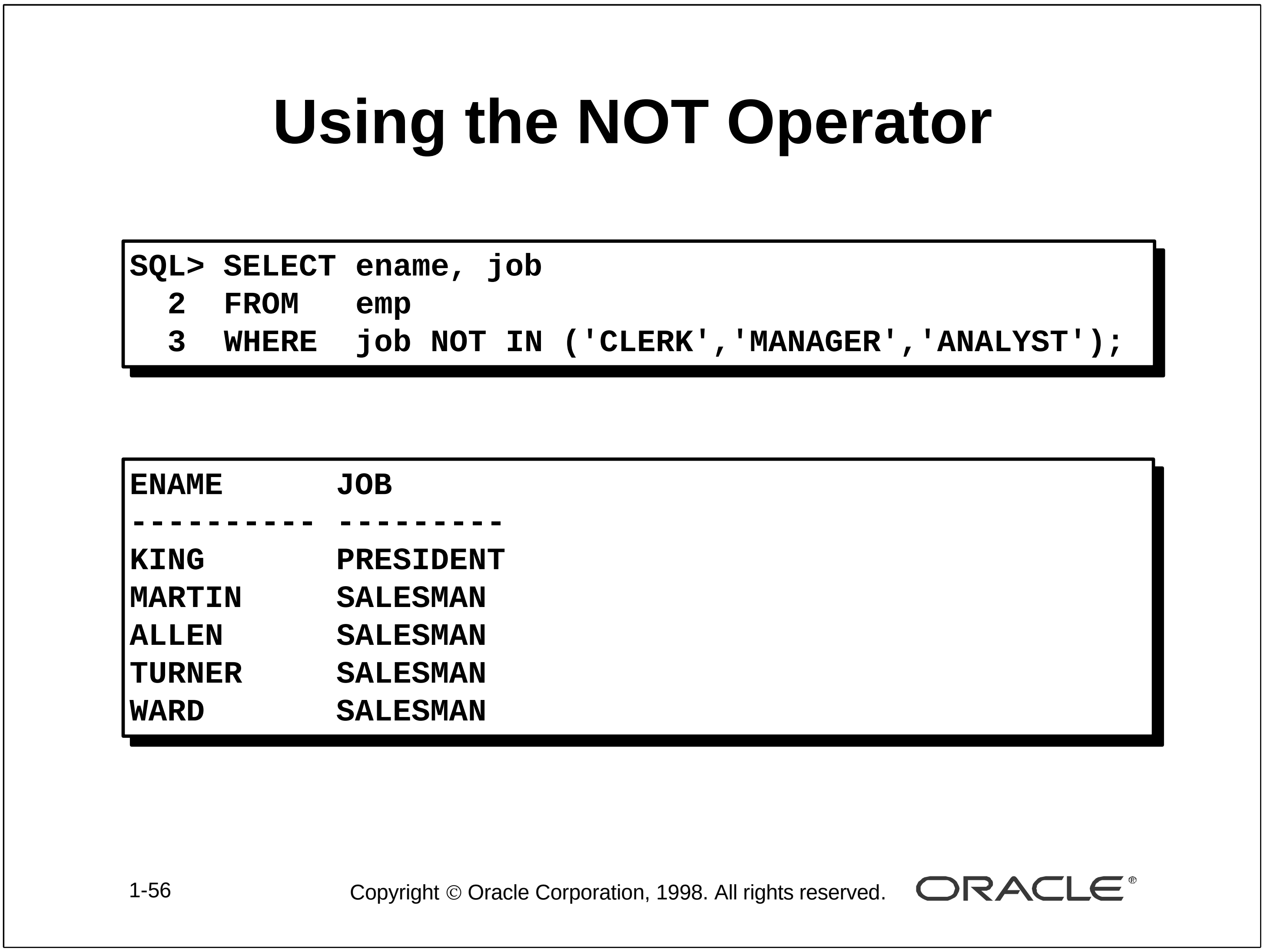
### OR Truth Table

The following table shows the results of combining two expressions with OR:

| **OR** | **TRUE** | **FALSE** | **UNKNOWN** |
|---|---|---|---|
| **TRUE** | TRUE | TRUE | TRUE |
| **FALSE** | TRUE | FALSE | UNKNOWN |
| **UNKNOWN** | TRUE | UNKNOWN | UNKNOWN |

### Class Management Note

Demo: *l2or.sql*



# Using the NOT Operator

```
SQL> SELECT ename, job
  2  FROM   emp
  3  WHERE  job NOT IN ('CLERK','MANAGER','ANALYST');
```

```
ENAME      JOB
---------- ---------
KING       PRESIDENT
MARTIN     SALESMAN
ALLEN      SALESMAN
TURNER     SALESMAN
WARD       SALESMAN
```

### The NOT Operator

The example above displays name and job title of all the employees whose job title *is not* CLERK, MANAGER, or ANALYST.

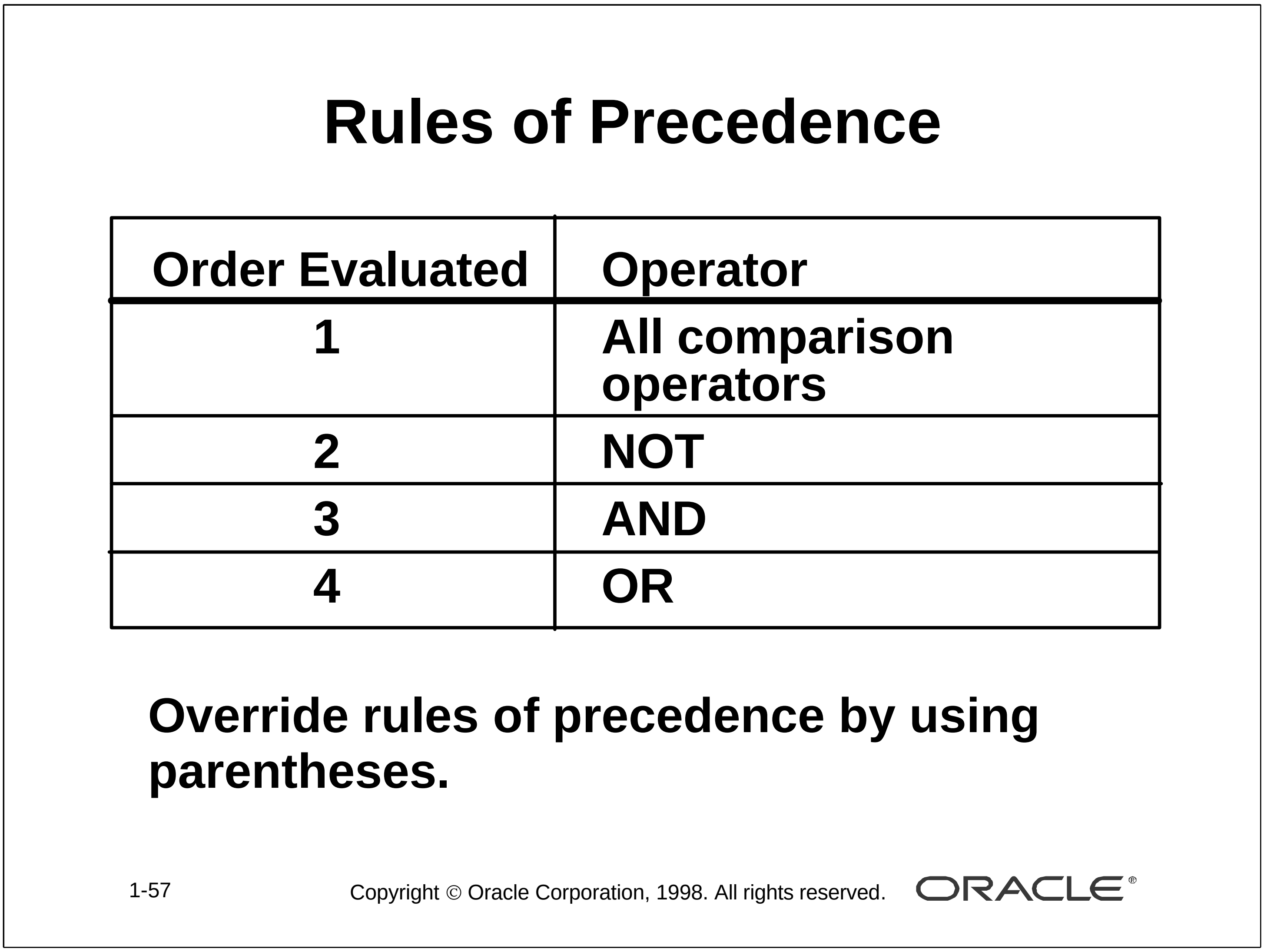
### NOT Truth Table

The following table shows the result of applying the NOT operator to a condition:

| NOT | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| | FALSE | TRUE | UNKNOWN |

**Note:** The NOT operator can also be used with other SQL operators such as BETWEEN, LIKE, and NULL.

```
... WHERE  sal  NOT  BETWEEN  1000 AND  1500
... WHERE  ename NOT LIKE ’%A%’
... WHERE  comm  IS  NOT  NULL
```



# Rules of Precedence

| Order Evaluated | Operator |
|---|---|
| 1 | All comparison operators |
| 2 | NOT |
| 3 | AND |
| 4 | OR |

Override rules of precedence by using parentheses.

1-57

# Rules of Precedence

```
SQL> SELECT ename, job, sal
  2  FROM   emp
  3  WHERE  job='SALESMAN'
  4  OR     job='PRESIDENT'
  5  AND    sal>1500;
```

```
ENAME      JOB             SAL
---------- --------- ---------
KING       PRESIDENT      5000
MARTIN     SALESMAN       1250
ALLEN      SALESMAN       1600
TURNER     SALESMAN       1500
WARD       SALESMAN       1250
```

### Example of Precedence of AND Operator

In the example, there are two conditions:

- The first condition is that job is PRESIDENT *and* salary is greater than 1500.
- The second condition is that job is SALESMAN.

Therefore, the SELECT statement reads as follows:

"Select the row if an employee is a PRESIDENT *and* earns more than $1500 *or* if the employee is a SALESMAN."

### Class Management Note

Demo: *l2sal1.sql*

# Rules of Precedence

## Use parentheses to force priority.

```
SQL> SELECT   ename, job, sal
  2  FROM     emp
  3  WHERE    (job='SALESMAN'
  4  OR       job='PRESIDENT')
  5  AND      sal>1500;
```

```
ENAME      JOB             SAL
---------- --------- ---------
KING       PRESIDENT      5000
ALLEN      SALESMAN       1600
```

### Using Parentheses

In the example, there are two conditions:

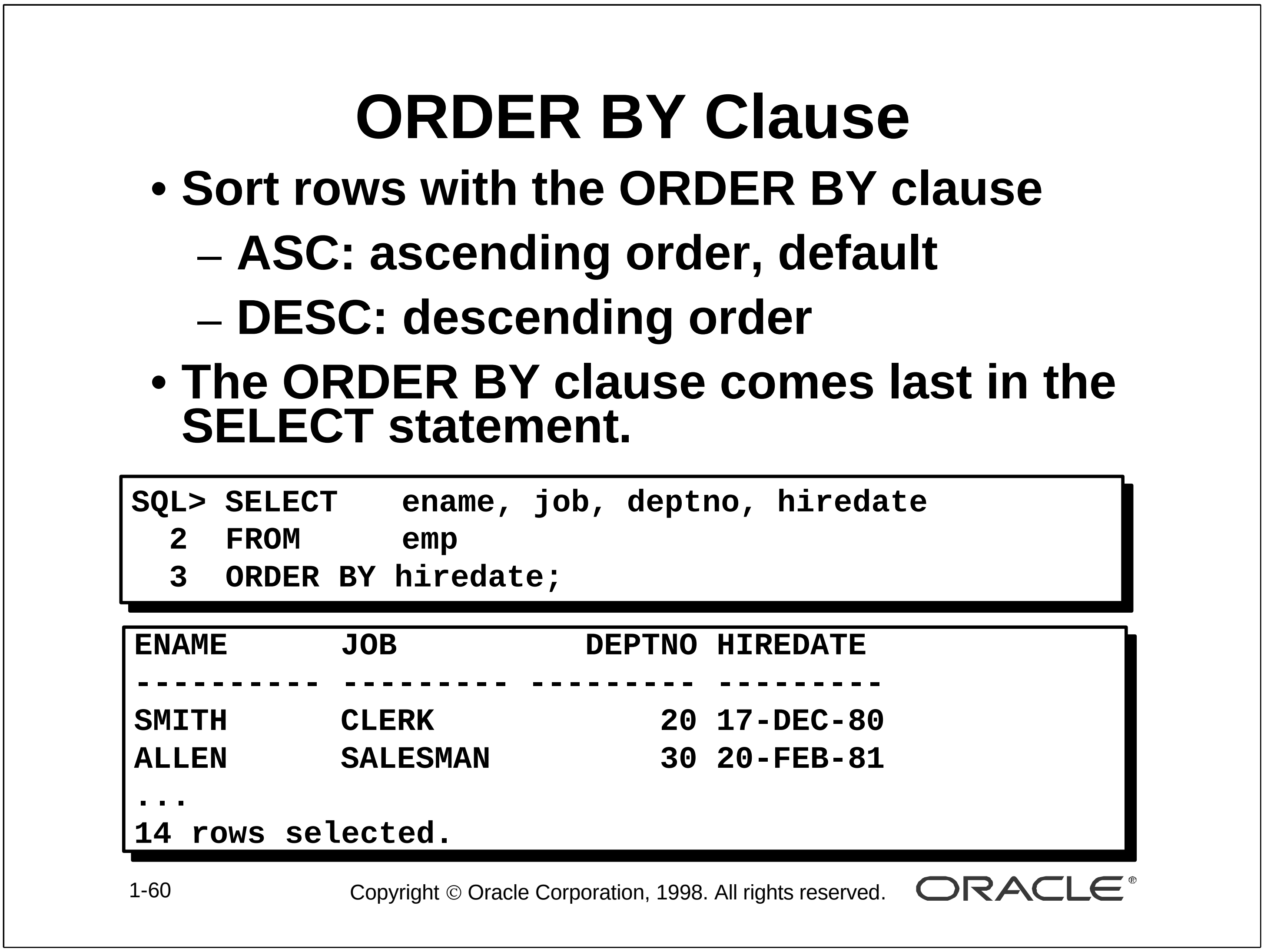
- The first condition is that job is PRESIDENT *or* SALESMAN.
- The second condition is that salary is greater than 1500.

Therefore, the SELECT statement reads as follows:

“Select the row if an employee is a PRESIDENT or a SALESMAN and if the employee earns more than $1500.”

### Class Management Note

Demo: *l2sal2.sql*



# ORDER BY Clause

- Sort rows with the ORDER BY clause
  - ASC: ascending order, default
  - DESC: descending order
- The ORDER BY clause comes last in the SELECT statement.

```
SQL> SELECT    ename, job, deptno, hiredate
  2  FROM      emp
  3  ORDER BY hiredate;
```

```
ENAME      JOB          DEPTNO HIREDATE
---------- --------- --------- ---------
SMITH      CLERK            20 17-DEC-80
ALLEN      SALESMAN         30 20-FEB-81
...
14 rows selected.
```

1-60

### The ORDER BY Clause

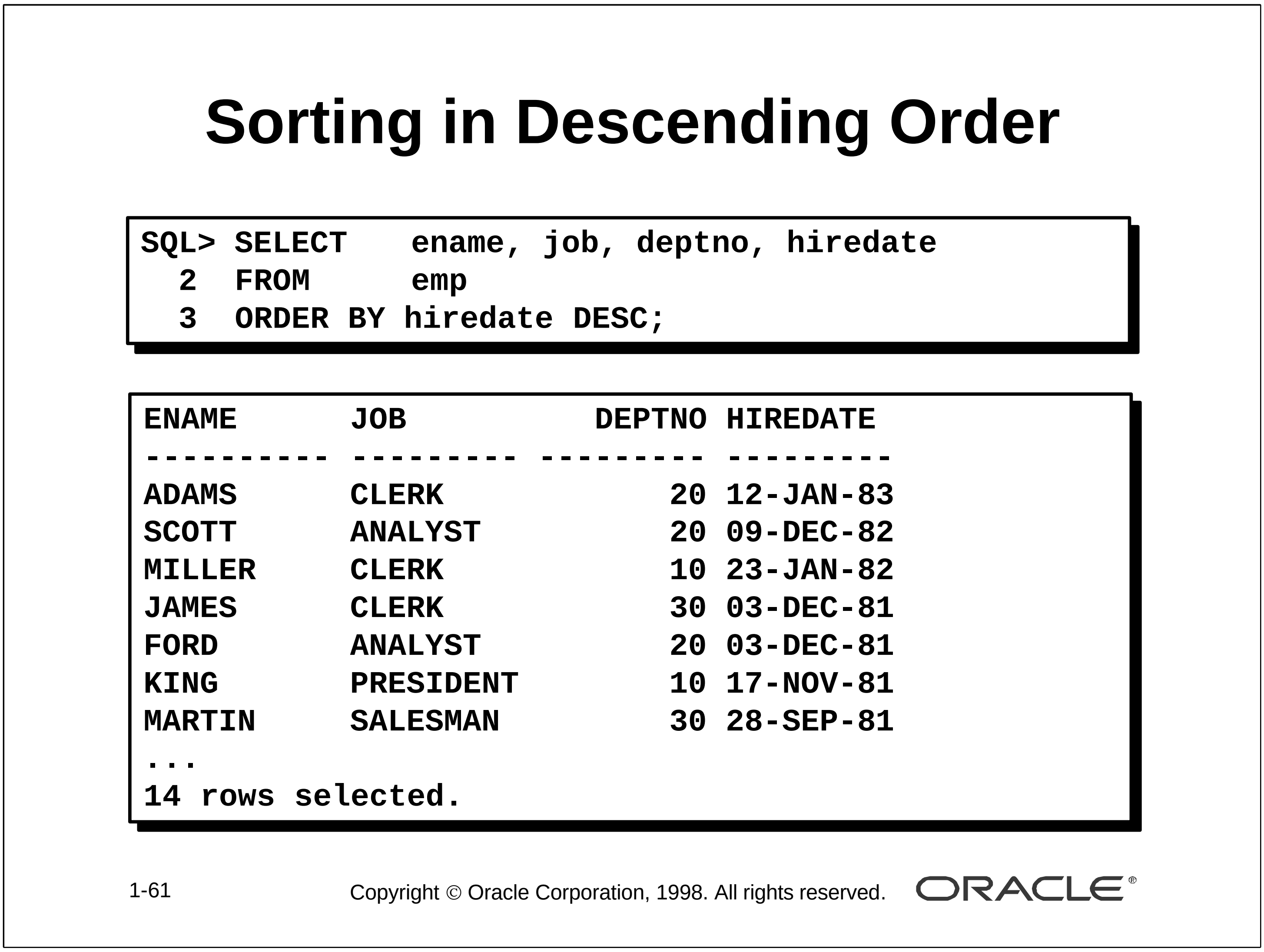
The order of rows returned in a query result is undefined. The ORDER BY clause can be used to sort the rows. If used, you must place the ORDER BY clause last. You can specify an expression or an alias to sort.

### Syntax

```
SELECT    expr
FROM      table
[WHERE    condition (s)]
[ORDER BY {column, expr} [ASC|DESC]];
```

| where: |  |  |
|---|---|---|
|  | ORDER BY | specifies the order in which the retrieved rows are displayed. |
|  | ASC | orders the rows in ascending order. This is the default order. |
|  | DESC | orders the rows in descending order. |

If the ORDER BY clause is not used, the sort order is undefined, and the Oracle8 Server may not fetch rows in the same order for the same query twice. Use the ORDER BY clause to display the rows in a specific order.



# Sorting in Descending Order

```
SQL> SELECT    ename, job, deptno, hiredate
  2  FROM      emp
  3  ORDER BY hiredate DESC;
```

```
ENAME      JOB          DEPTNO HIREDATE
---------- --------- --------- ---------
ADAMS      CLERK            20 12-JAN-83
SCOTT      ANALYST          20 09-DEC-82
MILLER     CLERK            10 23-JAN-82
JAMES      CLERK            30 03-DEC-81
FORD       ANALYST          20 03-DEC-81
KING       PRESIDENT        10 17-NOV-81
MARTIN     SALESMAN         30 28-SEP-81
...
14 rows selected.
```

### Default Ordering of Data

The default sort order is ascending:

- Numeric values are displayed with the lowest values first—for example, 1-999.
- Date values are displayed with the earliest value first—for example, 01-JAN-92 before 01-JAN-95.
- Character values are displayed in alphabetical order—for example, A first and Z last.
- Null values are displayed last for ascending sequences and first for descending sequences.

### Reversing the Default Order

To reverse the order in which rows are displayed, specify the keyword DESC after the column name in the ORDER BY clause. The example above sorts the result by the most recently hired employee.

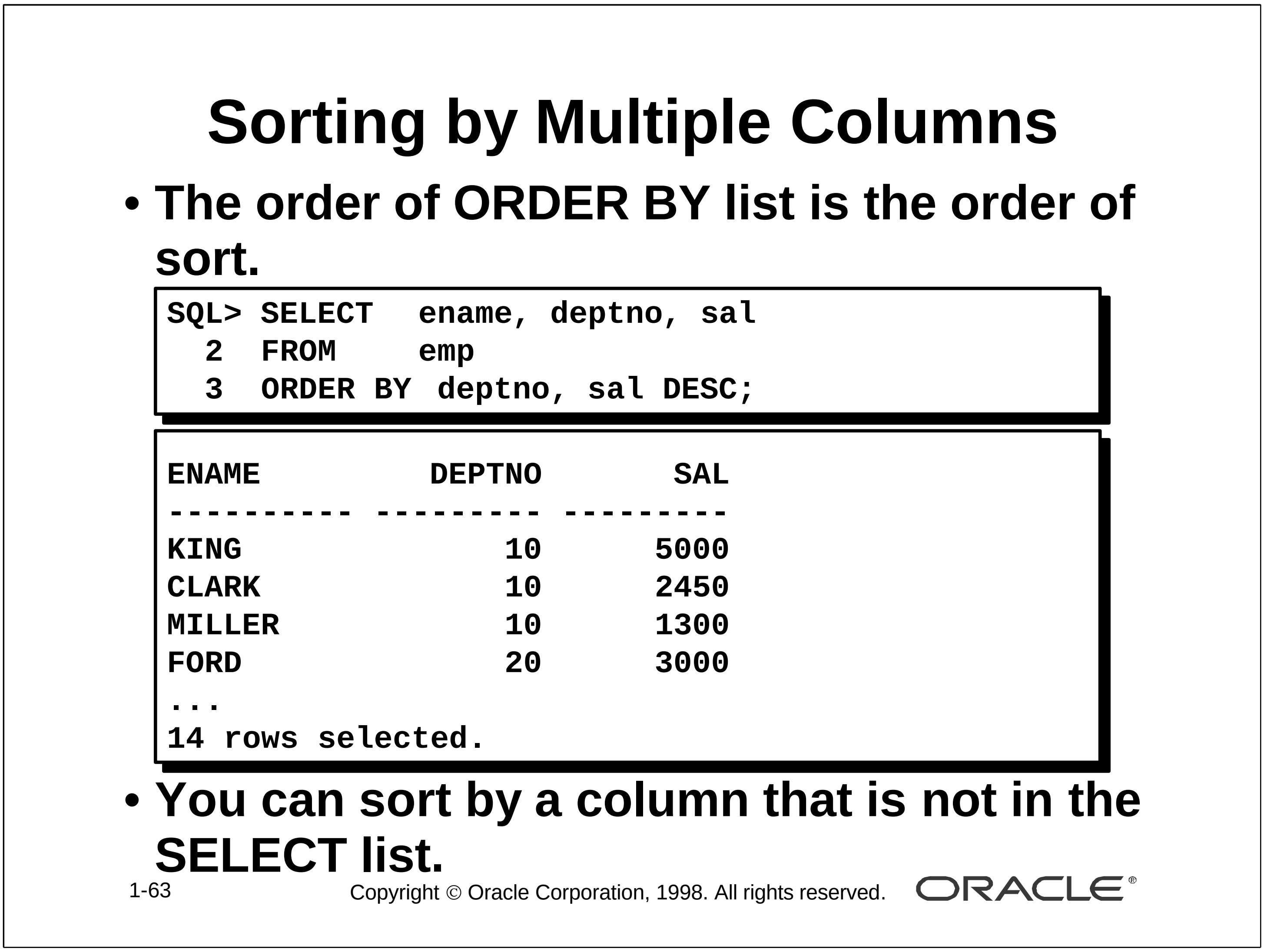
# Sorting by Column Alias

```
SQL> SELECT   empno, ename, sal*12 annsal
  2  FROM     emp
  3  ORDER BY annsal;
```

```
    EMPNO ENAME         ANNSAL
--------- ---------- ---------
     7369 SMITH           9600
     7900 JAMES          11400
     7876 ADAMS          13200
     7654 MARTIN         15000
     7521 WARD           15000
     7934 MILLER         15600
     7844 TURNER         18000
...
14 rows selected.
```

### Sorting By Column Aliases

You can use a column alias in the ORDER BY clause. The above example sorts the data by annual salary.



# Sorting by Multiple Columns

- The order of ORDER BY list is the order of sort.

```
SQL> SELECT   ename, deptno, sal
  2  FROM     emp
  3  ORDER BY deptno, sal DESC;
```

```
ENAME         DEPTNO       SAL
---------- --------- ---------
KING              10      5000
CLARK             10      2450
MILLER            10      1300
FORD              20      3000
...
14 rows selected.
```

- You can sort by a column that is not in the SELECT list.

1-63

ORACLE®

### Sorting by Multiple Columns

You can sort query results by more than one column. The sort limit is the number of columns in the given table.

In the ORDER BY clause, specify the columns, and separate the column names using commas. If you want to reverse the order of a column, specify DESC after its name. You can order by columns that are not included in the SELECT clause.

### Example

Display name and salary of all employees. Order the result by department number and then descending order by salary.

```
SQL> SELECT    ename,  salary
  2  FROM      emp
  3  ORDER BY  deptno, sal DESC;
```

### Class Management Note

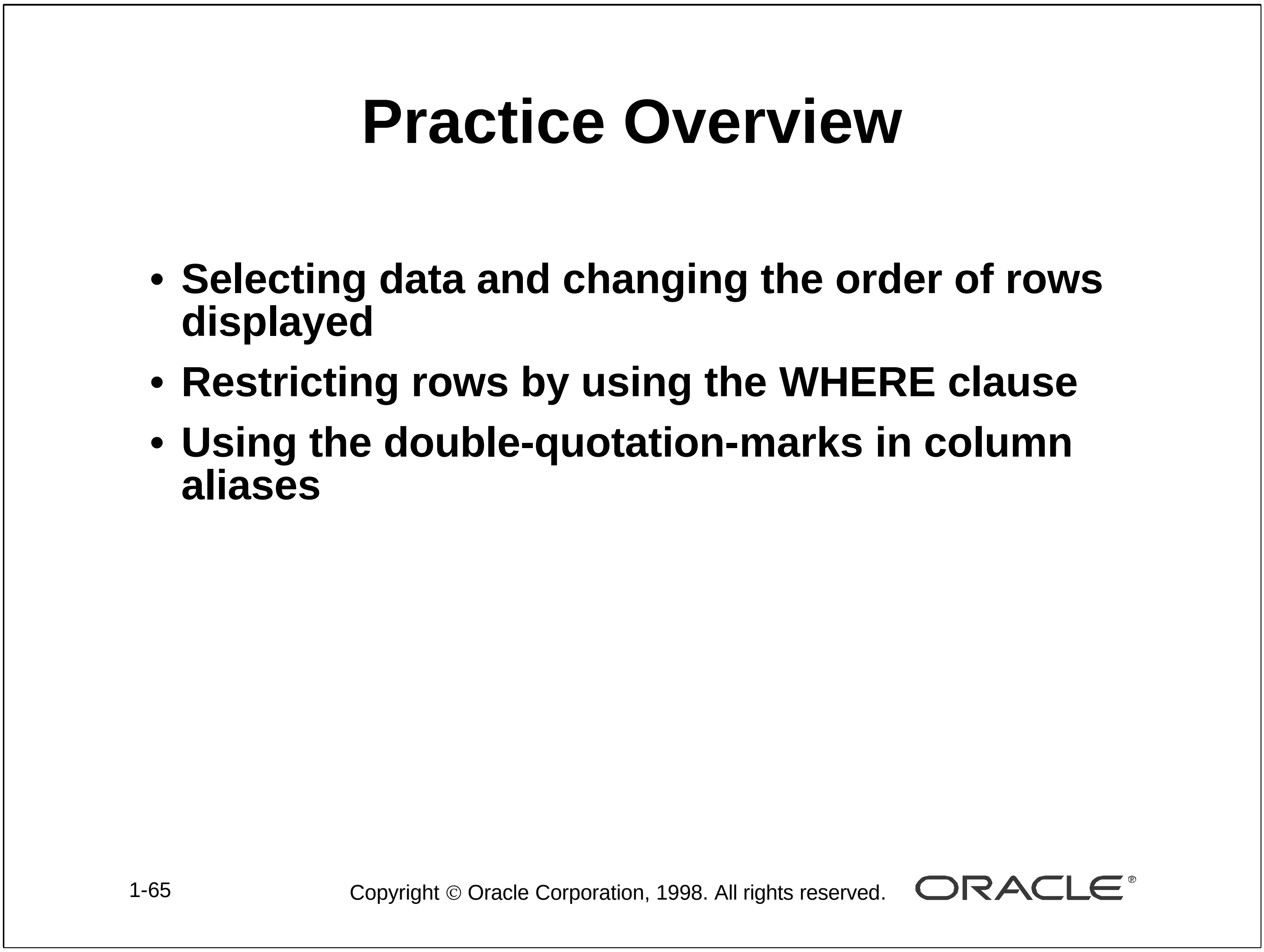
Show that the DEPTNO column is sorted in ascending order and the SAL column in descending order.

# Summary

```
SELECT       [DISTINCT] {*, column [alias], ...}
FROM         table
[WHERE       condition(s)]
[ORDER BY    {column, expr, alias} [ASC|DESC]];
```

### Summary

In this lesson, you have learned about restricting and sorting rows returned by the SELECT statement. You have also learned how to implement various operators.



# Practice Overview

- Selecting data and changing the order of rows displayed
- Restricting rows by using the WHERE clause
- Using the double-quotation-marks in column aliases

**Practice Overview**

This practice gives you a variety of exercises using the WHERE clause and the ORDER BY clause.

## Practice 2

1. Create a query to display the name and salary of employees earning more than $2850. Save your SQL statement to a file named *p2q1.sql*. Run your query.

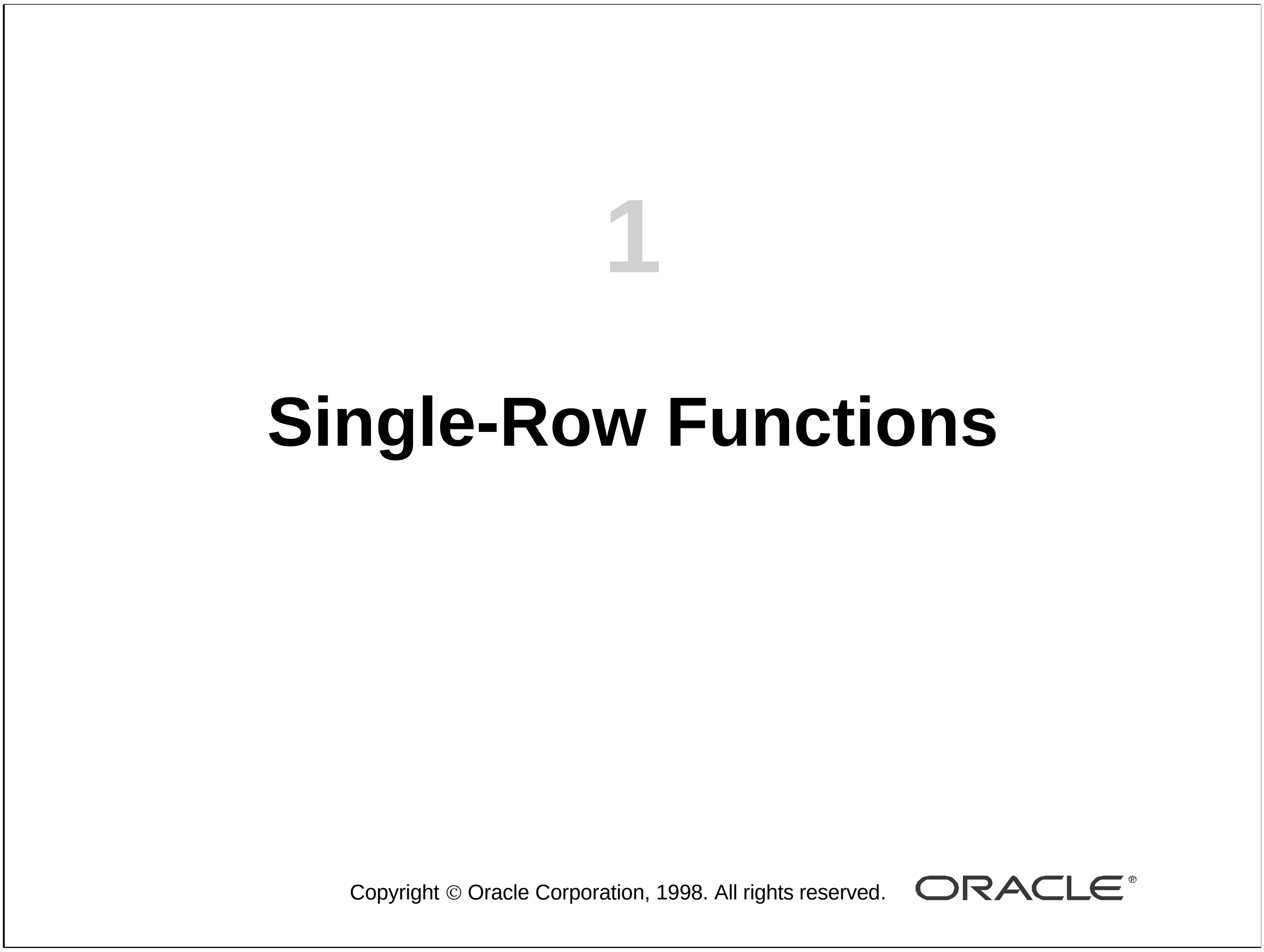
```
ENAME    SAL
-------- ----
KING     5000
JONES    2975
FORD     3000
SCOTT    3000
```

2. Create a query to display the employee name and department number for employee number 7566.

```
ENAME  DEPTNO
------ ------
JONES      20
```

3. Modify *p2q1.sql* to display the name and salary for all employees whose salary is not in the range of $1500 and $2850. Resave your SQL statement to a file named *p2q3.sql*. Rerun your query.

```
ENAME     SAL
------- -----
KING     5000
JONES    2975
MARTIN   1250
JAMES     950
WARD     1250
FORD     3000
SMITH     800
SCOTT    3000
ADAMS    1100
MILLER   1300
10 rows selected.
```



1

# Single-Row Functions

| Schedule: | Timing | Topic |
|---|---|---|
| | 55 minutes | Lecture |
| | 30 minutes | Practice |
| | 85 minutes | Total |

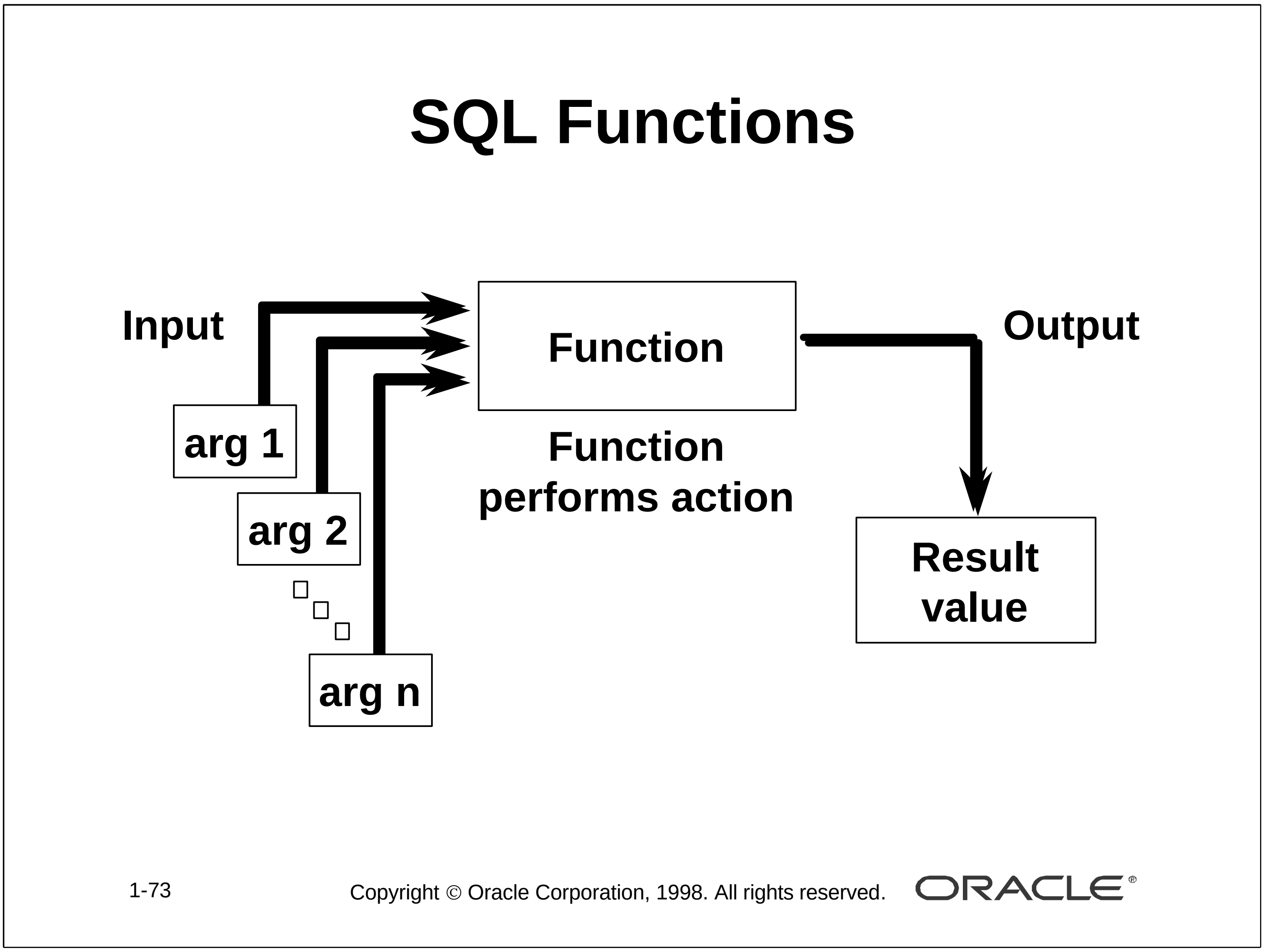
# Objectives

**At the end of this lesson, you should be able to:**

- **Describe various types of functions available in SQL**
- **Use character, number, and date functions in SELECT statements**
- **Describe the use of conversion functions**

**Lesson Aim**

Functions make the basic query block more powerful and are used to manipulate data values. This is the first of two lessons that explore functions. You will focus on single-row character, number, and date functions, as well as those functions that convert data from one type to another—for example, character data to numeric.



# SQL Functions

Input

arg 1

arg 2

□
□
□

arg n

Function

Function
performs action

Output

Result
value

1-73

ORACLE®

### SQL Functions

Functions are a very powerful feature of SQL and can be used to do the following:

- Perform calculations on data
- Modify individual data items
- Manipulate output for groups of rows
- Format dates and numbers for display
- Convert column datatypes

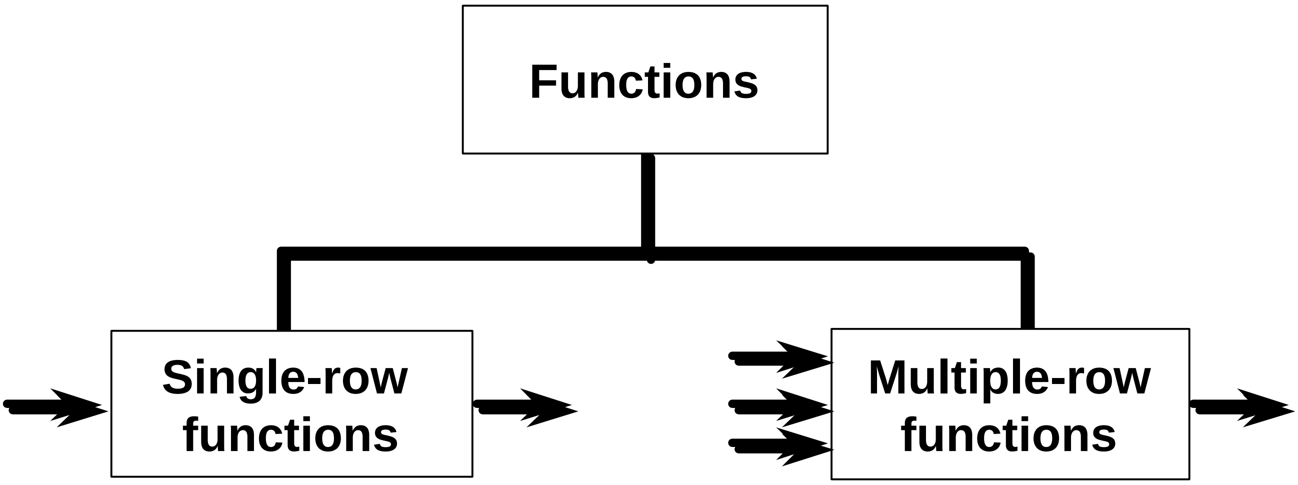
SQL functions accept argument(s) and return value(s).

**Note:** Most of the functions described in this lesson are specific to Oracle’s version of SQL. More on Oracle’s version of SQL is covered in the course, *Oracle SQL Specifics.*

### Class Management Note

This lesson does not discuss all functions in great detail. Present the most common functions without a long explanation.

# Two Types of SQL Functions



1-74

## SQL Functions

There are two distinct types of functions:

- Single-row functions
- Multiple-row functions

### Single-Row Functions

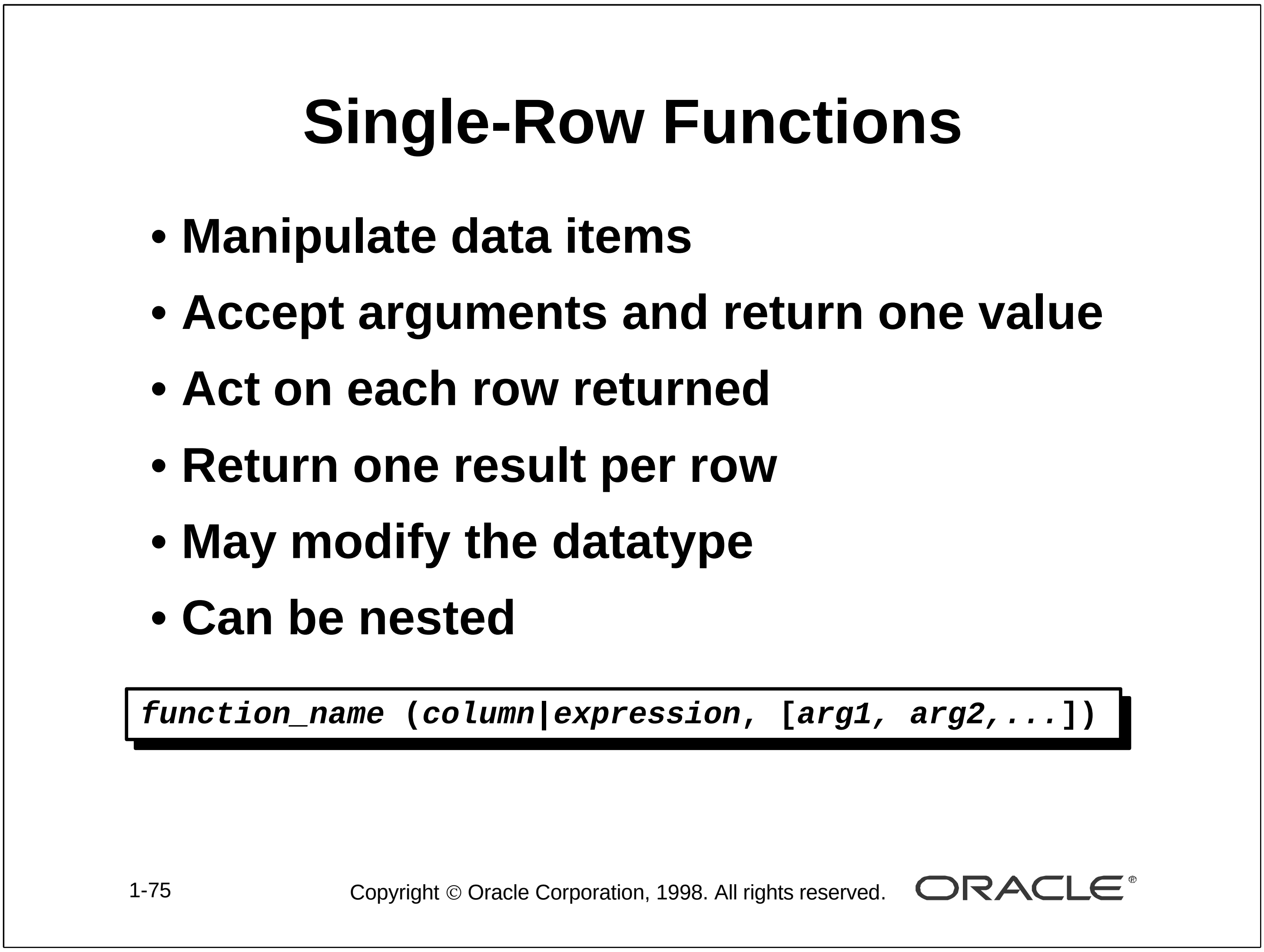
These functions operate on single-rows only and return one result per row. There are different types of single-row functions. This lesson covers those listed below.

- Character
- Number
- Date
- Conversion

### Multiple-Row Functions

These functions manipulate groups of rows to give one result per group of rows.

For more information, see *Oracle 8 Server SQL Reference, Release 8.0* for the complete list of available functions and syntax.



# Single-Row Functions

- Manipulate data items
- Accept arguments and return one value
- Act on each row returned
- Return one result per row
- May modify the datatype
- Can be nested

```
function_name (column|expression, [arg1, arg2,...])
```

1-75

ORACLE®

### Single-Row Functions

Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row returned by the query. An argument can be one of the following:

- A user-supplied constant
- A variable value
- A column name
- An expression

### Features of Single-Row Functions

- They act on each row returned in the query.
- They return one result per row.
- They may return a data value of a different type than that referenced.
- They may expect one or more arguments.
- You can use them in SELECT, WHERE, and ORDER BY clauses. You can nest them.

In the syntax:

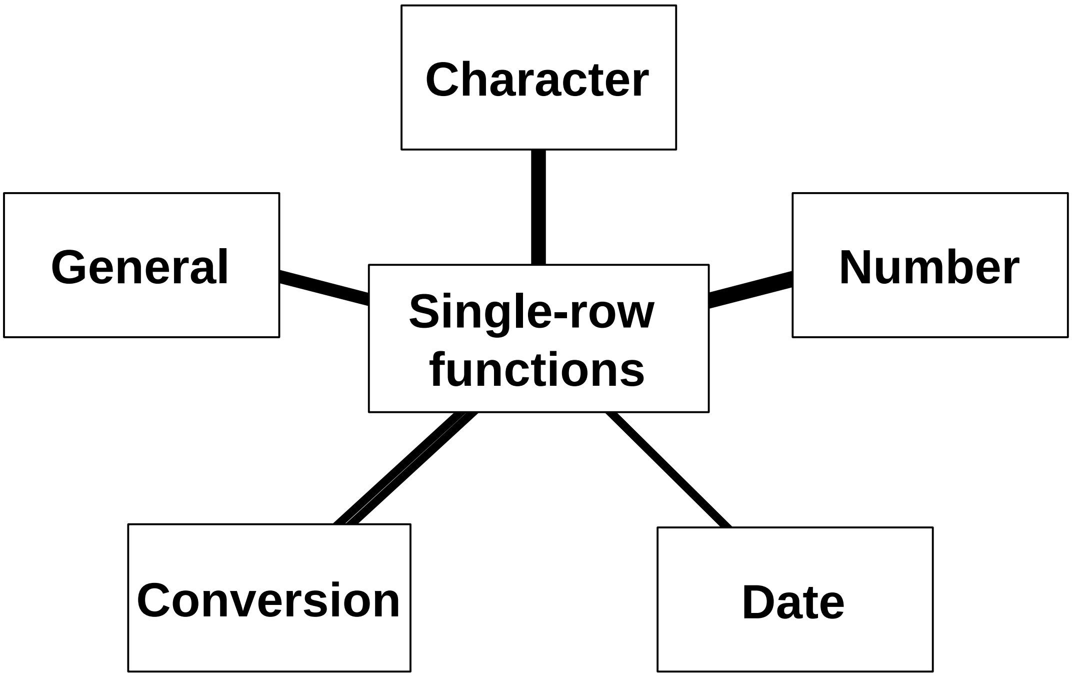
*function_name* is the name of the function

*column* is any named database column

*expression* is any character string or calculated expression

*arg1, arg2* is any argument to be used by the function

# Single-Row Functions



Character

General

Single-row functions

Number

Conversion

Date

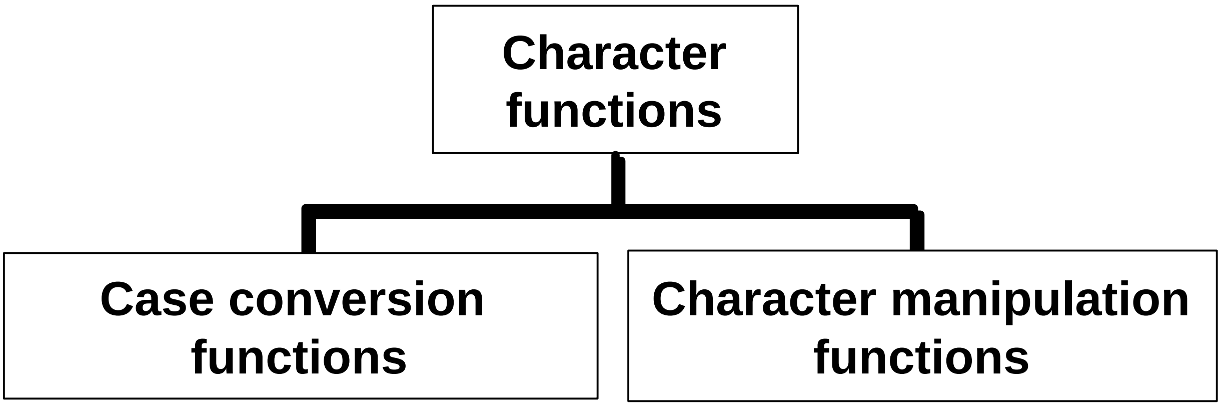
1-76

### Single-Row Functions

This lesson covers the following single-row functions:

- Character functions—Accept character input and can return both character and number values.
- Number functions—Accept numeric input and return numeric values.
- Date functions—Operate on values of the date datatype. All date functions return a value of date datatype except the MONTHS_BETWEEN function, which returns a number.
- Conversion functions—Convert a value from one datatype to another.
- General functions
  - NVL function
  - DECODE function

# Character Functions


Character functions

Case conversion functions

Character manipulation functions

**LOWER**

**UPPER**

**INITCAP**

**CONCAT**

**SUBSTR**

**LENGTH**

**INSTR**

**LPAD**

1-77

ORACLE®

## Character Functions

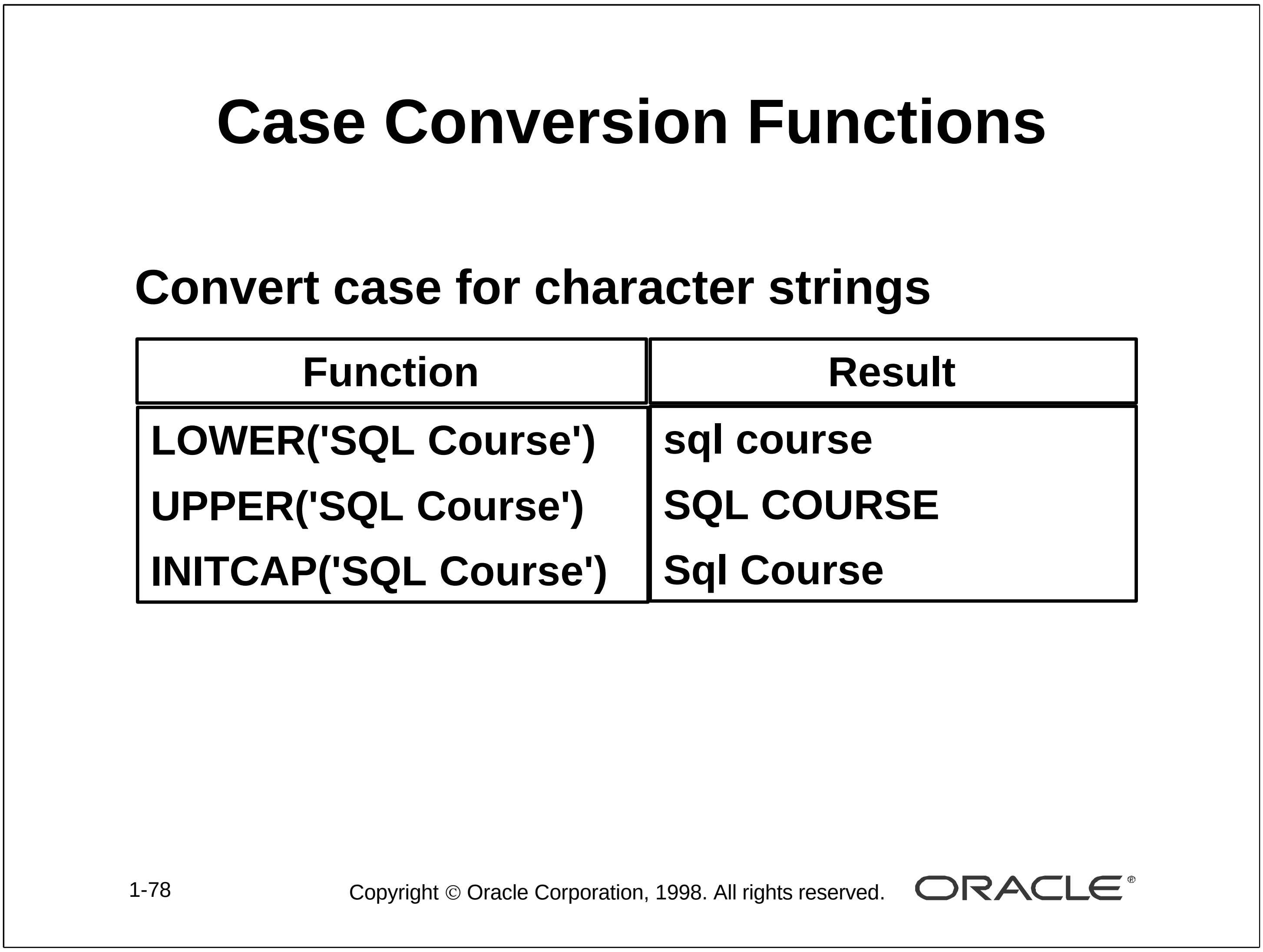
Single-row character functions accept character data as input and can return both character and number values. Character functions can be divided into:

- Case conversion functions
- Character manipulation functions

| **Function** | **Purpose** |
|---|---|
| LOWER(*column*\|*expression*) | Converts alpha character values to lowercase |
| UPPER(*column*\|*expression*) | Converts alpha character values to uppercase |
| INITCAP(*column*\|*expression*) | Converts alpha character values to uppercase for the first letter of each word, all other letters in lowercase |
| CONCAT(*column1*\|*expression1, column2*\|*expression2*) | Concatenates the first character value to the second character value. Equivalent to concatenation operator (\|\|) |
| SUBSTR(*column*\|*expression,m[,n]*) | Returns specified characters from character value starting at character position *m, n* characters long. If *m* is negative, the count starts from the end of the character value. If *n* is omitted, all characters to the end of the string are returned |
| LENGTH(*column*\|*expression*) | Returns the number of characters in value |
| INSTR(*column*\|*expression,m*) | Returns the numeric position of a named character |
| LPAD(*column*\|*expression, n,* ‘*string*’) | Pads the character value right-justified to a total width of *n* character positions |

**Note:** This list is a subset of the available character functions.

For more information, see
*Oracle8 Server SQL Reference, Release 8.0*, “Character Functions.”



# Case Conversion Functions

## Convert case for character strings

| Function | Result |
|---|---|
| LOWER('SQL Course') | sql course |
| UPPER('SQL Course') | SQL COURSE |
| INITCAP('SQL Course') | Sql Course |

1-78

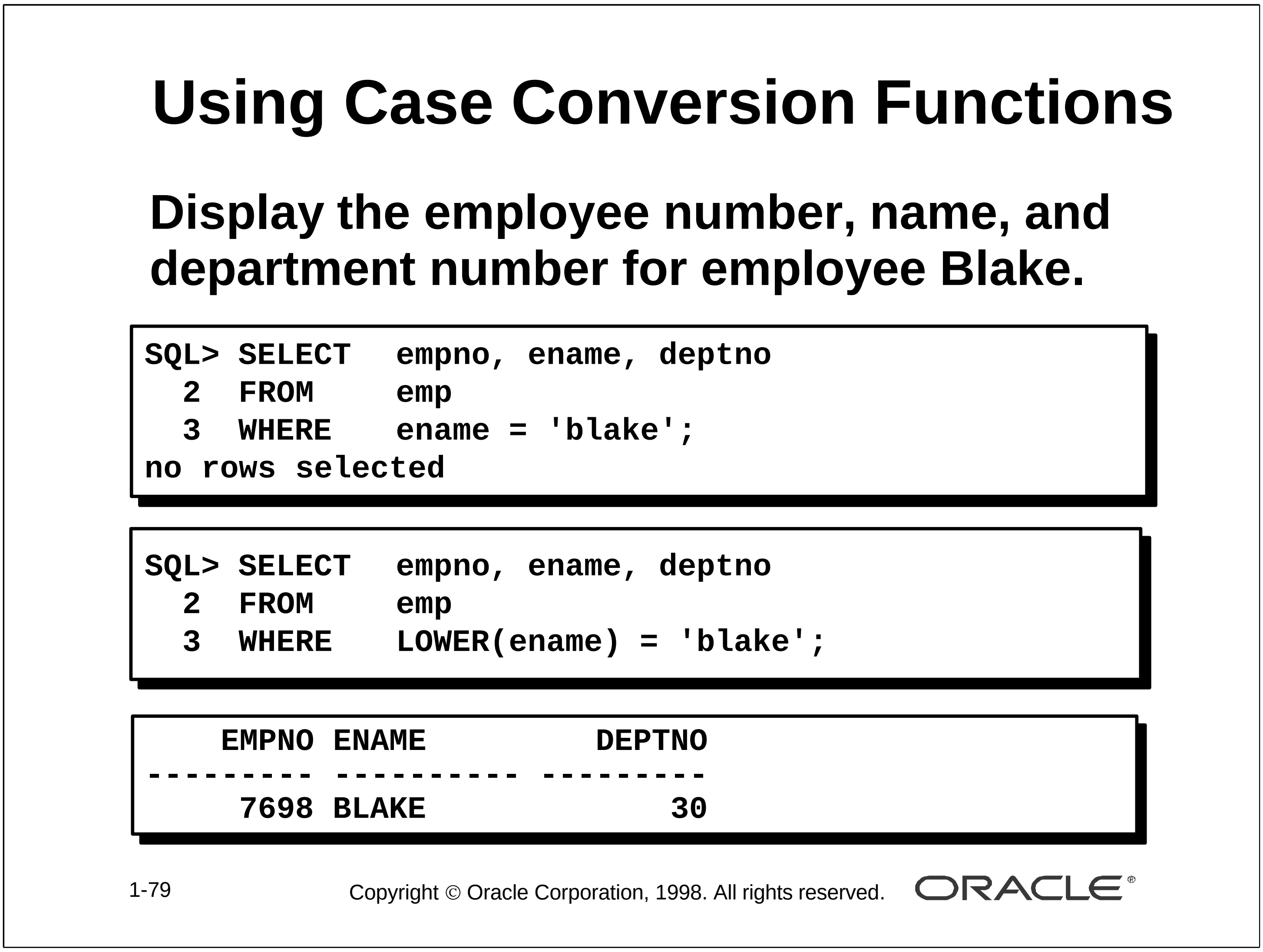
### Case Conversion Functions

LOWER, UPPER, and INITCAP are the three case conversion functions.

- LOWER—Converts mixed case or uppercase character string to lowercase
- UPPER—Converts mixed case or lowercase character string to uppercase
- INITCAP—Converts first letter of each word to uppercase and remaining letters to lowercase

```
SQL> SELECT  'The job title for '||INITCAP(ename)||' is '
  2               ||LOWER(job) AS "EMPLOYEE DETAILS"
  3  FROM     emp;
```

```
EMPLOYEE DETAILS
-----------------------------------------
The job title for King is president
The job title for Blake is manager
The job title for Clark is manager
...
14 rows selected.
```



# Using Case Conversion Functions

## Display the employee number, name, and department number for employee Blake.

```
SQL> SELECT   empno, ename, deptno
  2  FROM     emp
  3  WHERE    ename = 'blake';
no rows selected
```

```
SQL> SELECT   empno, ename, deptno
  2  FROM     emp
  3  WHERE    LOWER(ename) = 'blake';
```

```
    EMPNO ENAME         DEPTNO
--------- ---------- ---------
     7698 BLAKE             30
```

1-79

ORACLE®

### Case Conversion Functions

The example above displays the employee number, name, and department number of employee BLAKE.

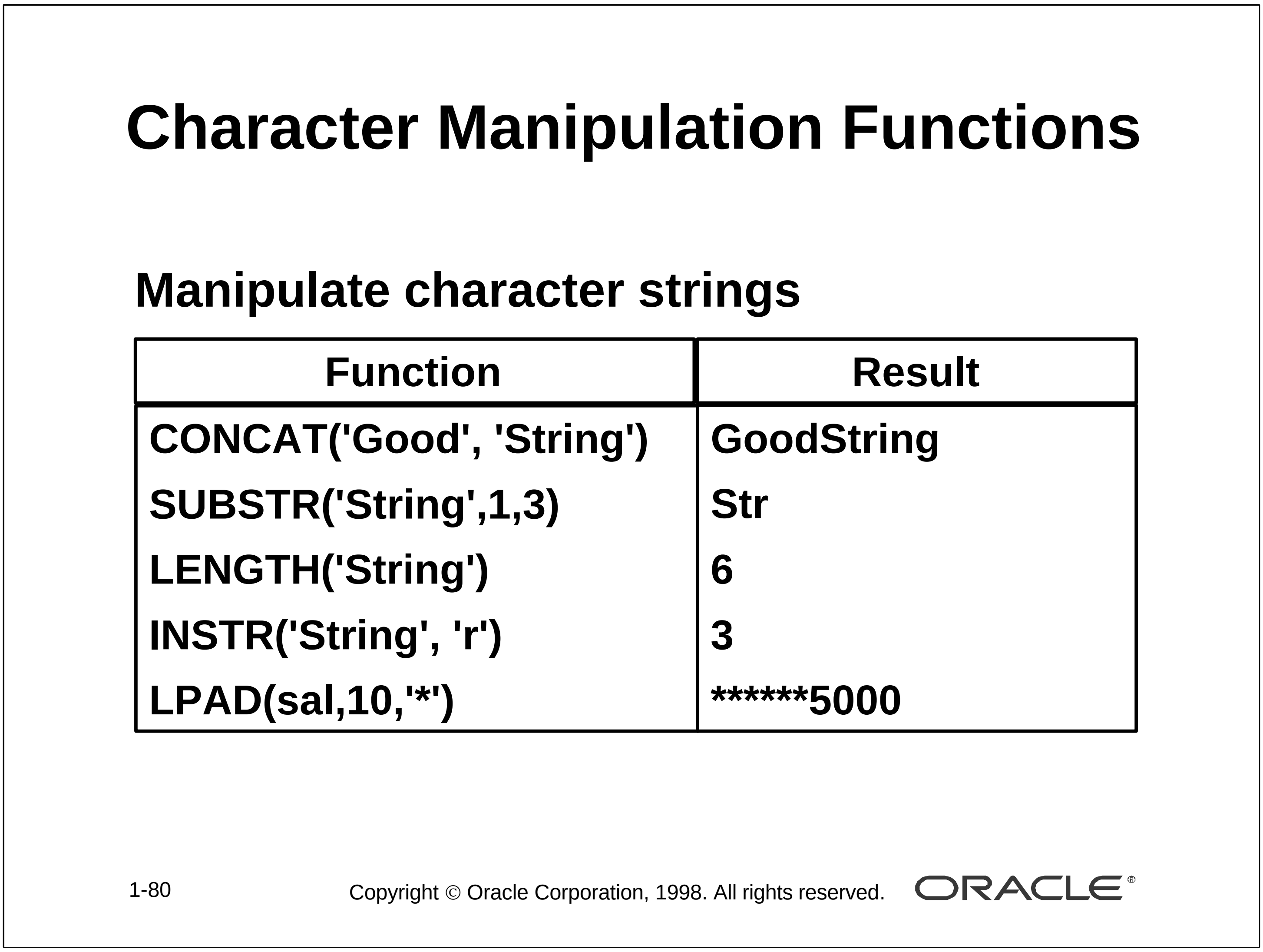
The WHERE clause of the first SQL statement specifies the employee name as ‘blake’. Since all the data in the EMP table is stored in uppercase, the name ‘blake’ does not find a match in the EMP table and as a result no rows are selected.

The WHERE clause of the second SQL statement specifies that the employee name in the EMP table be converted to lowercase and then be compared to ‘blake’. Since both the names are in lowercase now, a match is found and one row is selected. The WHERE clause can be rewritten in the following manner to produce the same result:

```
… WHERE   ename = ’BLAKE’
```

The name in the output appears as it was stored in the database. To display the name with the first letter capitalized, use the INITCAP function in the SELECT statement.

```
SQL> SELECT    empno, INITCAP(ename), deptno
  2  FROM      emp
  3  WHERE     LOWER(ename) = ’blake’;
```



# Character Manipulation Functions

## Manipulate character strings

| Function | Result |
|---|---|
| CONCAT('Good', 'String') | GoodString |
| SUBSTR('String',1,3) | Str |
| LENGTH('String') | 6 |
| INSTR('String', 'r') | 3 |
| LPAD(sal,10,'*') | ******5000 |

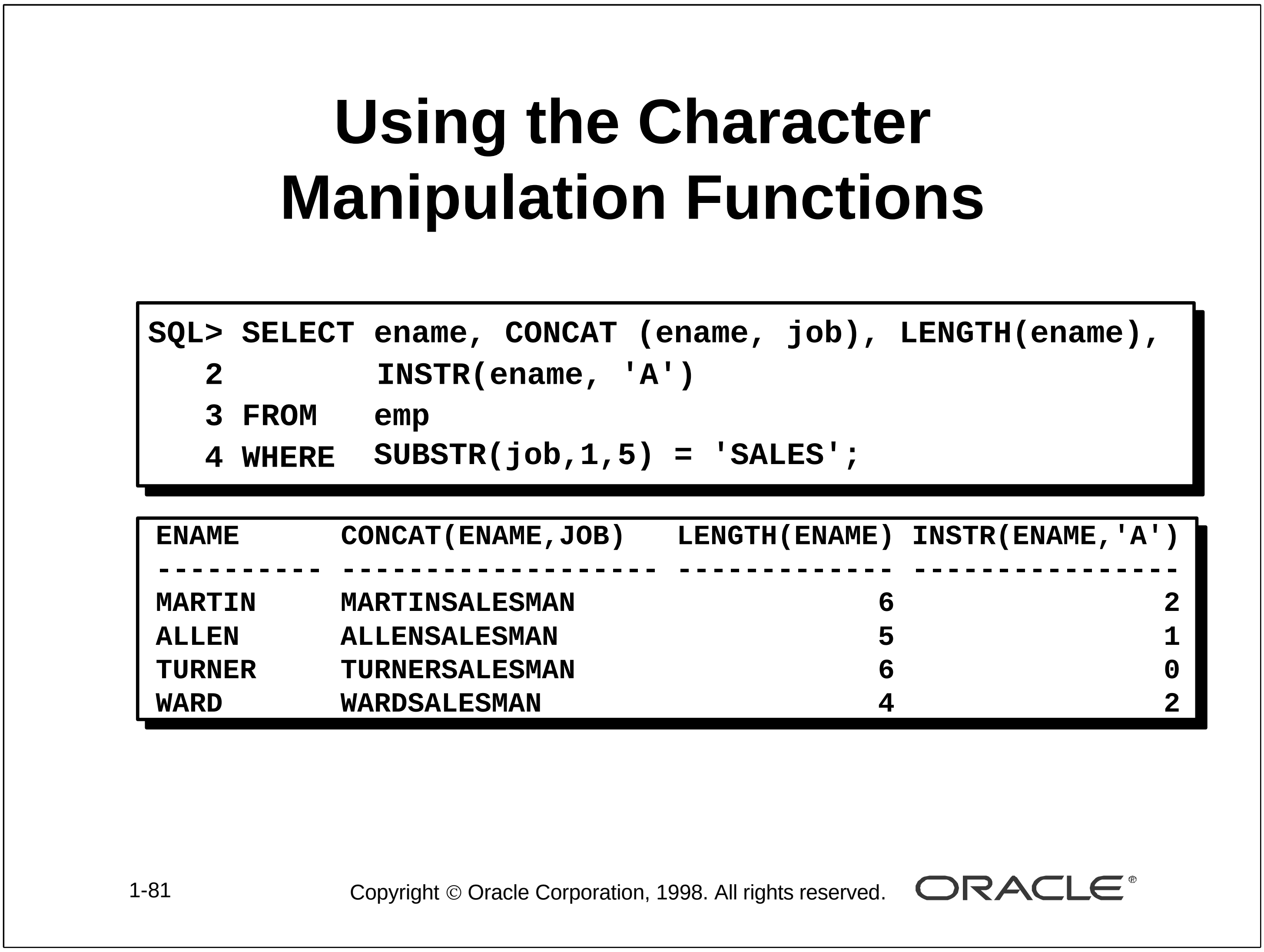
1-80

### Character Manipulation Functions

CONCAT, SUBSTR, LENGTH, INSTR, and LPAD are the five character manipulation functions covered in this lesson.

- CONCAT—Joins values together. You are limited to using two parameters with CONCAT.
- SUBSTR—Extracts a string of determined length.
- LENGTH—Shows the length of a string as a numeric value.
- INSTR—Finds numeric position of a named character.
- LPAD—Pads the character value right-justified.

**Note:** RPAD character manipulation function pads the character value left justified.



# Using the Character Manipulation Functions

```
SQL> SELECT ename, CONCAT (ename, job), LENGTH(ename),
   2        INSTR(ename, 'A')
   3 FROM   emp
   4 WHERE  SUBSTR(job,1,5) = 'SALES';
```

```
ENAME      CONCAT(ENAME,JOB)   LENGTH(ENAME) INSTR(ENAME,'A')
---------- ------------------- ------------- ----------------
MARTIN     MARTINSALESMAN                  6                2
ALLEN      ALLENSALESMAN                   5                1
TURNER     TURNERSALESMAN                  6                0
WARD       WARDSALESMAN                    4                2
```

### Character Manipulation Functions

The above example displays employee name and job joined together, length of the employee name, and the numeric position of the letter A in the employee name, for all employees who are in sales.

**Example**

Modify the above SQL statement to display the data for those employees whose names end with an N.

```
SQL> SELECT  ename, CONCAT(ename, job), LENGTH(ename),
    INSTR(ename, ’A’)
  2  FROM    emp
  3  WHERE   SUBSTR(ename, -1, 1) = ’N’;
```

```
ENAME    CONCAT(ENAME,JOB)   LENGTH(ENAME) INSTR(ENAME,’A’)
-------- ------------------- ------------- ----------------
MARTIN   MARTINSALESMAN                  6                2
ALLEN    ALLENSALESMAN                   5                1
```

# Number Functions

- ROUND: Rounds value to specified decimal

ROUND(45.926, 2) → 45.93

- TRUNC: Truncates value to specified decimal

TRUNC(45.926, 2) → 45.92

- MOD: Returns remainder of division

MOD(1600, 300) → 100

### Number Functions

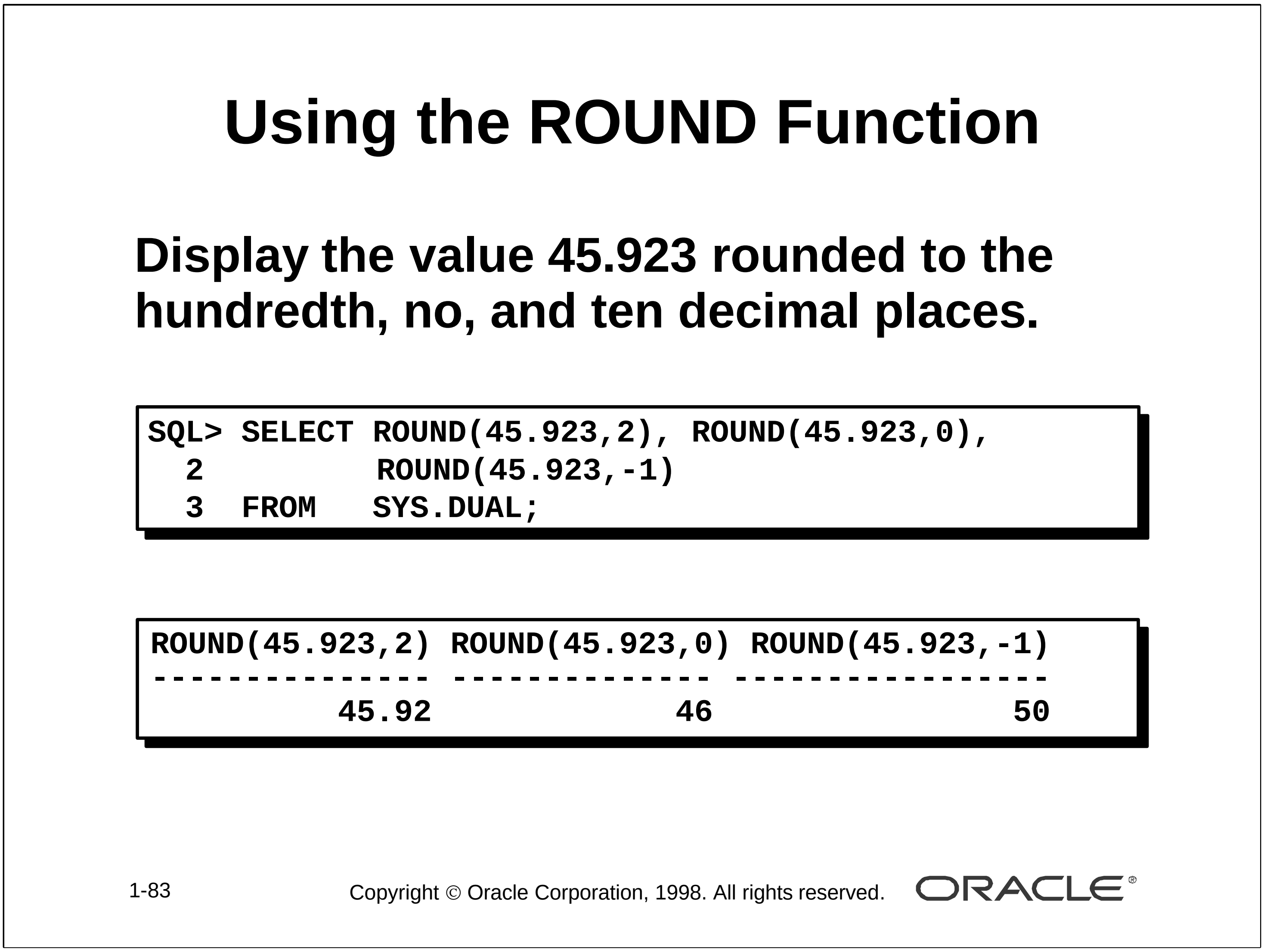
Number functions accept numeric input and return numeric values. This section describes some of the number functions.

| Function | Purpose |
|---|---|
| ROUND(*column*\|*expression, n*) | Rounds the column, expression, or value to *n* decimal places. If *n* is omitted, no decimal places. If *n* is negative, numbers to left of the decimal point are rounded. |
| TRUNC(*column*\|*expression,n*) | Truncates the column, expression, or value to *n* decimal places, or if *n* is omitted, no decimal places. If *n* is negative, numbers left of the decimal point are truncated to zero. |
| MOD(*m,n*) | Returns the remainder of *m* divided by *n*. |

**Note:** This list is a subset of the available number functions.

For more information, see

*Oracle8 Server SQL Reference, Release 8.0,* "Number Functions."



# Using the ROUND Function

## Display the value 45.923 rounded to the hundredth, no, and ten decimal places.

```
SQL> SELECT ROUND(45.923,2), ROUND(45.923,0),
  2         ROUND(45.923,-1)
  3  FROM   SYS.DUAL;
```

```
ROUND(45.923,2) ROUND(45.923,0) ROUND(45.923,-1)
--------------- --------------- ----------------
          45.92              46               50
```

1-83

### ROUND Function

The ROUND function rounds the column, expression, or value to *n* decimal places. If the second argument is 0 or is missing, the value is rounded to zero decimal places. If the second argument is 2, the value is rounded to two decimal places, or to the hundredths. Conversely, if the second argument is -2, the value is rounded to two decimal places to the left, or to the hundreds.

The ROUND function can also be used with date functions. You will see examples later in this lesson.

The SYS.DUAL is a dummy table that maintains some system information that is sometimes needed. You will see more about this later.

# Using the TRUNC Function

## Display the value 45.923 truncated to the hundredth, no, and ten decimal places.

```
SQL> SELECT TRUNC(45.923,2), TRUNC(45.923),
  2         TRUNC(45.923,-1)
  3  FROM   SYS.DUAL;
```

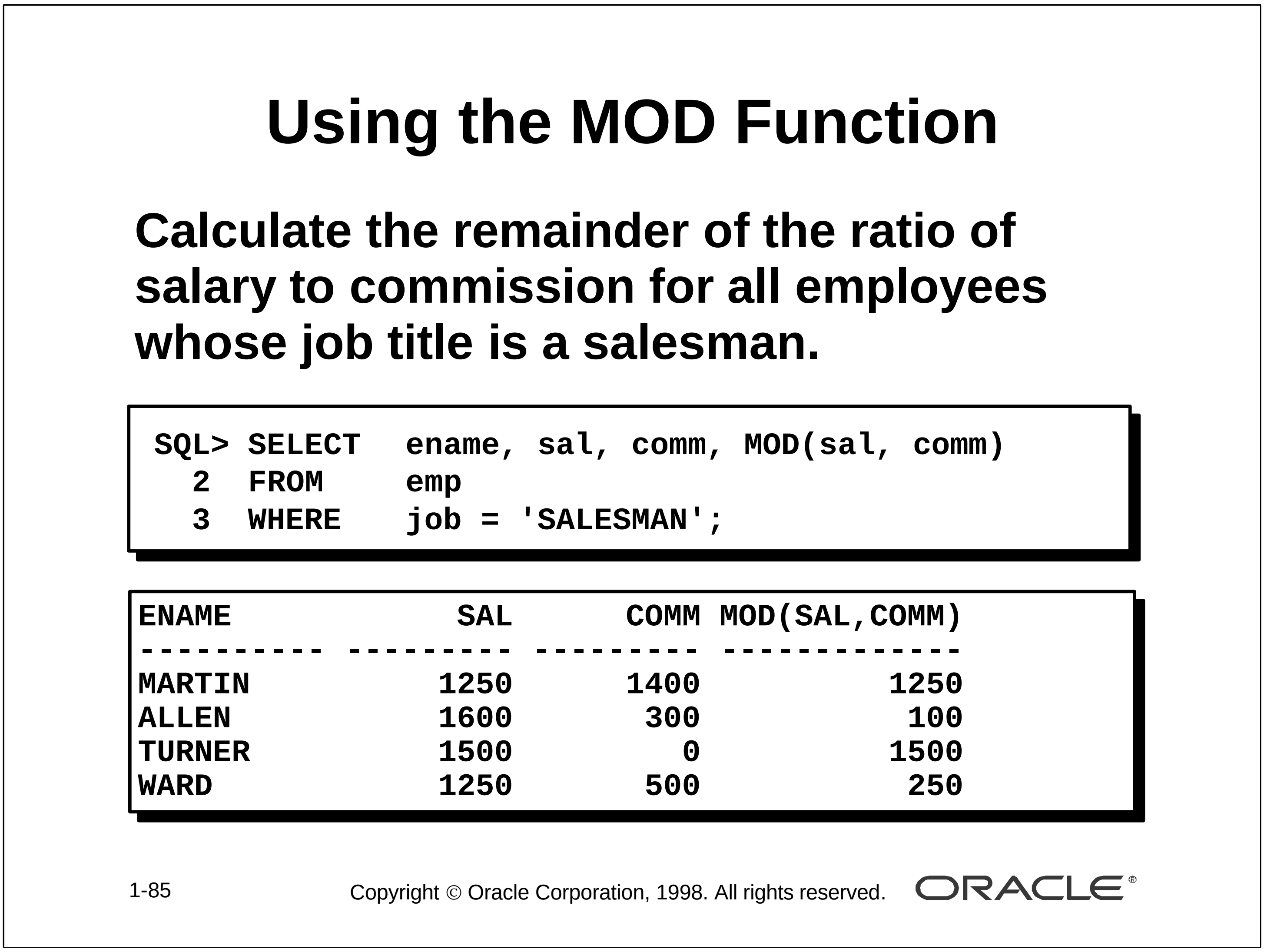
```
TRUNC(45.923,2) TRUNC(45.923) TRUNC(45.923,-1)
--------------- ------------- ---------------
          45.92            45              40
```

### TRUNC Function

The TRUNC function truncates the column, expression, or value to *n* decimal places.

The TRUNC function works with similar arguments as the ROUND function. If the second argument is 0 or is missing, the value is truncated to zero decimal places. If the second argument is 2, the value is truncated to two decimal places, or to the hundredths. Conversely, if the second argument is -2, the value is truncated to two decimal places to the left, or to the hundreds.

Like the ROUND function, the TRUNC function can also be used with date functions.



# Using the MOD Function

## Calculate the remainder of the ratio of salary to commission for all employees whose job title is a salesman.

```
SQL> SELECT   ename, sal, comm, MOD(sal, comm)
  2  FROM     emp
  3  WHERE    job = 'SALESMAN';
```

```
ENAME            SAL      COMM MOD(SAL,COMM)
---------- --------- --------- -------------
MARTIN          1250      1400          1250
ALLEN           1600       300           100
TURNER          1500         0          1500
WARD            1250       500           250
```

### MOD Function

The MOD function finds the remainder of value1 divided by value2. The example above calculates the remainder of the ratio of salary to commission for all employees whose job title is a salesman.

# Working with Dates

- Oracle stores dates in an internal numeric format: Century, year, month, day, hours, minutes, seconds.
- The default date format is DD-MON-YY.
- SYSDATE is a function returning date and time.
- DUAL is a dummy table used to view SYSDATE.

### Oracle Date Format

Oracle stores dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds.

The default display and input format for any date is DD-MON-YY. Valid Oracle dates are between January 1, 4712 B.C., and December 31, 9999 A.D.

### SYSDATE

SYSDATE is a date function that returns the current date and time. You can use SYSDATE just as you would use any other column name. For example, you can display the current date by selecting SYSDATE from a table. It is customary to select SYSDATE from a dummy table called DUAL.

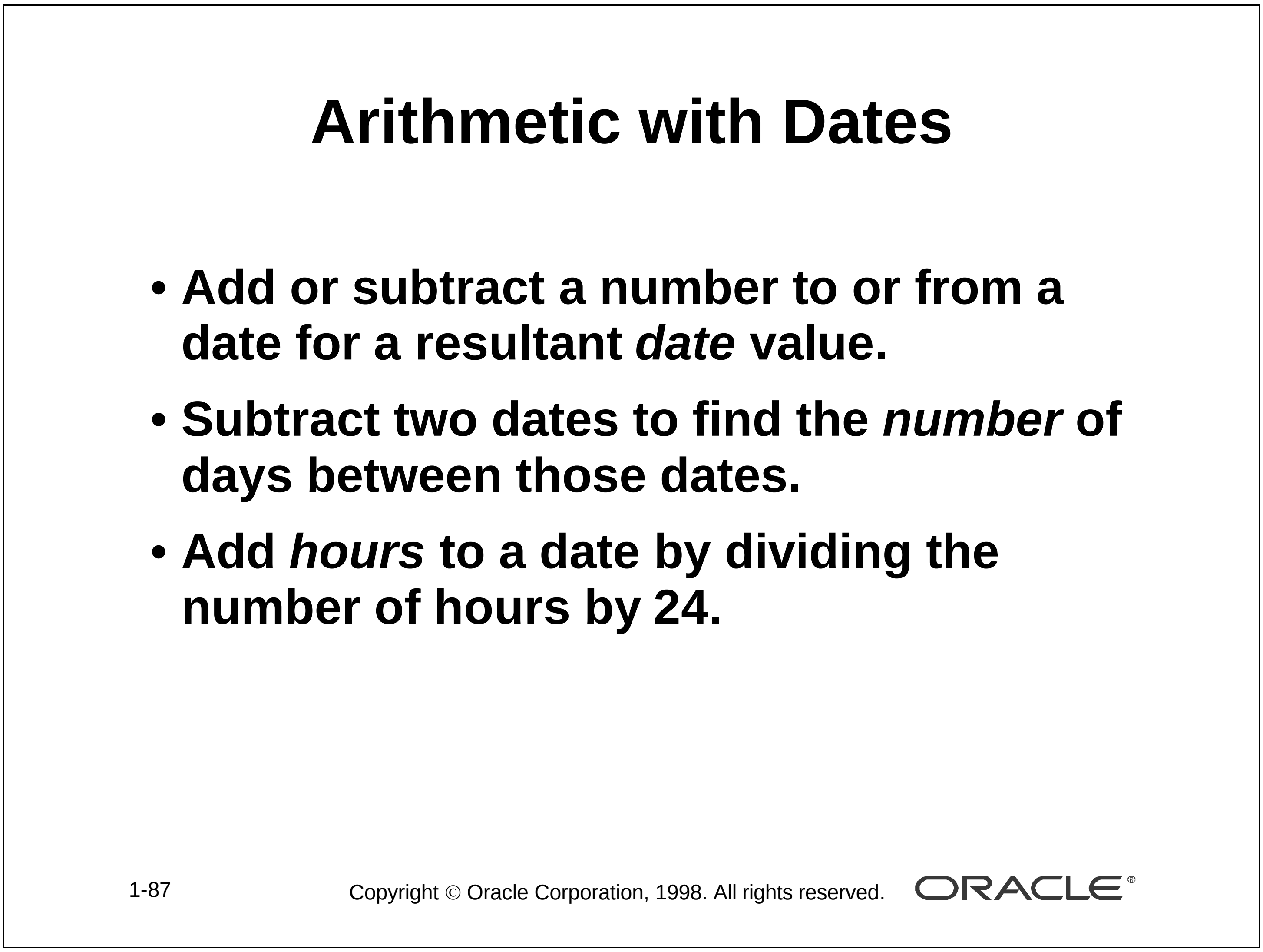
### DUAL

The DUAL table is owned by the user SYS and can be accessed by all users. It contains one column, DUMMY, and one row with the value X. The DUAL table is useful when you want to return a value once only—for instance, the value of a constant, pseudocolumn, or expression that is not derived from a table with user data.

### Example

Display the current date using the DUAL table.

```
SQL> SELECT  SYSDATE
  2  FROM    SYS.DUAL;
```



# Arithmetic with Dates

- Add or subtract a number to or from a date for a resultant *date* value.
- Subtract two dates to find the *number* of days between those dates.
- Add *hours* to a date by dividing the number of hours by 24.

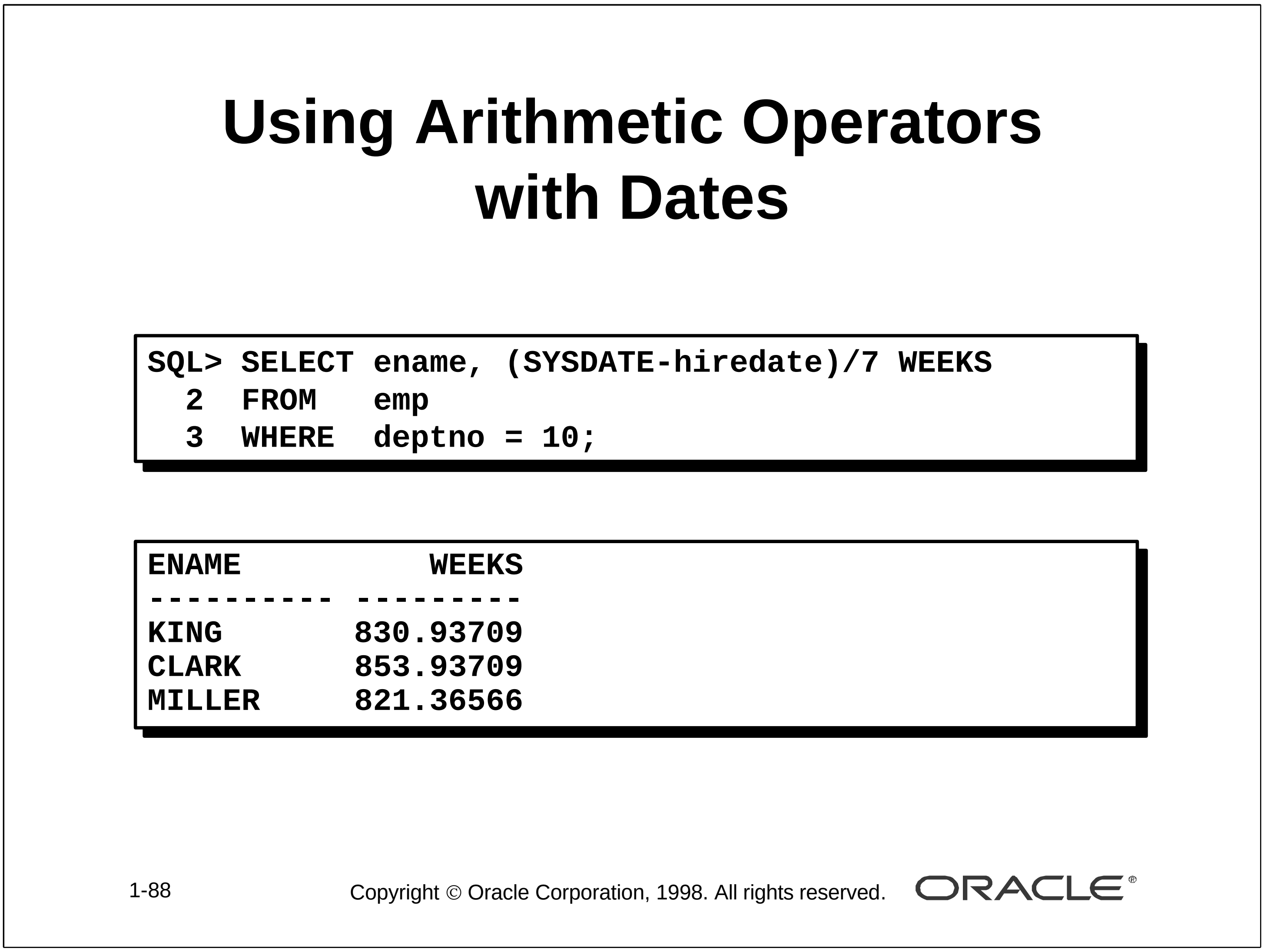
1-87

### Arithmetic with Dates

Since the database stores dates as numbers, you can perform calculations using arithmetic operators such as addition and subtraction. You can add and subtract number constants as well as dates.

You can perform the following operations:

| **Operation** | **Result** | **Description** |
|---|---|---|
| date + number | Date | Adds a  number of days to a date |
| date - number | Date | Subtracts a number of days from a date |
| date - date | Number of days | Subtracts one date from another |
| date + number/24 | Date | Adds a number of hours to a date |



# Using Arithmetic Operators with Dates

```
SQL> SELECT ename, (SYSDATE-hiredate)/7 WEEKS
  2  FROM   emp
  3  WHERE  deptno = 10;
```

```
ENAME          WEEKS
---------- ---------
KING       830.93709
CLARK      853.93709
MILLER     821.36566
```

1-88

ORACLE®

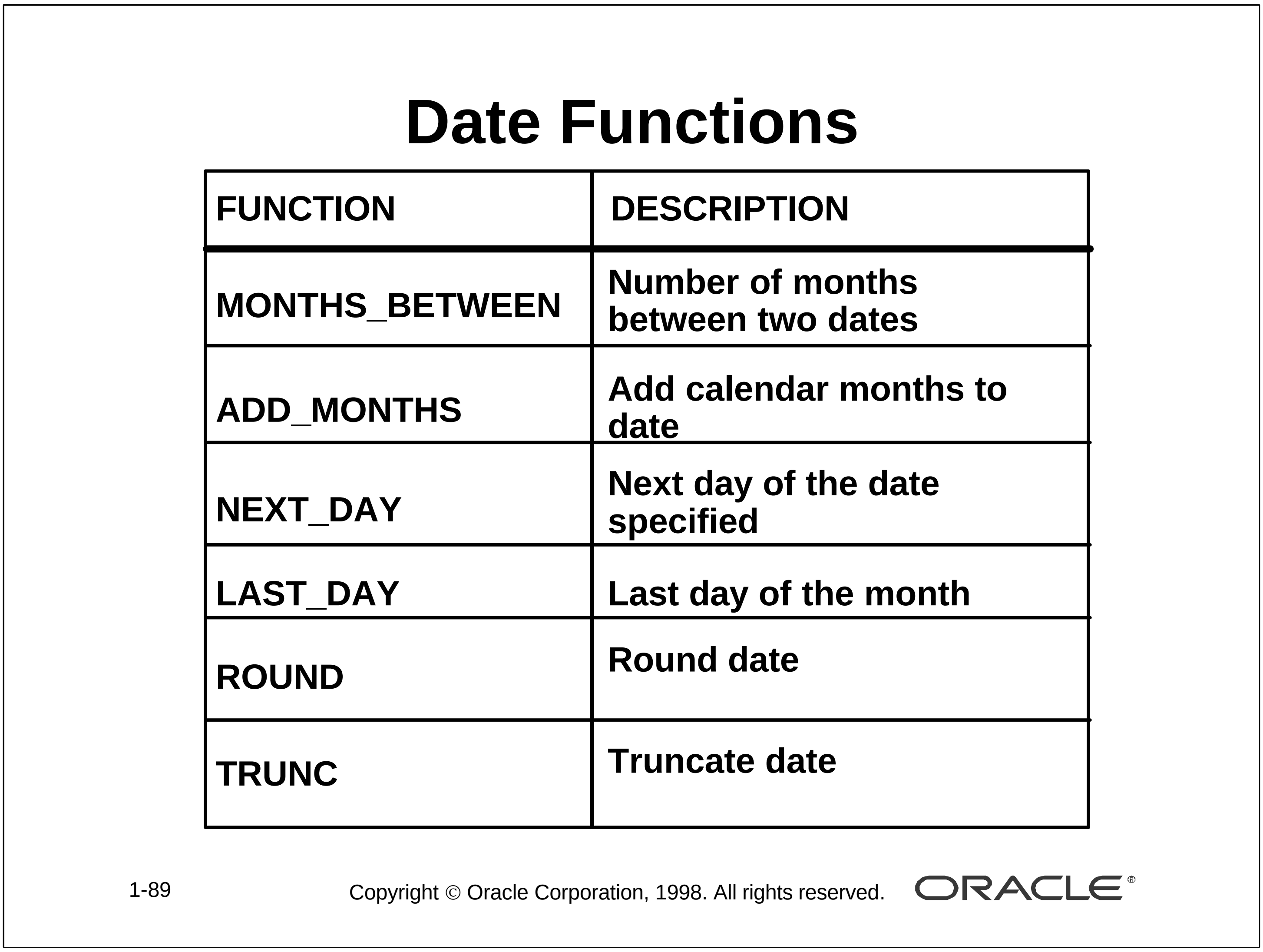
### Arithmetic with Dates (continued)

The example on the slide displays the name and the number of weeks employed for all employees in department 10. It subtracts the current date (SYSDATE) from the date on which the employee was hired and divides the result by 7 to calculate the number of weeks that a worker has been employed.

**Note:** SYSDATE is a SQL function that returns the current date and time. Your results may differ from the example.

### Class Management Note

If an older date is subtracted from a more current date, the difference is a negative number.



# Date Functions

| FUNCTION | DESCRIPTION |
|---|---|
| MONTHS_BETWEEN | Number of months between two dates |
| ADD_MONTHS | Add calendar months to date |
| NEXT_DAY | Next day of the date specified |
| LAST_DAY | Last day of the month |
| ROUND | Round date |
| TRUNC | Truncate date |

1-89

## Date Functions

Date functions operate on Oracle dates. All date functions return a value of DATE datatype except MONTHS_BETWEEN, which returns a numeric value.

- MONTHS_BETWEEN(*date1, date2*)—Finds the number of months between *date1* and *date2*. The result can be positive or negative. If *date1* is later than *date2*, the result is positive; if *date1* is earlier than *date2*, the result is negative. The noninteger part of the result represents a portion of the month.
- ADD_MONTHS(*date, n*)—Adds *n* number of calendar months to *date*. *n* must be an integer and can be negative.
- NEXT_DAY(*date,* '*char*')—Finds the date of the next specified day of the week ('*char*') following *date*. *char* may be a number representing a day or a character string.
- LAST_DAY(*date*)—Finds the date of the last day of the month that contains *date*.
- ROUND(*date*[,'*fmt*'])—Returns *date* rounded to the unit specified by the format model *fmt*. If the format model *fmt* is omitted, *date* is rounded to the nearest date.
- TRUNC(*date*[, '*fmt*'])—Returns *date* with the time portion of the day truncated to the unit specified by the format model *fmt*. If the format model *fmt* is omitted, *date* is truncated to the nearest day.

This list is a subset of the available date functions. The format models are covered later in this chapter. Examples of format models are month or year.

# Using Date Functions

- MONTHS_BETWEEN ('01-SEP-95','11-JAN-94') → 19.6774194

- ADD_MONTHS ('11-JAN-94',6) → '11-JUL-94'

- NEXT_DAY ('01-SEP-95','FRIDAY') → '08-SEP-95'

- LAST_DAY('01-SEP-95') → '30-SEP-95'

**Date Functions (continued)**

For all employees employed for fewer than 200 months, display the employee number, hire date, number of months employed, six month review date, first friday after hire date, and the last day of the month when hired.

```
SQL> SELECT   empno, hiredate,
  2           MONTHS_BETWEEN(SYSDATE, hiredate) TENURE,
  3           ADD_MONTHS(hiredate, 6) REVIEW,
  4           NEXT_DAY(hiredate, ′FRIDAY′), LAST_DAY(hiredate)
  5  FROM     emp
  6  WHERE    MONTHS_BETWEEN (SYSDATE, hiredate)<200;
```

```
    EMPNO HIREDATE     TENURE REVIEW    NEXT_DAY( LAST_DAY(
--------- --------- --------- --------- --------- ---------
     7839 17-NOV-81 192.24794 17-MAY-82 20-NOV-81 30-NOV-81
     7698 01-MAY-81 198.76407 01-NOV-81 08-MAY-81 31-MAY-81
...
11 rows selected.
```

# Using Date Functions

- ROUND('25-JUL-95','MONTH') → 01-AUG-95
- ROUND('25-JUL-95','YEAR') → 01-JAN-96
- TRUNC('25-JUL-95','MONTH') → 01-JUL-95
- TRUNC('25-JUL-95','YEAR') → 01-JAN-95

### Date Functions (continued)

The ROUND and TRUNC functions can be used for number and date values. When using these functions with dates, they round or truncate to the specified format model. Therefore, you can round dates to the nearest year or month.

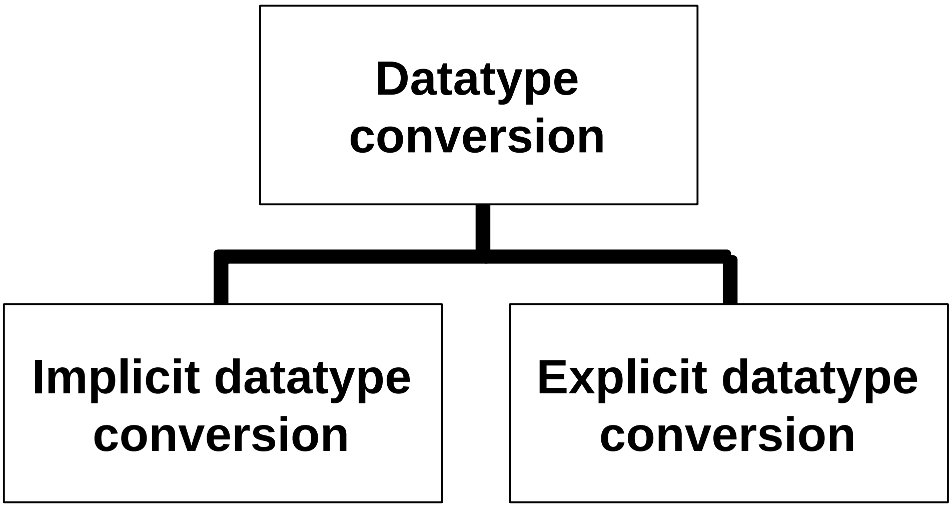
### Example

Compare the hire dates for all employees who started in 1987. Display the employee number, hire date, and month started using the ROUND and TRUNC functions.

```
SQL> SELECT empno, hiredate,
  2         ROUND(hiredate, ’MONTH’), TRUNC(hiredate, ’MONTH’)
  3  FROM   emp
  4  WHERE  hiredate like ’%87’;
```

```
    EMPNO HIREDATE  ROUND(HIR TRUNC(HIR
--------- --------- --------- ---------
     7788 19-APR-87 01-MAY-87 01-APR-87
     7876 23-MAY-87 01-JUN-87 01-MAY-87
```

# Conversion Functions


Datatype conversion

Implicit datatype conversion

Explicit datatype conversion

### Conversion Functions

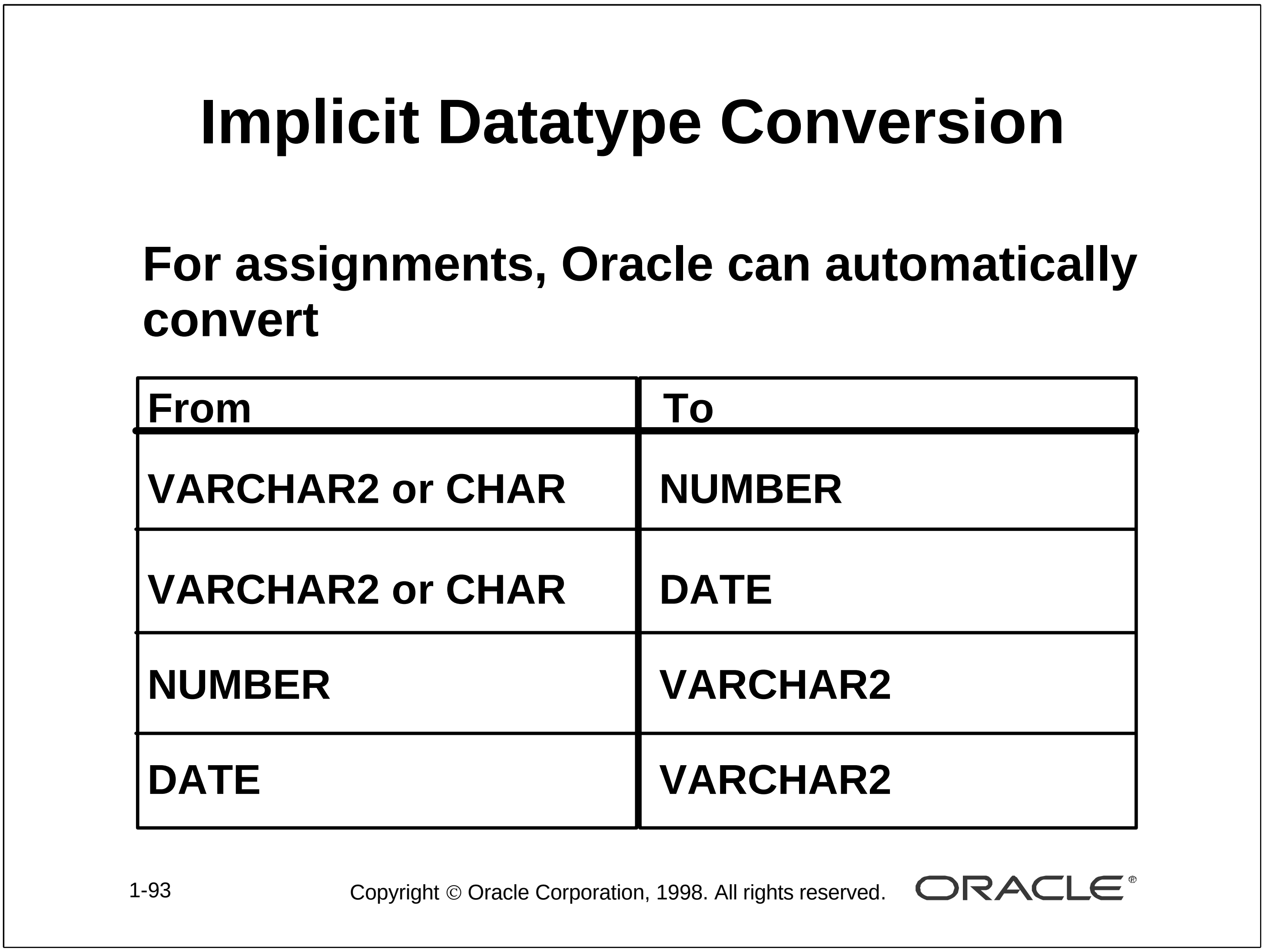
In addition to Oracle datatypes, columns of tables in an Oracle8 database can be defined using ANSI, DB2, and SQL/DS datatypes. However, Oracle8 Server internally converts such datatypes to Oracle8 datatypes.

In some cases, Oracle8 Server allows data of one datatype where it expects data of a different datatype. This is allowed when Oracle8 Server can automatically convert the data to the expected datatype. This datatype conversion can be done *implicitly* by Oracle8 Server or *explicitly* by the user.

Implicit datatype conversions work according to the rules explained in next two slides.

Explicit datatype conversions are done by using the conversion functions. Conversion functions convert a value from one datatype to another. Generally, the form of the function names follow the convention *datatype* TO *datatype*. The first datatype is the input datatype; the last datatype is the output.

**Note:** Though implicit datatype conversion is available, it is recommended that you do explicit datatype conversion to ensure reliability of your SQL statements.



# Implicit Datatype Conversion

## For assignments, Oracle can automatically convert

| From | To |
|---|---|
| VARCHAR2 or CHAR | NUMBER |
| VARCHAR2 or CHAR | DATE |
| NUMBER | VARCHAR2 |
| DATE | VARCHAR2 |

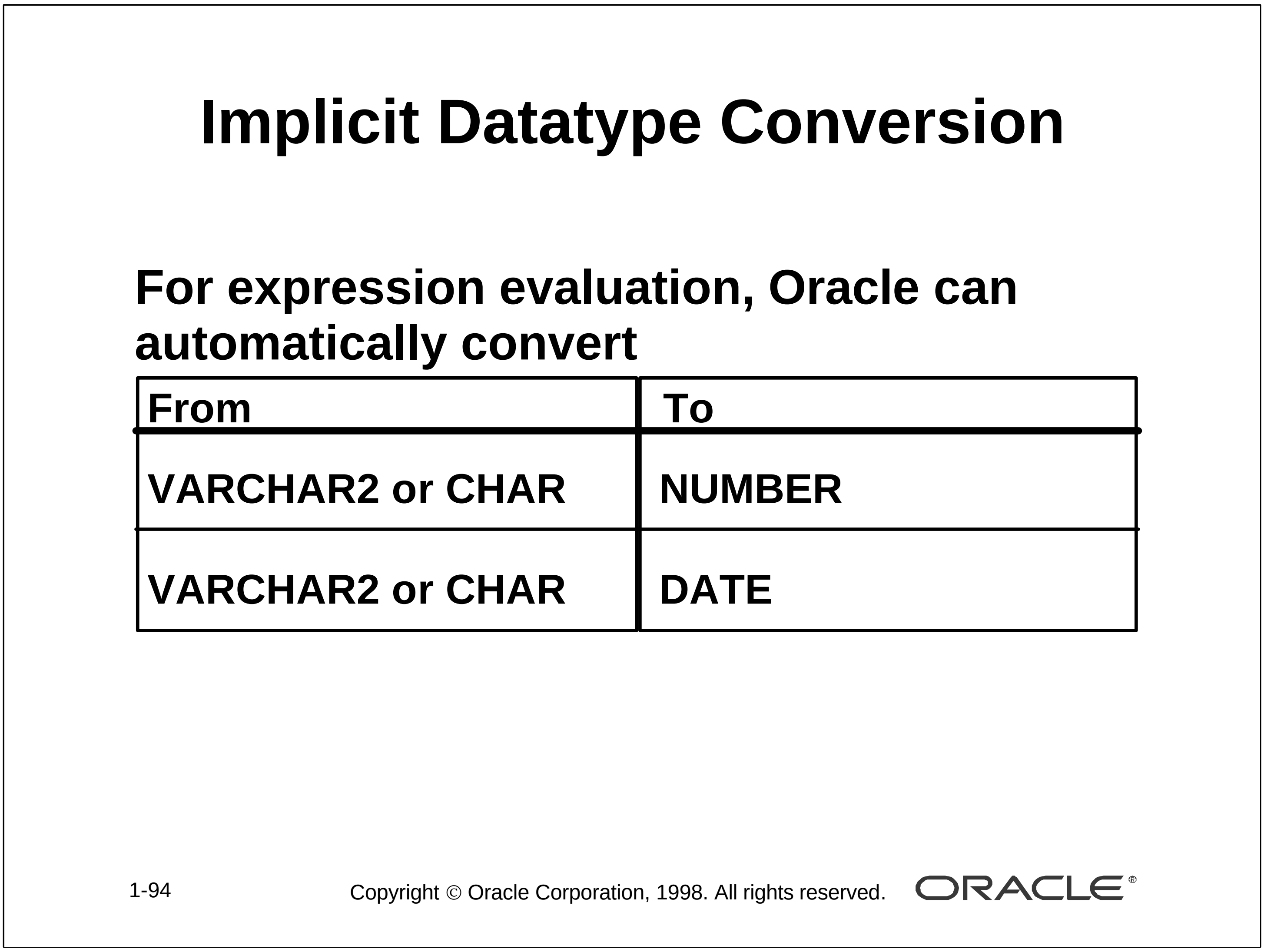
1-93

### Implicit Datatype Conversion

For assignments, Oracle8 Server can automatically convert the following:

- VARCHAR2 or CHAR to NUMBER
- VARCHAR2 or CHAR to DATE
- NUMBER to VARCHAR2
- DATE to VARCHAR2

The assignment succeeds if Oracle8 Server can convert the datatype of the value used in the assignment to that of the assignment's target.



# Implicit Datatype Conversion

## For expression evaluation, Oracle can automatically convert

| From | To |
|---|---|
| VARCHAR2 or CHAR | NUMBER |
| VARCHAR2 or CHAR | DATE |

1-94

ORACLE®

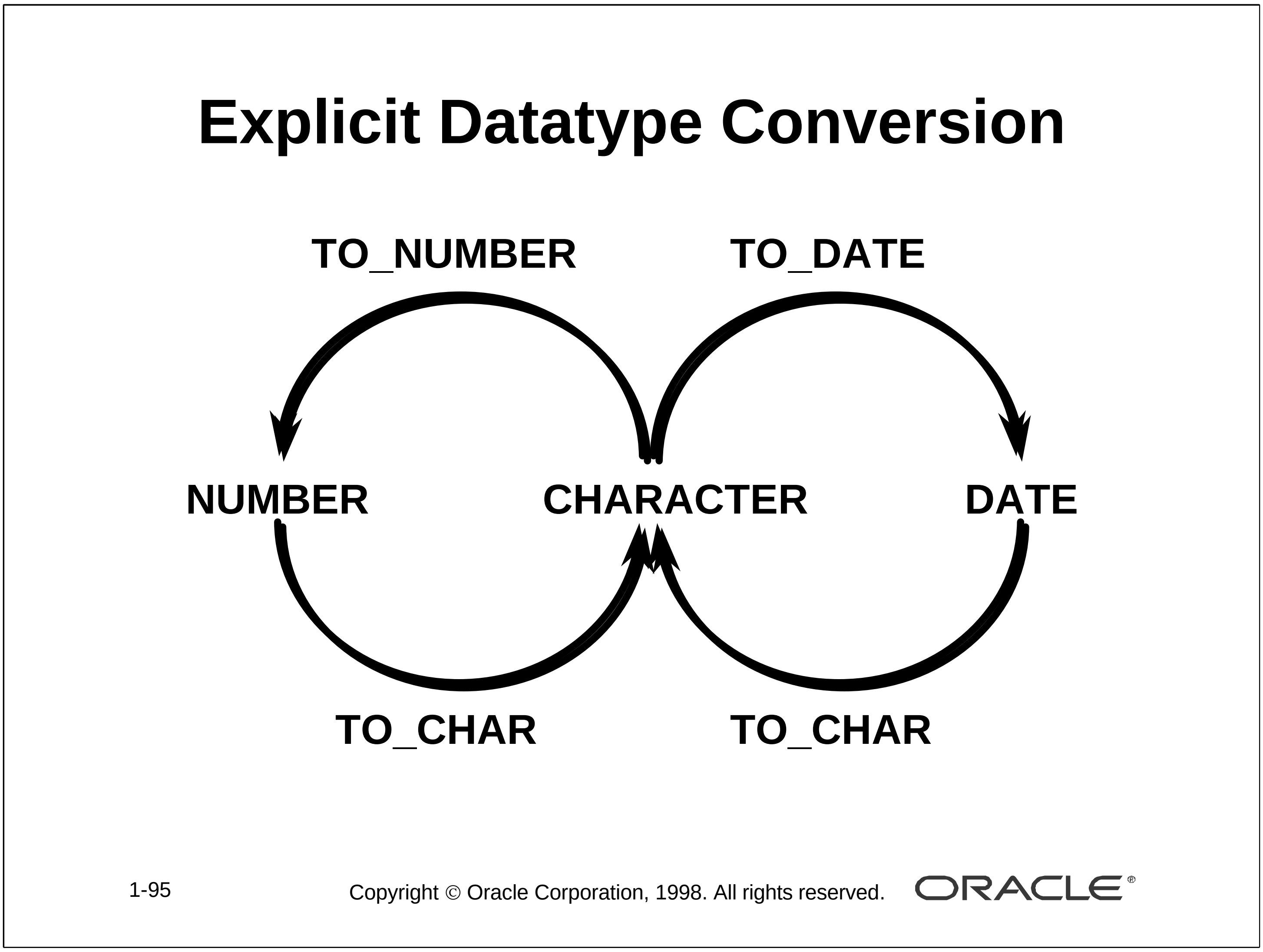
### Implicit Datatype Conversion

For expression evaluation, Oracle8 Server can automatically convert the following:

- VARCHAR2 or CHAR to NUMBER
- VARCHAR2 or CHAR to DATE

In general, Oracle8 Server uses the rule for expression when a datatype conversion is needed in places not covered by a rule for assignment conversions.

**Note:** CHAR to NUMBER conversions succeed only if the character string represents a valid number. CHAR to DATE conversions succeed only if the character string has the default format DD-MON-YY.



# Explicit Datatype Conversion

1-95

### Explicit Datatype Conversion

SQL provides three functions to convert a value from one datatype to another.

| Function | Purpose |
|---|---|
| TO_CHAR(*number*\|*date*,[‘*fm*t’]) | Converts a number or date value to a VARCHAR2 character string with format model *fmt*. |
| TO_NUMBER(*char*) | Converts a character string containing digits to a number. |
| TO_DATE(C*har*,[‘*fmt*’] | Converts a character string representing a date to a date value according to the *fmt* specified. If *fmt* is omitted, format is DD-MON-YY. |

**Note:** This list is a subset of the available conversion functions.

For more information, see *Oracle8 Server SQL Reference, Release 8.0*, “Conversion Functions.”

### Class Management Note

An additional conversion function is CHR(*number*) that returns the character having the binary equivalent of *number* as a VARCHAR2 value in the database character set.

# TO_CHAR Function with Dates

```
TO_CHAR(date, 'fmt')
```

## The format model:

- **Must be enclosed in single quotation marks and is case sensitive**
- **Can include any valid date format element**
- **Has an *fm* element to remove padded blanks or suppress leading zeros**
- **Is separated from the date value by a comma**

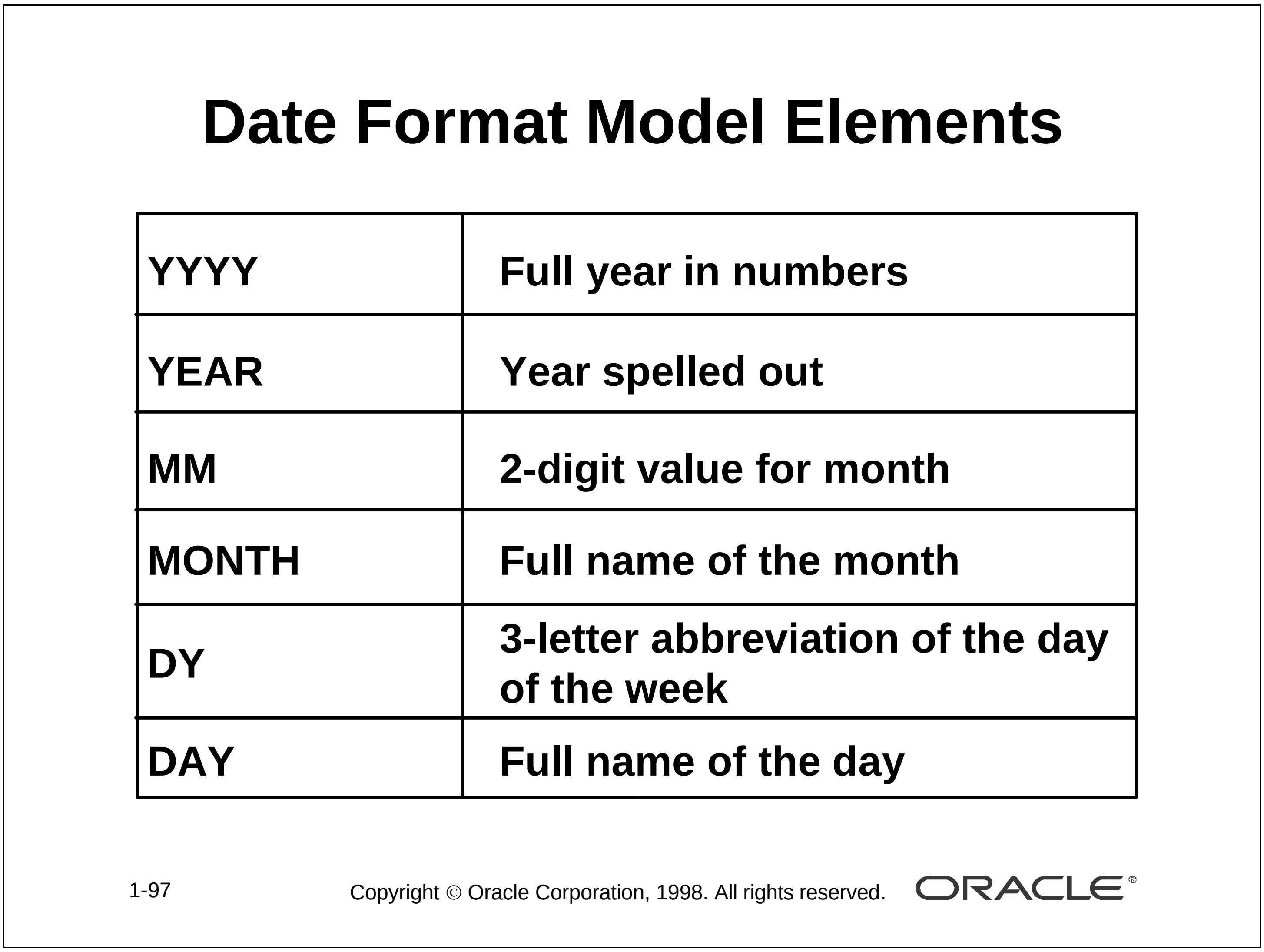
### Displaying a Date in a Specific Format

Previously, all Oracle date values were displayed in the DD-MON-YY format. The TO_CHAR function allows you to convert a date from this default format to one specified by you.

### Guidelines

- The format model must be enclosed in single quotation marks and is case sensitive.
- The format model can include any valid date format element. Be sure to separate the date value from the format model by a comma.
- The names of days and months in the output are automatically padded with blanks.
- To remove padded blanks or to suppress leading zeros, use the fill mode *fm* element.
- You can resize the display width of the resulting character field with the SQL*Plus COLUMN command.
- The resultant column width is 80 characters by default.

```
SQL> SELECT  empno, TO_CHAR(hiredate, ’MM/YY’) Month_Hired
  2  FROM    emp
  3  WHERE   ename = ’BLAKE’;
```



# Date Format Model Elements

| | |
|---|---|
| YYYY | Full year in numbers |
| YEAR | Year spelled out |
| MM | 2-digit value for month |
| MONTH | Full name of the month |
| DY | 3-letter abbreviation of the day of the week |
| DAY | Full name of the day |

**Sample Valid Date Format Elements**

| Element | Description |
|---|---|
| SCC or CC | Century; S prefixes BC date with - |
| Years in dates YYYY or SYYYY | Year; S prefixes BC date with - |
| YYY or YY or Y | Last 3, 2, or 1 digit(s) of year |
| Y,YYY | Year with comma in this position |
| IYYY, IYY, IY, I | 4, 3, 2, or 1 digit year based on the ISO standard |
| SYEAR or YEAR | Year spelled out; S prefixes BC date with - |
| BC or AD | BC/AD indicator |
| B.C. or A.D. | BC/AD indicator with periods |
| Q | Quarter of year |
| MM | Month, two-digit value |
| MONTH | Name of month padded with blanks to length of 9 characters |
| MON | Name of month, three-letter abbreviation |
| RM | Roman numeral month |
| WW or W | Week of year or month |
| DDD or DD or D | Day of year, month, or week |
| DAY | Name of day padded with blanks to length of 9 characters. |
| DY | Name of day; 3-letter abbreviation. |
| J | Julian day; the number of days since 31 December 4713 BC. |

# Date Format Model Elements

- **Time elements format the time portion of the date.**

| | |
|---|---|
| **HH24:MI:SS AM** | **15:45:32 PM** |

- **Add character strings by enclosing them in double quotation marks.**

| | |
|---|---|
| **DD "of" MONTH** | **12 of OCTOBER** |

- **Number suffixes spell out numbers.**

| | |
|---|---|
| **ddspth** | **fourteenth** |

### Time Formats

Use the formats listed in the following tables to display time information and literals and to change numerals to spelled numbers.

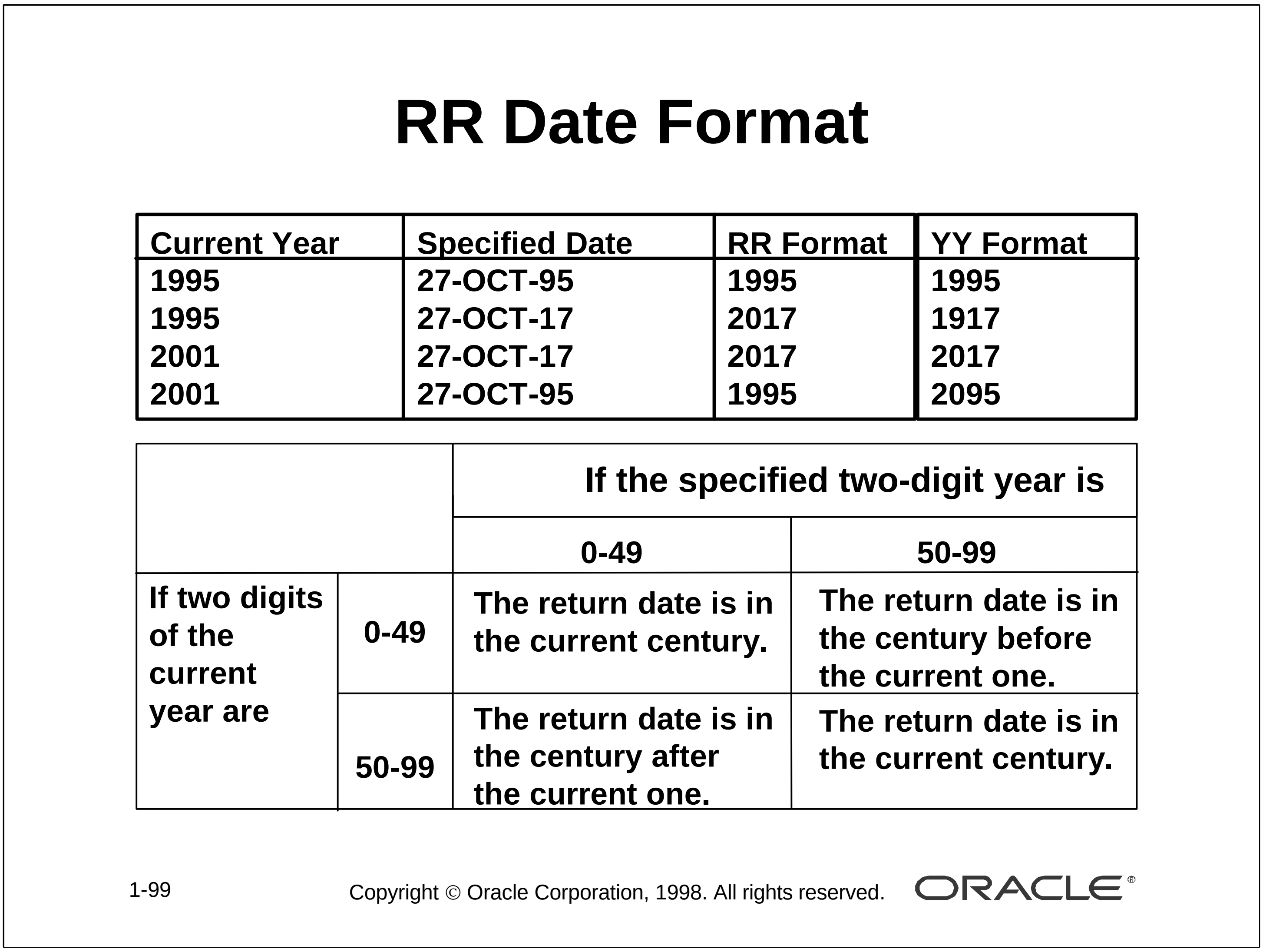
| Element | Description |
|---|---|
| AM or PM | Meridian indicator |
| A.M. or P.M. | Meridian indicator with periods |
| HH or HH12 or HH24 | Hour of day or hour (1-12) or hour (0-23) |
| MI | Minute (0-59) |
| SS | Second (0-59). |
| SSSSS | Seconds past midnight (0-86399). |

### Other Formats

| Element | Description |
|---|---|
| / . , | Punctuation is reproduced in the result |
| “ of the ” | Quoted string is reproduced in the result |

### Specifying Suffixes to Influence Number Display

| Element | Description |
|---|---|
| TH | Ordinal number (for example, DDTH for 4TH) |
| SP | Spelled-out number (for example, DDSP for FOUR) |
| SPTH or THSP | Spelled-out ordinal numbers (for example, DDSPTH for FOURTH) |



# RR Date Format

| Current Year | Specified Date | RR Format | YY Format |
|---|---|---|---|
| 1995 | 27-OCT-95 | 1995 | 1995 |
| 1995 | 27-OCT-17 | 2017 | 1917 |
| 2001 | 27-OCT-17 | 2017 | 2017 |
| 2001 | 27-OCT-95 | 1995 | 2095 |

| | | If the specified two-digit year is | |
|---|---|---|---|
| | | 0-49 | 50-99 |
| If two digits of the current year are | 0-49 | The return date is in the current century. | The return date is in the century before the current one. |
| | 50-99 | The return date is in the century after the current one. | The return date is in the current century. |

ORACLE®

### The RR Date Format Element

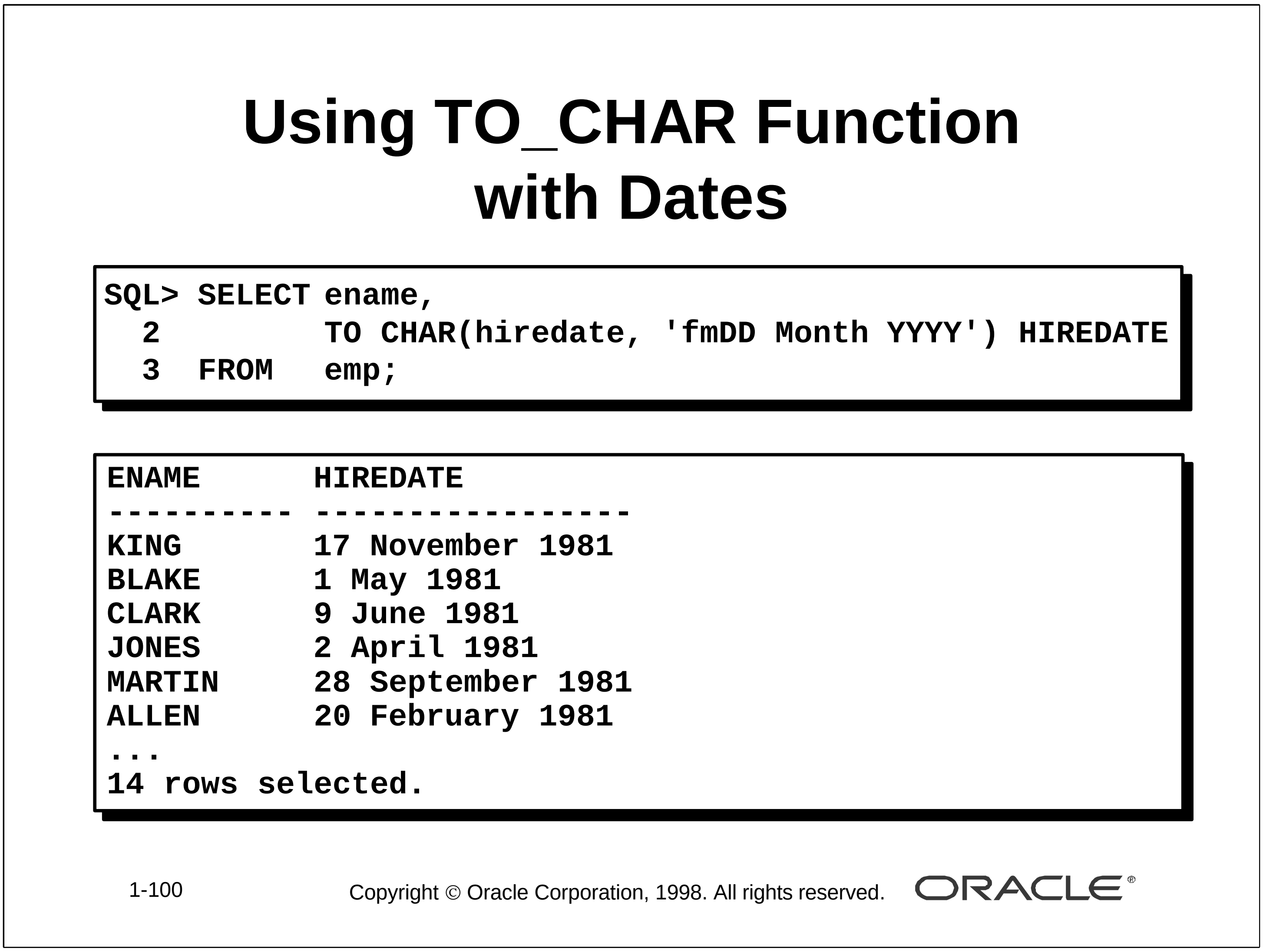
The RR date format is similar to the YY element, but it allows you to specify different centuries. You can use the RR date format element instead of YY, so that the century of the return value varies according to the specified two-digit year and the last two digits of the current year. The table on the slide summarizes the behavior of the RR element.

| Current Year | Given Date | Interpreted (RR) | Interpreted (YY) |
|---|---|---|---|
| 1994 | 27-OCT-95 | 1995 | 1995 |
| 1994 | 27-OCT-17 | 2017 | 1917 |
| 2001 | 27-OCT-17 | 2017 | 2017 |

### Class Management Note

RR is available in Oracle7, not Oracle Version 6. NLS parameters can be added to the `init.ora` file to set default date formats and language names and abbreviations. For more information, see *Oracle8 Server SQL Reference, Release 8.0,* “Alter Session” clause.

Demo: (for Date Format Model Elements) *l3hire.sql*



# Using TO_CHAR Function with Dates

```
SQL> SELECT ename,
  2         TO CHAR(hiredate, 'fmDD Month YYYY') HIREDATE
  3  FROM   emp;
```

```
ENAME      HIREDATE
---------- -----------------
KING       17 November 1981
BLAKE      1 May 1981
CLARK      9 June 1981
JONES      2 April 1981
MARTIN     28 September 1981
ALLEN      20 February 1981
...
14 rows selected.
```

### TO_CHAR Function with Dates

The above SQL statement displays the name and hire dates for all the employees. The hire date looks like 17 November 1981.

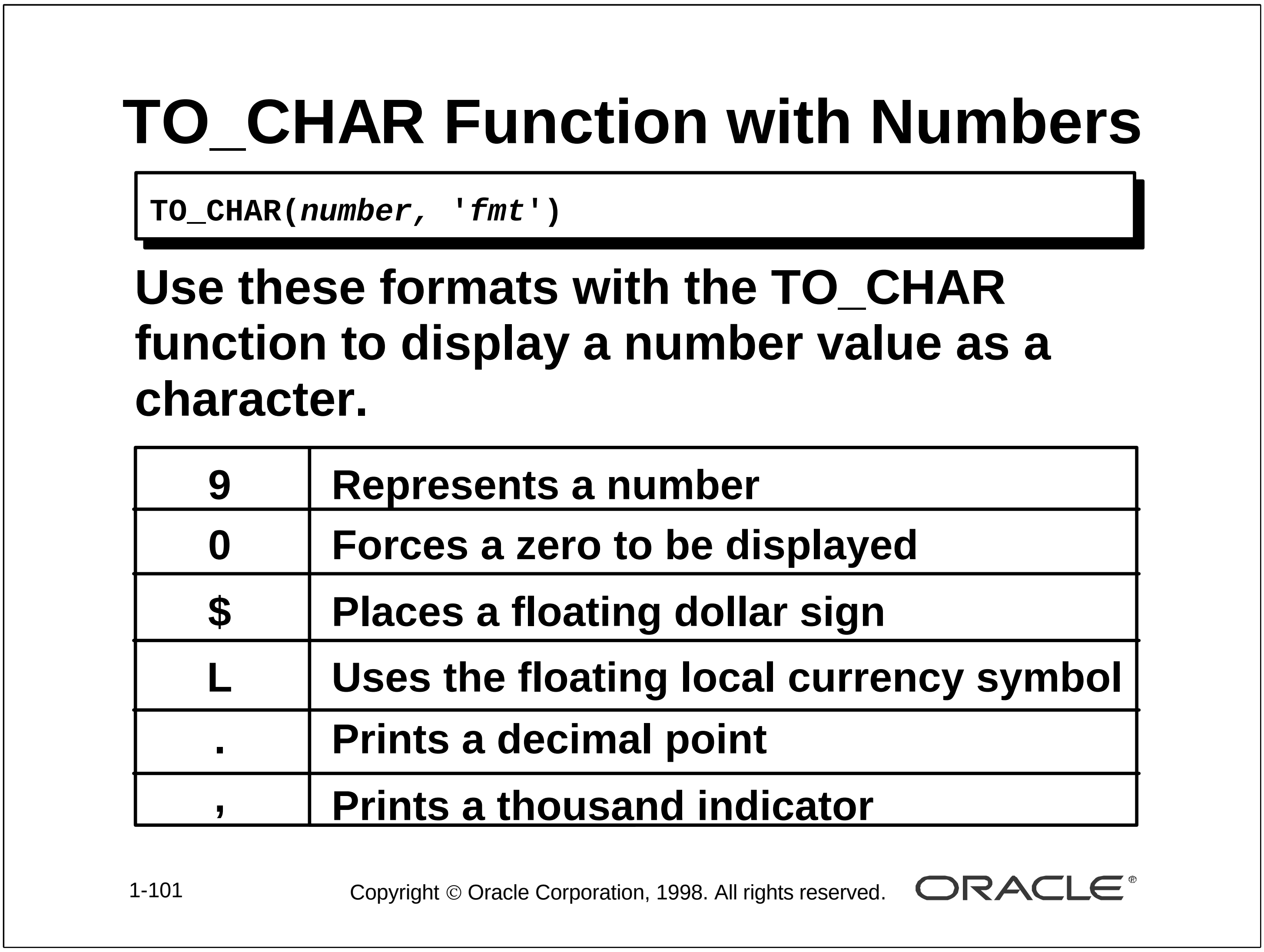
### Example

Modify the above example to display the dates in a format that looks like Seventh of February 1981 08:00:00 AM.

```
SQL> SELECT  ename,
  2          TO_CHAR(hiredate, ’fmDdspth "of" Month YYYY fmHH:MI:SSAM’)
  3          HIREDATE
  4  FROM    emp;
```

```
ENAME      HIREDATE
---------- ------------------------------------------------
KING       Seventeenth of November 1981 12:00:00 AM
BLAKE      First of May 1981 12:00:00 AM
...
14 rows selected.
```

Notice that the month follows the format model specified (INITCAP).



# TO_CHAR Function with Numbers

```
TO_CHAR(number, 'fmt')
```

**Use these formats with the TO_CHAR function to display a number value as a character.**

| | |
|---|---|
| 9 | Represents a number |
| 0 | Forces a zero to be displayed |
| $ | Places a floating dollar sign |
| L | Uses the floating local currency symbol |
| . | Prints a decimal point |
| , | Prints a thousand indicator |

1-101

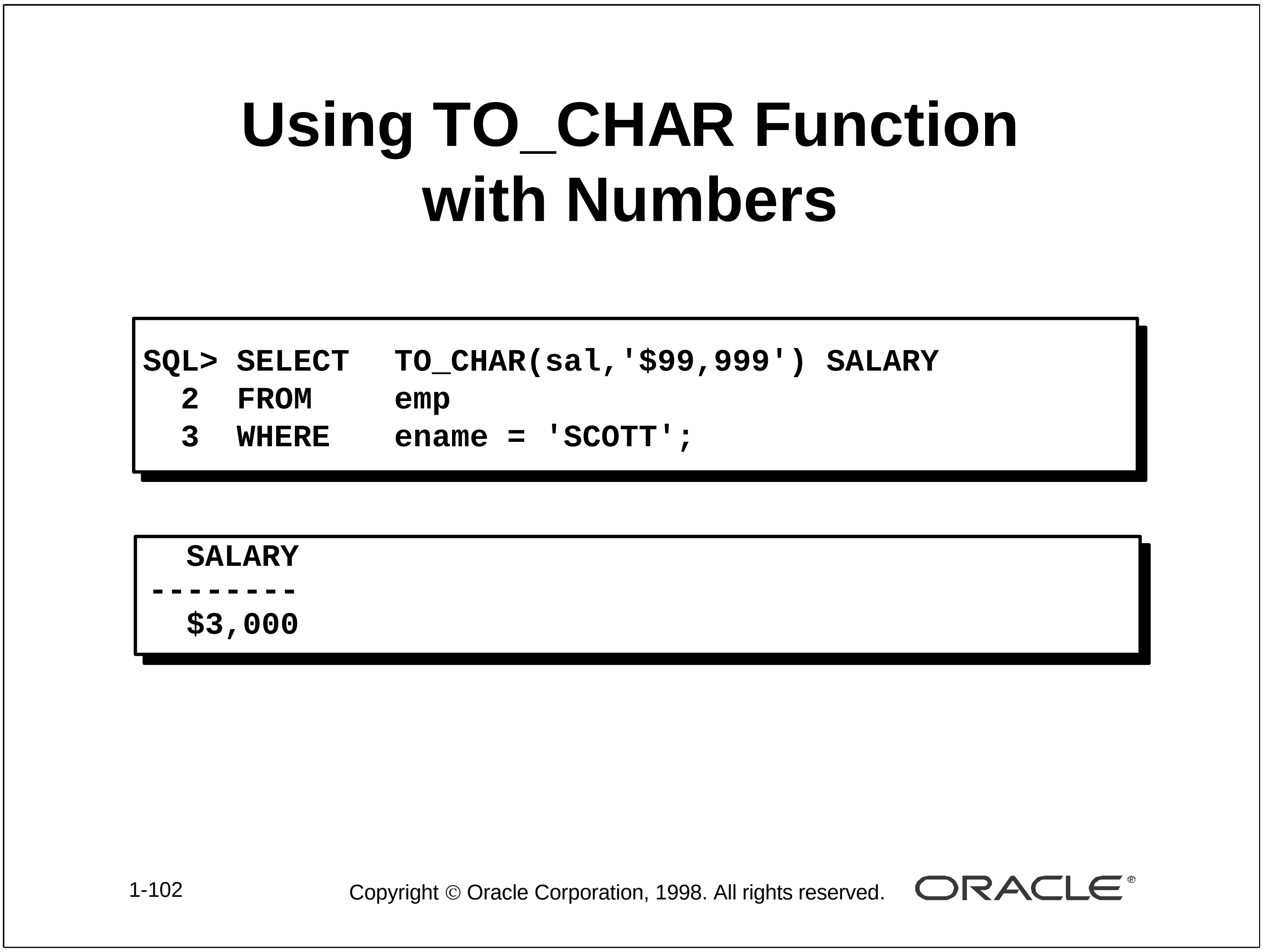
### TO_CHAR Function with Numbers

When working with number values such as character strings, you should convert those numbers to the character datatype using the TO_CHAR function, which translates a value of NUMBER datatype to VARCHAR2 datatype. This technique is especially useful with concatenation.

### Number Format Elements

If you are converting a number to character datatype, you can use the elements listed below.

| Element | Description | Example | Result |
|---|---|---|---|
| 9 | Numeric position (number of 9s determine display width) | 999999 | 1234 |
| 0 | Display leading zeros | 099999 | 001234 |
| $ | Floating dollar sign | $999999 | $1234 |
| L | Floating local currency symbol | L999999 | FF1234 |
| . | Decimal point in position specified | 999999.99 | 1234.00 |
| , | Comma in position specified | 999,999 | 1,234 |
| MI | Minus signs to right (negative values) | 999999MI | 1234- |
| PR | Parenthesize negative numbers | 999999PR | <1234> |
| EEEE | Scientific notation (format must specify four Es) | 99.999EEEE | 1.234E+03 |
| V | Multiply by 10 *n* times (*n* = no. of 9s after V) | 9999V99 | 123400 |
| B | Display zero values as blank, not 0 | B9999.99 | 1234.00 |



# Using TO_CHAR Function with Numbers

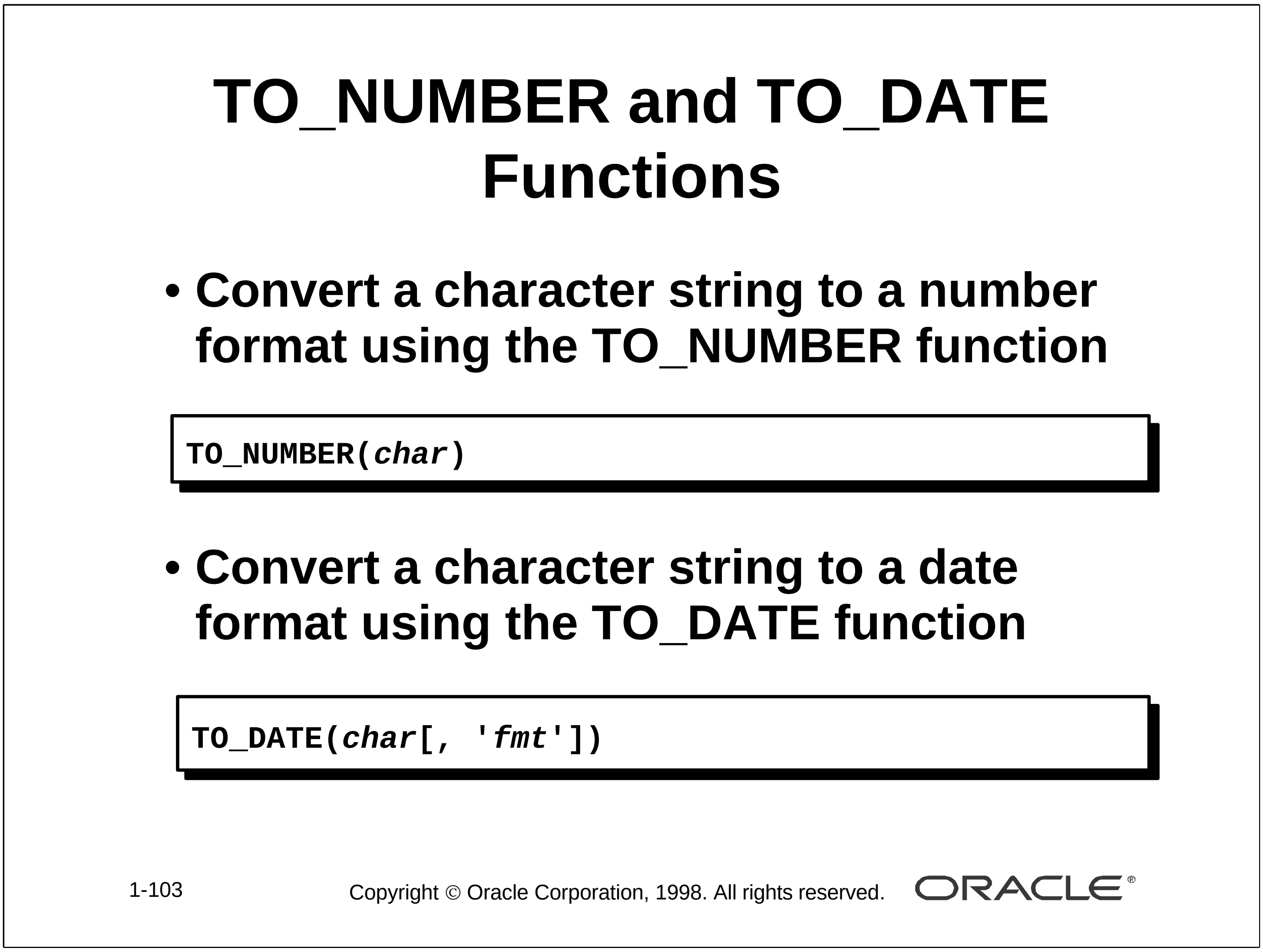
```
SQL> SELECT   TO_CHAR(sal,'$99,999') SALARY
  2  FROM     emp
  3  WHERE    ename = 'SCOTT';
```

```
  SALARY
--------
  $3,000
```

1-102

**Guidelines**

- The Oracle8 Server displays a string of pound signs (#) in place of a whole number whose digits exceed the number of digits provided in the format model.
- The Oracle8 Server rounds the stored decimal value to the number of decimal spaces provided in the format model.



# TO_NUMBER and TO_DATE Functions

- **Convert a character string to a number format using the TO_NUMBER function**

```
TO_NUMBER(char)
```

- **Convert a character string to a date format using the TO_DATE function**

```
TO_DATE(char[, 'fmt'])
```

### TO_NUMBER and TO_DATE Functions

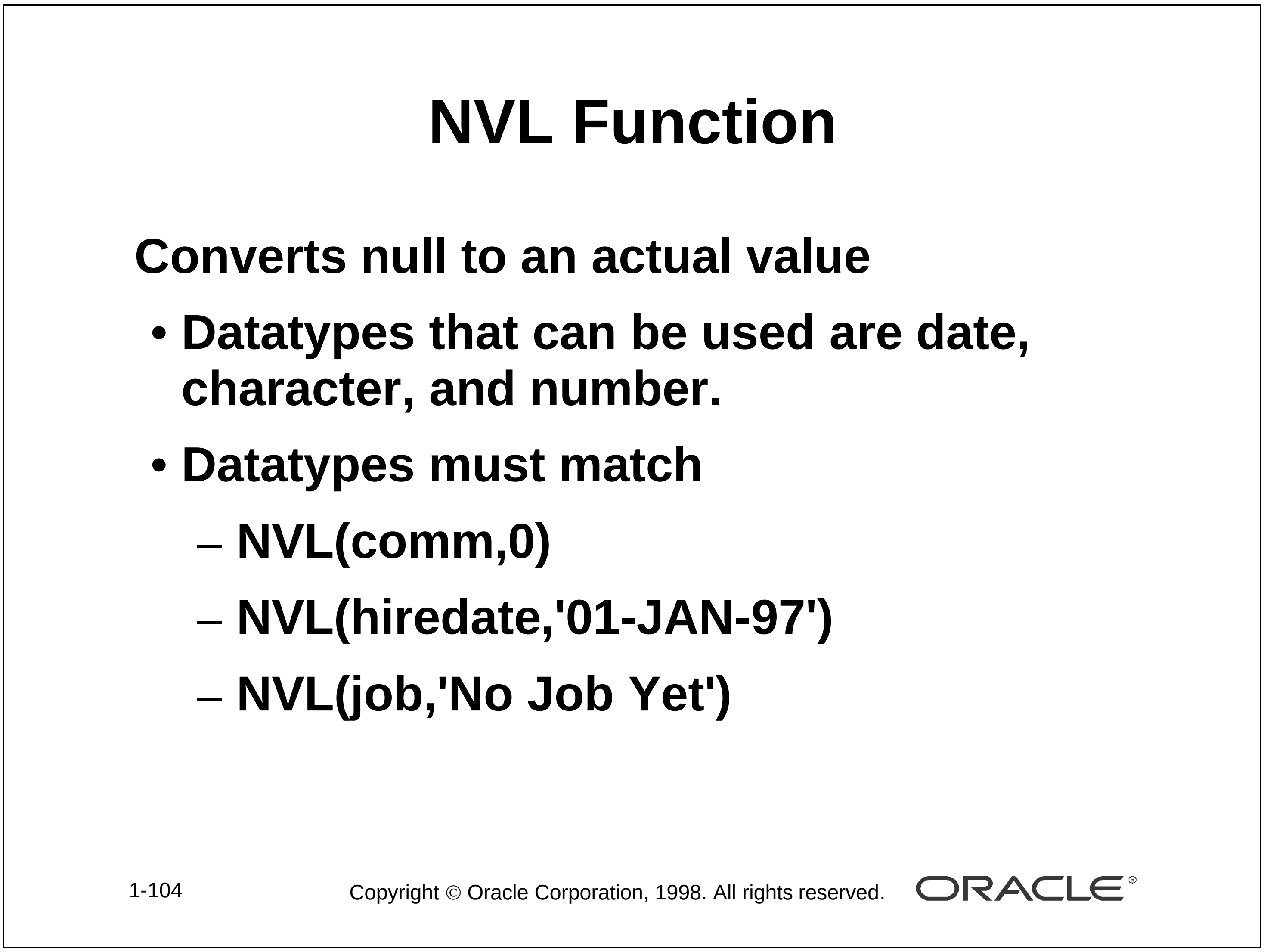
You may want to convert a character string to either a number or a date. To accomplish this task, you use the TO_NUMBER or TO_DATE functions. The format model you choose will be based on the previously demonstrated format elements.

### Example

Display the names and hire dates of all the employees who joined on February 22, 1981.

```
SQL> SELECT ename, hiredate
  2  FROM   emp
  3  WHERE  hiredate = TO_DATE(’February 22, 1981’, ’Month dd,  YYYY’);
```

```
ENAME      HIREDATE
---------- --------
WARD       22-FEB-81
```



# NVL Function

**Converts null to an actual value**

- **Datatypes that can be used are date, character, and number.**
- **Datatypes must match**
  - **NVL(comm,0)**
  - **NVL(hiredate,'01-JAN-97')**
  - **NVL(job,'No Job Yet')**

### The NVL Function

To convert a null value to an actual value, use the NVL function.

### Syntax

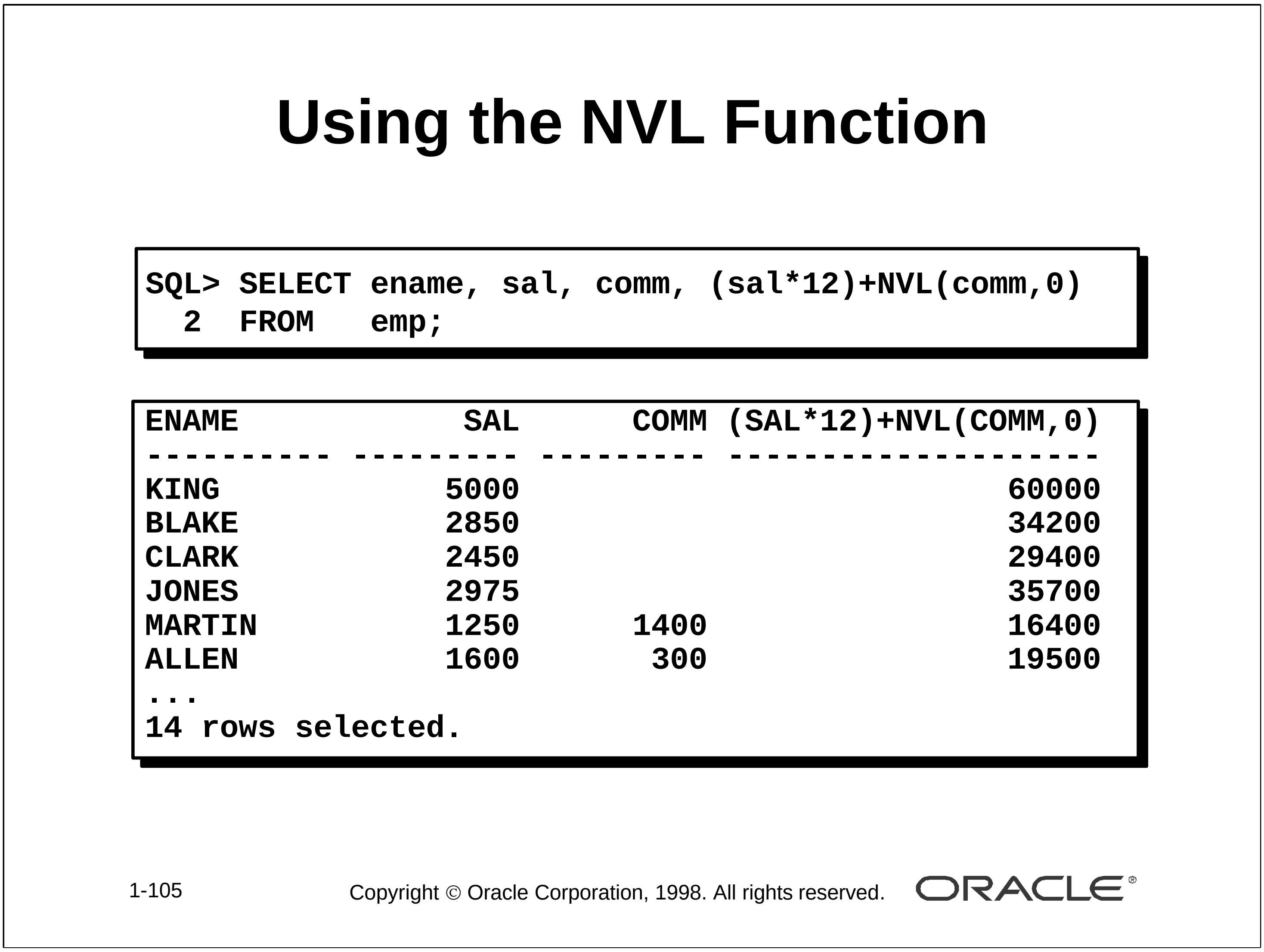
```
NVL (expr1, expr2)
```

| | | |
|---|---|---|
| **where:** | *expr1* | is the source value or expression that may contain null |
| | *expr2* | is the target value for converting null |

You can use the NVL function to convert any datatype, but the return value is always the same as the datatype of *expr1*.

### NVL Conversions for Various Datatypes

| **Datatype** | **Conversion Example** |
|---|---|
| NUMBER | NVL(*number_column*,9) |
| DATE | NVL(*date_column*, ‘01-JAN-95’) |
| CHAR or VARCHAR2 | NVL(*character_column*, ‘Unavailable’) |



# Using the NVL Function

```
SQL> SELECT ename, sal, comm, (sal*12)+NVL(comm,0)
  2  FROM   emp;
```

```
ENAME            SAL      COMM (SAL*12)+NVL(COMM,0)
---------- --------- --------- --------------------
KING            5000                          60000
BLAKE           2850                          34200
CLARK           2450                          29400
JONES           2975                          35700
MARTIN          1250      1400                16400
ALLEN           1600       300                19500
...
14 rows selected.
```

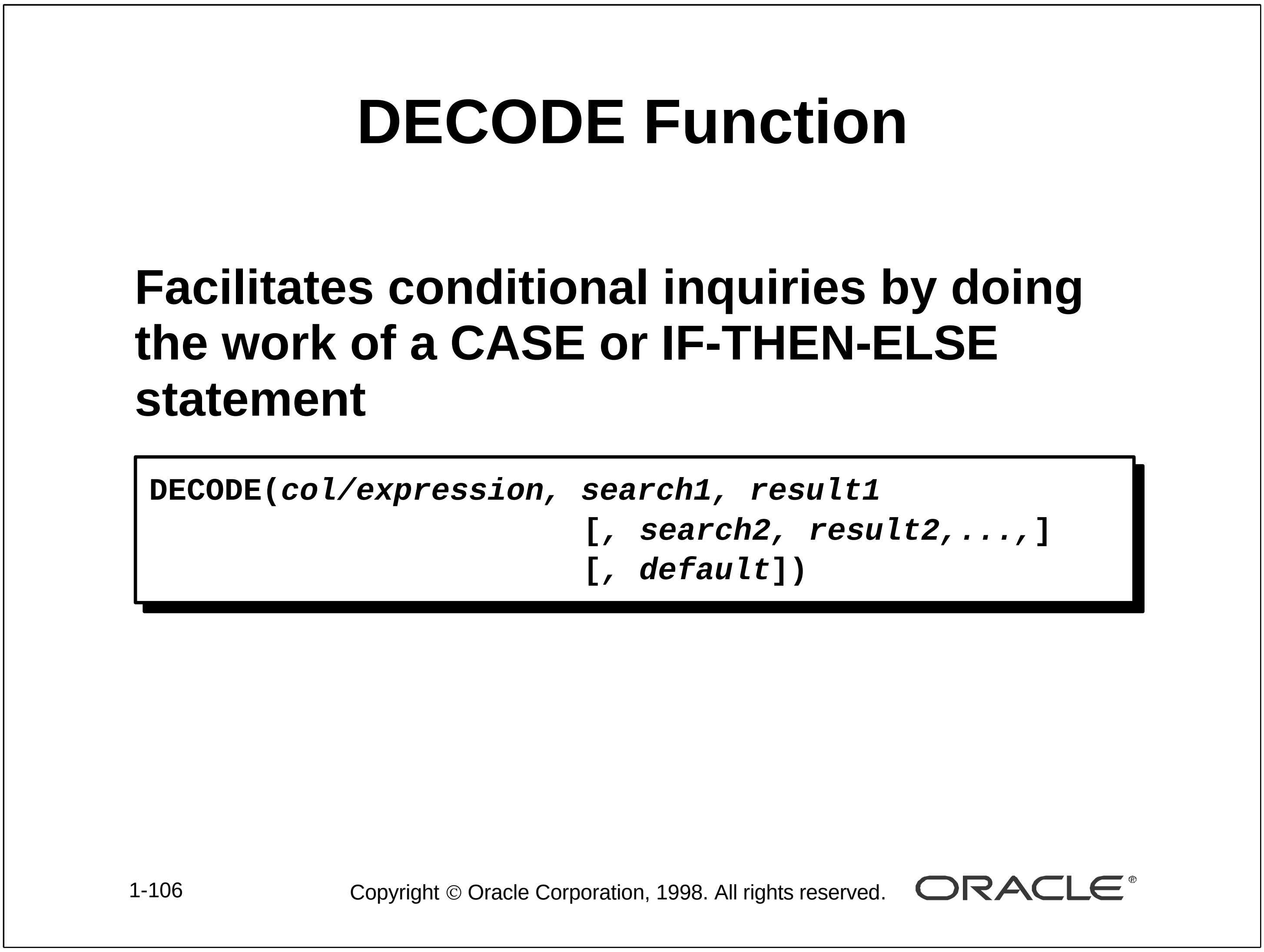
1-105

### NVL Function

To calculate the annual compensation of all employees, you need to multiply the monthly salary by 12 and then add the commission to it.

```
SQL> SELECT ename, sal, comm, (sal*12)+comm
  2  FROM   emp;

ENAME      JOB       (SAL*12)+COMM
---------- --------- -------------
KING       PRESIDENT
BLAKE      MANAGER
CLARK      MANAGER
JONES      MANAGER
MARTIN     SALESMAN          16400
...
14 rows selected.
```

Notice that the annual compensation is calculated only for those employees who earn a commission. If any column value in an expression is null, the result is null. To calculate values for all employees, you must convert the null value to a number before applying the arithmetic operator. In the example on the slide, the NVL function to is used to convert null values to zero.



# DECODE Function

## Facilitates conditional inquiries by doing the work of a CASE or IF-THEN-ELSE statement

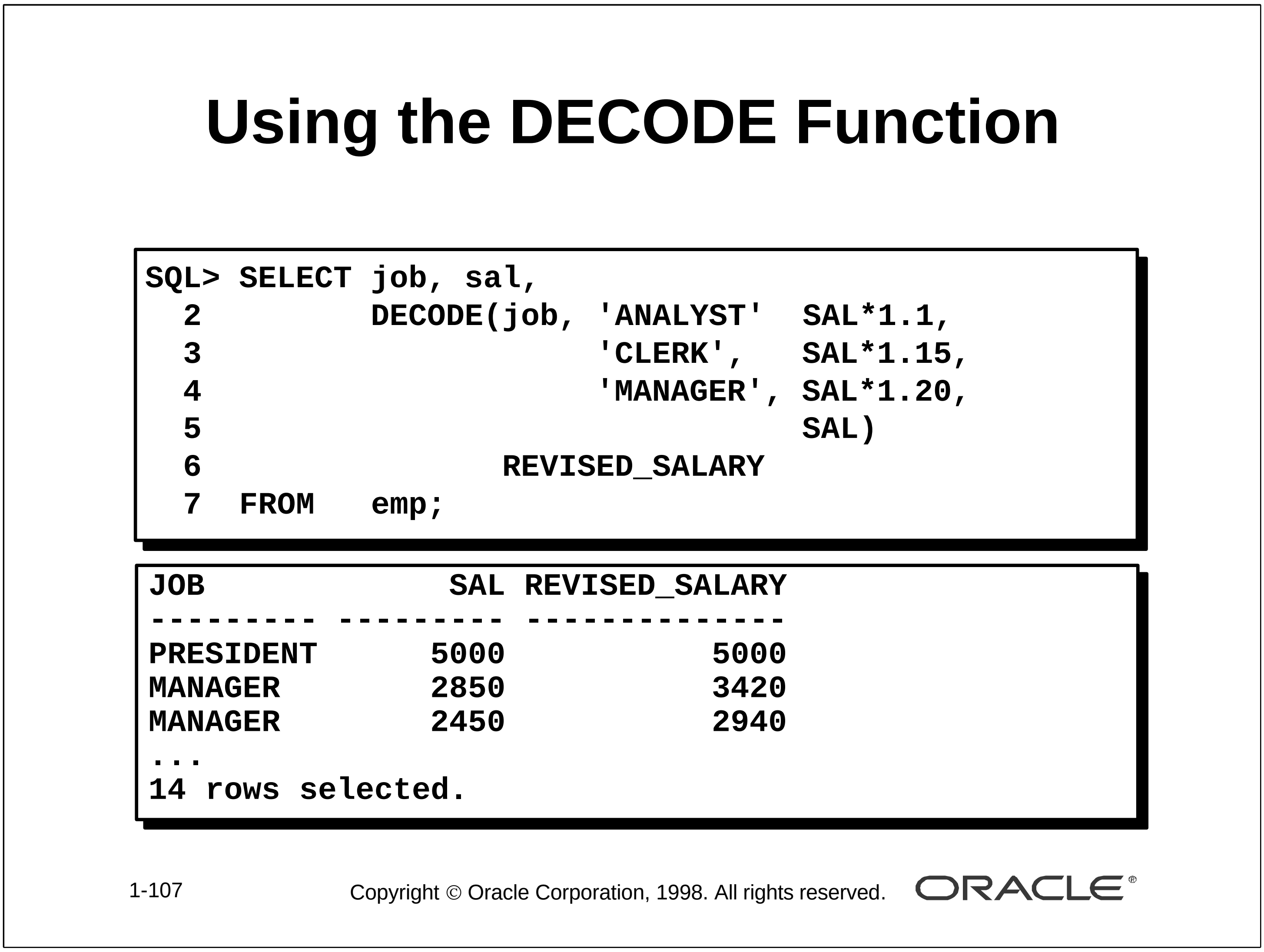
```
DECODE(col/expression, search1, result1
                          [, search2, result2,...,]
                          [, default])
```

1-106

### The DECODE Function

The DECODE function decodes an expression in a way similar to the IF-THEN-ELSE logic used in various languages. The DECODE function decodes *expression* after comparing it to each *search* value. If the expression is the same as *search*, *result* is returned.

If the default value is omitted, a null value will be returned where a search value does not match any of the result values.



# Using the DECODE Function

```
SQL> SELECT job, sal,
  2         DECODE(job, 'ANALYST'  SAL*1.1,
  3                     'CLERK',   SAL*1.15,
  4                     'MANAGER', SAL*1.20,
  5                                SAL)
  6                REVISED_SALARY
  7  FROM   emp;
```

```
JOB             SAL REVISED_SALARY
--------- --------- --------------
PRESIDENT      5000           5000
MANAGER        2850           3420
MANAGER        2450           2940
...
14 rows selected.
```

1-107

### Using the DECODE Function

In the SQL statement above, the value of JOB is decoded. If JOB is ANALYST, the salary increase is 10%; if JOB is CLERK, the salary increase is 15%; if JOB is MANAGER, the salary increase is 20%. For all other job roles, there is no increase in salary.

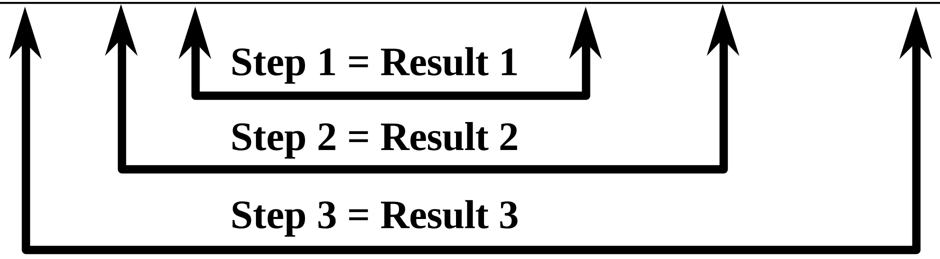
The same statement can be written as an IF-THEN-ELSE statement:

```
IF job = ’ANALYST’    THEN  sal = sal*1.1
IF job = ’CLERK’      THEN  sal = sal*1.15
IF job = ’MANAGER’    THEN  sal = sal*1.20
ELSE sal = sal
```

# Nesting Functions

- Single-row functions can be nested to any level
- Nested functions are evaluated from deepest level to the least deep level

```
F3(F2(F1(col,arg1),arg2),arg3)
```



Step 1 = Result 1

Step 2 = Result 2

Step 3 = Result 3

## Nesting Functions

Single-row functions can be nested to any depth. Nested functions are evaluated from the innermost level to the outermost level. Some examples follow to show you the flexibility of these functions.

# Nesting Functions

```
SQL> SELECT   ename,
  2            NVL(TO_CHAR(mgr),'No Manager')
  3  FROM     emp
  4  WHERE    mgr IS NULL;
```

```
ENAME      NVL(TO_CHAR(MGR),'NOMANAGER')
---------- -----------------------------
KING       No Manager
```

### Nesting Functions

The example above displays the head of the company, who has no manager. The evaluation of the SQL statement involves two steps:

1. Evaluate the inner function to convert a number value to a character string.
   – Result1 = TO_CHAR(mgr)
2. Evaluate the outer function to replace the null value with a text string.
   – NVL(Result1, ‘No Manager’)

The entire expression becomes the column heading since no column alias was given.

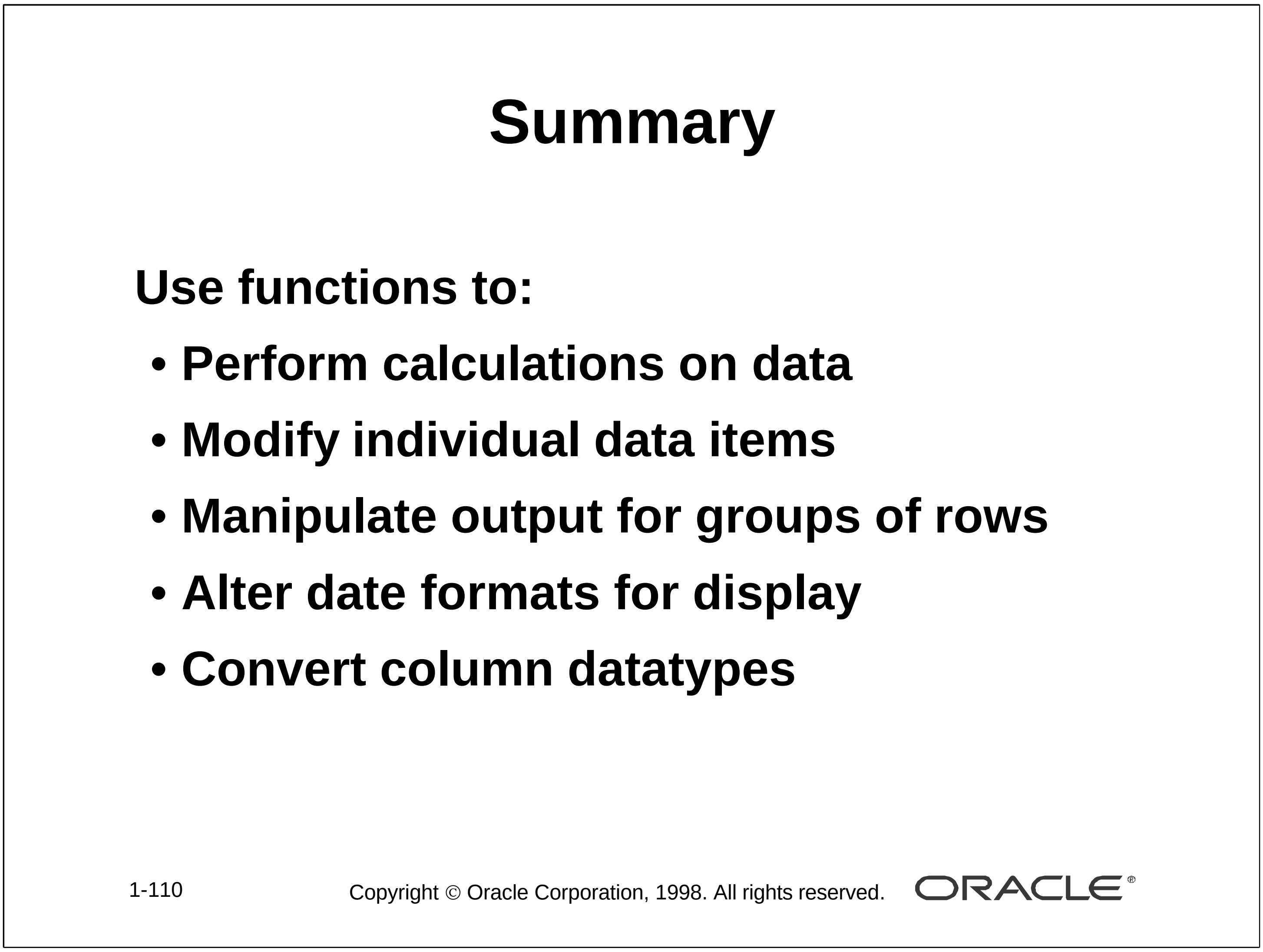
### Example

Display the date of the next Friday that is six months from the hire date. The resultant date should look like Friday, March 12th, 1982. Order the results by hire date.

```
SQL> SELECT    TO_CHAR(NEXT_DAY(ADD_MONTHS
  2            (hiredate, 6), ’FRIDAY’),
  3            ’fmDay, Month ddth, YYYY’)
  4            "Next 6 Month Review"
  5  FROM      emp
  6  ORDER BY  hiredate;
```

### Class Management Note

Demo: *l3nest.sql*



# Summary

Use functions to:

- Perform calculations on data
- Modify individual data items
- Manipulate output for groups of rows
- Alter date formats for display
- Convert column datatypes

1-110

ORACLE®

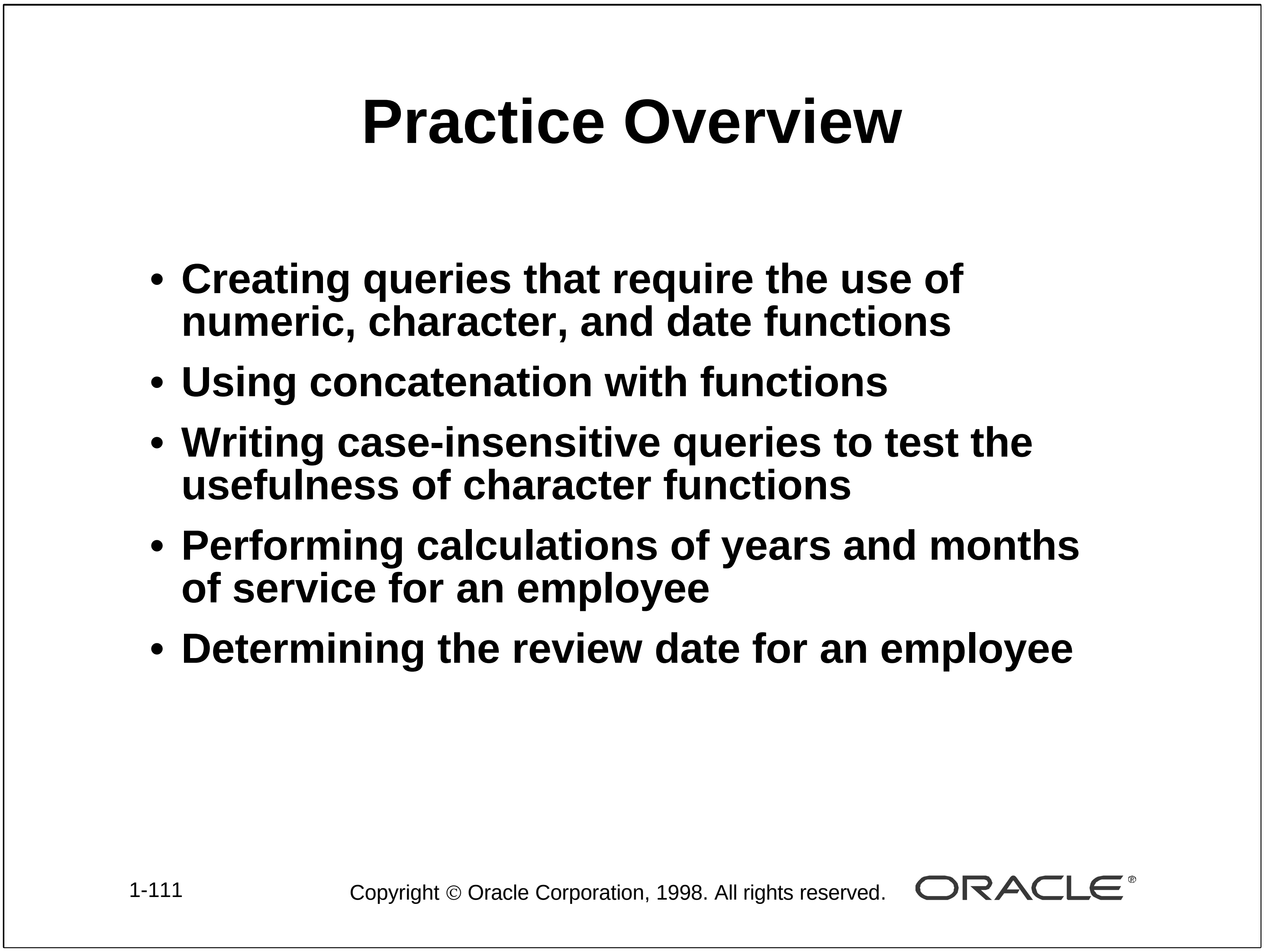
### Single-Row Functions

Single-row functions can be nested to any level. Single-row functions can manipulate

- Character data
  - LOWER, UPPER, INITCAP, CONCAT, SUBSTR, INSTR, LENGTH
- Number data
  - ROUND, TRUNC, MOD
- Date data
  - MONTHS_BETWEEN, ADD_MONTHS, NEXT_DAY, LAST_DAY, ROUND, TRUNC
  - Date values can also use arithmetic operators.
- Conversion functions can convert character, date, and numeric values.
  - TO_CHAR, TO_DATE, TO_NUMBER

### SYSDATE and DUAL

SYSDATE is a date function that returns the current date and time. It is customary to select SYSDATE from a dummy table called DUAL.



# Practice Overview

- Creating queries that require the use of numeric, character, and date functions
- Using concatenation with functions
- Writing case-insensitive queries to test the usefulness of character functions
- Performing calculations of years and months of service for an employee
- Determining the review date for an employee

1-111

### Practice Overview

This practice is designed to give you a variety of exercises using different functions available for character, number, and date data types.

Remember that for nested functions, the results are evaluated from the innermost function to the outermost function.

**Practice 3**

1. Write a query to display the current date. Label the column Date.

```
Date
---------
28-OCT-97
```

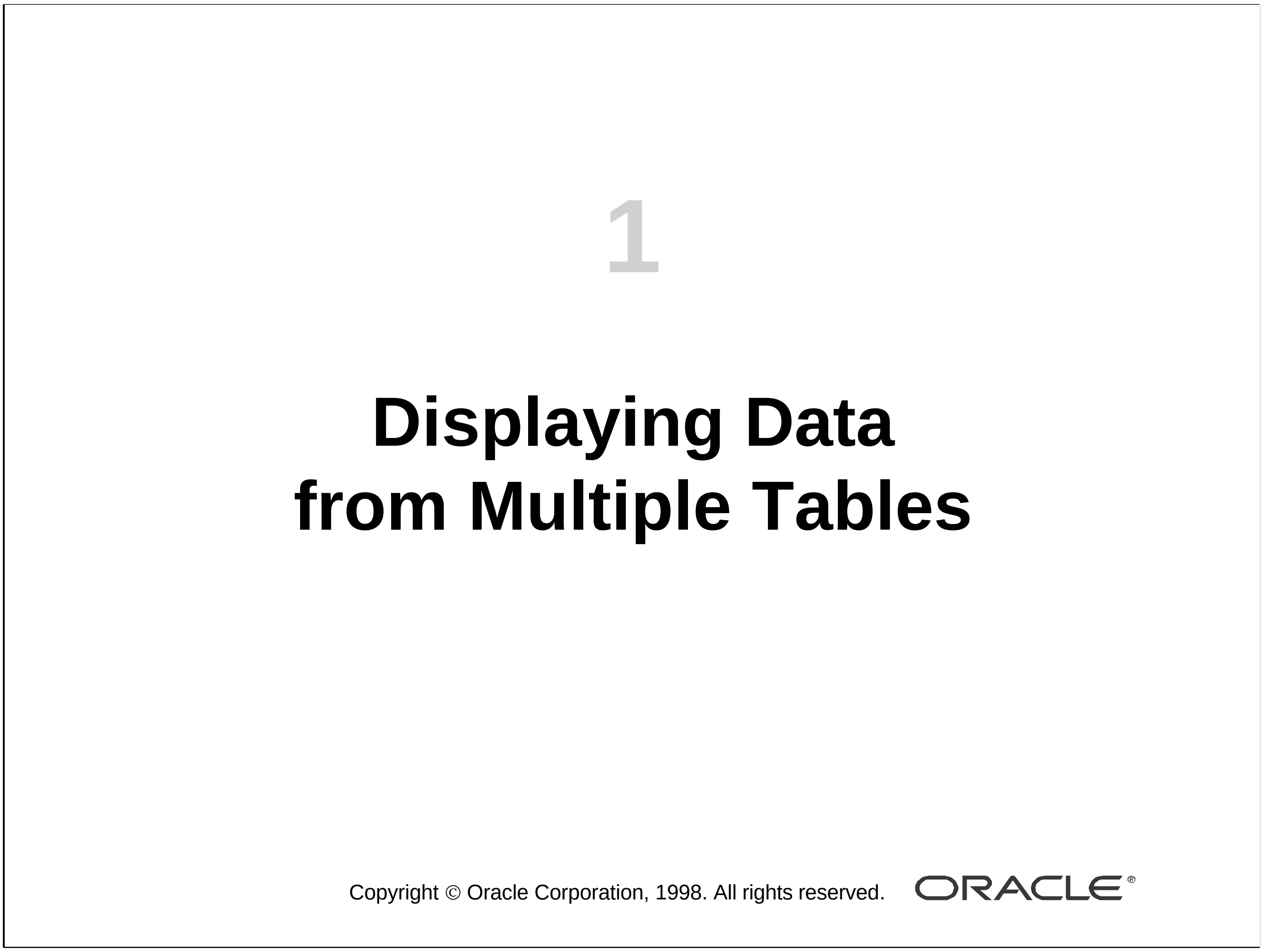
2. Display the employee number, name, salary, and salary increase by 15% expressed as a whole number. Label the column New Salary. Save your SQL statement to a file named *p3q2.sql*.

3. Run your query in the file *p3q2.sql*.

```
EMPNO ENAME      SAL New Salary
----- ------- ----- ----------
 7839 KING     5000       5750
 7698 BLAKE    2850       3278
 7782 CLARK    2450       2818
 7566 JONES    2975       3421
 7654 MARTIN   1250       1438
 7499 ALLEN    1600       1840
 7844 TURNER   1500       1725
 7900 JAMES     950       1093
 7521 WARD     1250       1438
 7902 FORD     3000       3450
 7369 SMITH     800        920
 7788 SCOTT    3000       3450
 7876 ADAMS    1100       1265
 7934 MILLER   1300       1495
14 rows selected.
```

4. Modify your query *p3q2.sql* to add an additional column that will subtract the old salary from the new salary. Label the column Increase. Rerun your query.

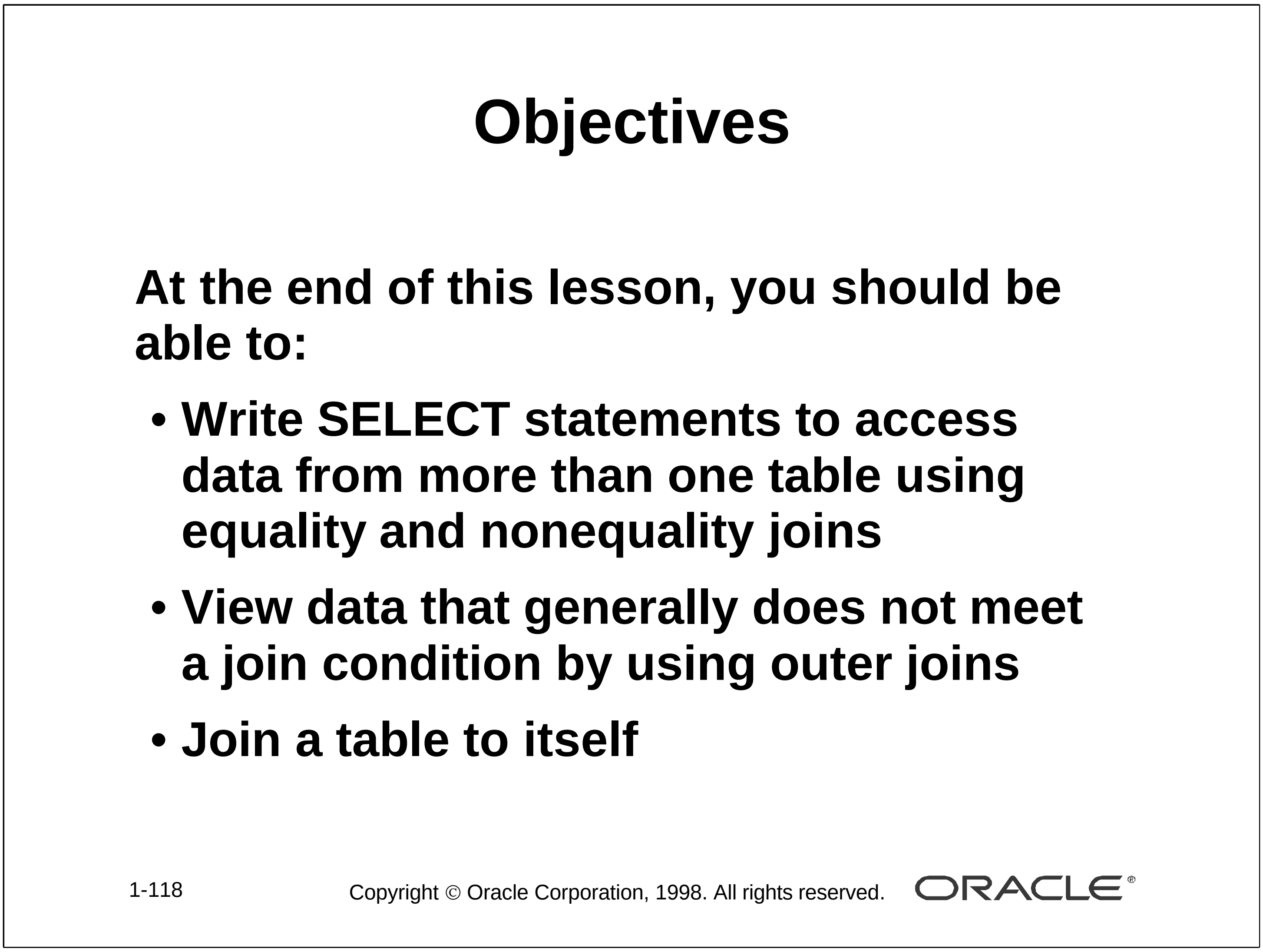
```
EMPNO ENAME     SAL New Salary Increase
----- ------ ----- ---------- --------
 7839 KING    5000       5750      750
 7698 BLAKE   2850       3278      428
 7782 CLARK   2450       2818      368
 7566 JONES   2975       3421      446
```



1

# Displaying Data from Multiple Tables

| Schedule: | Timing | Topic |
|---|---|---|
| | 40 minutes | Lecture |
| | 50 minutes | Practice |
| | 90 minutes | Total |



# Objectives

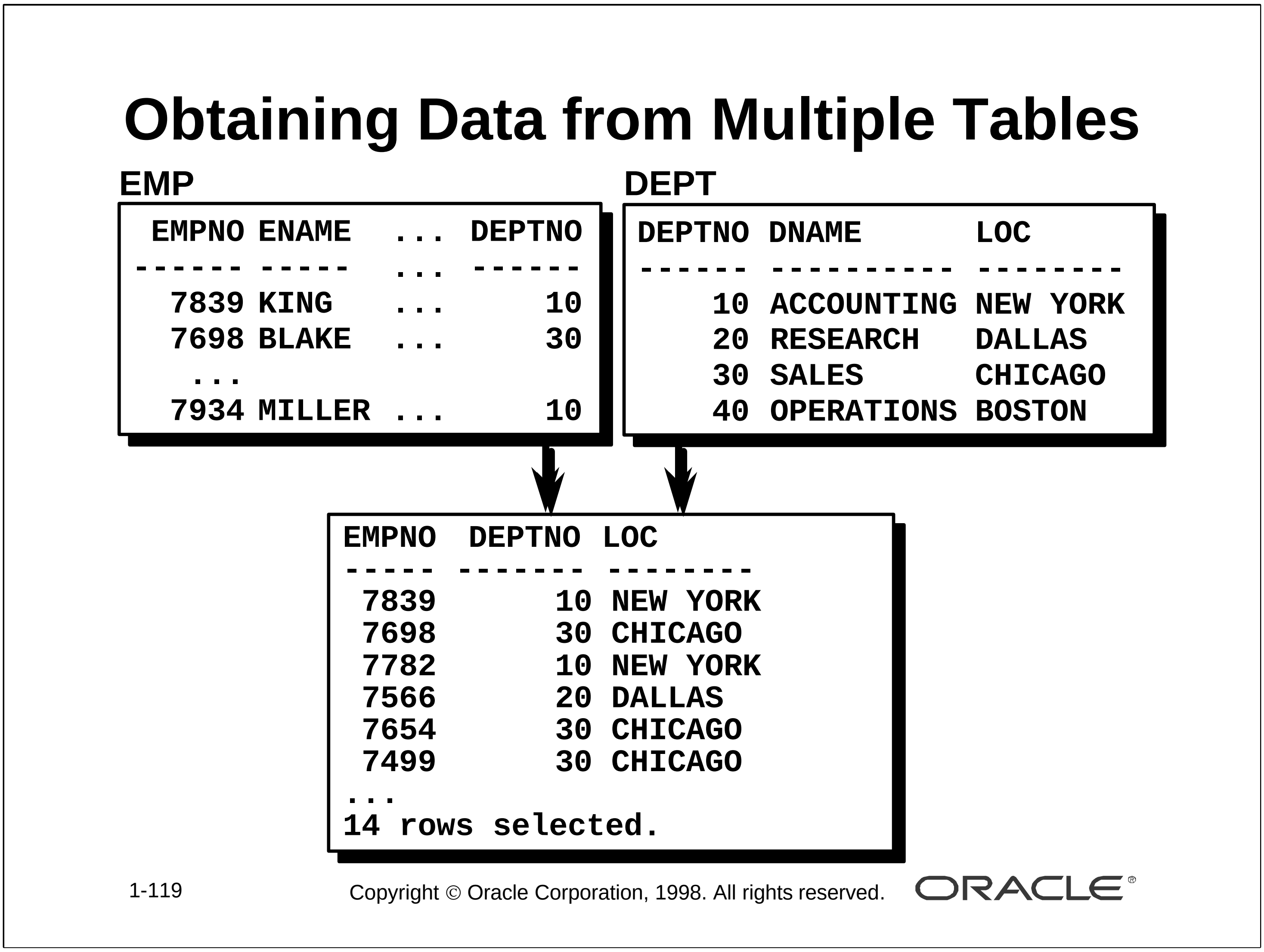
At the end of this lesson, you should be able to:

- Write SELECT statements to access data from more than one table using equality and nonequality joins
- View data that generally does not meet a join condition by using outer joins
- Join a table to itself

1-118

**Lesson Aim**

This lesson covers how to obtain data from more than one table, using the different methods available.



# Obtaining Data from Multiple Tables

EMP

```
 EMPNO ENAME   ...  DEPTNO
------ -----   ...  ------
  7839 KING    ...      10
  7698 BLAKE   ...      30
   ...
  7934 MILLER  ...      10
```

DEPT

```
DEPTNO DNAME      LOC
------ ---------- --------
    10 ACCOUNTING NEW YORK
    20 RESEARCH   DALLAS
    30 SALES      CHICAGO
    40 OPERATIONS BOSTON
```

```
EMPNO  DEPTNO LOC
----- ------- --------
 7839      10 NEW YORK
 7698      30 CHICAGO
 7782      10 NEW YORK
 7566      20 DALLAS
 7654      30 CHICAGO
 7499      30 CHICAGO
...
14 rows selected.
```

### Data from Multiple Tables

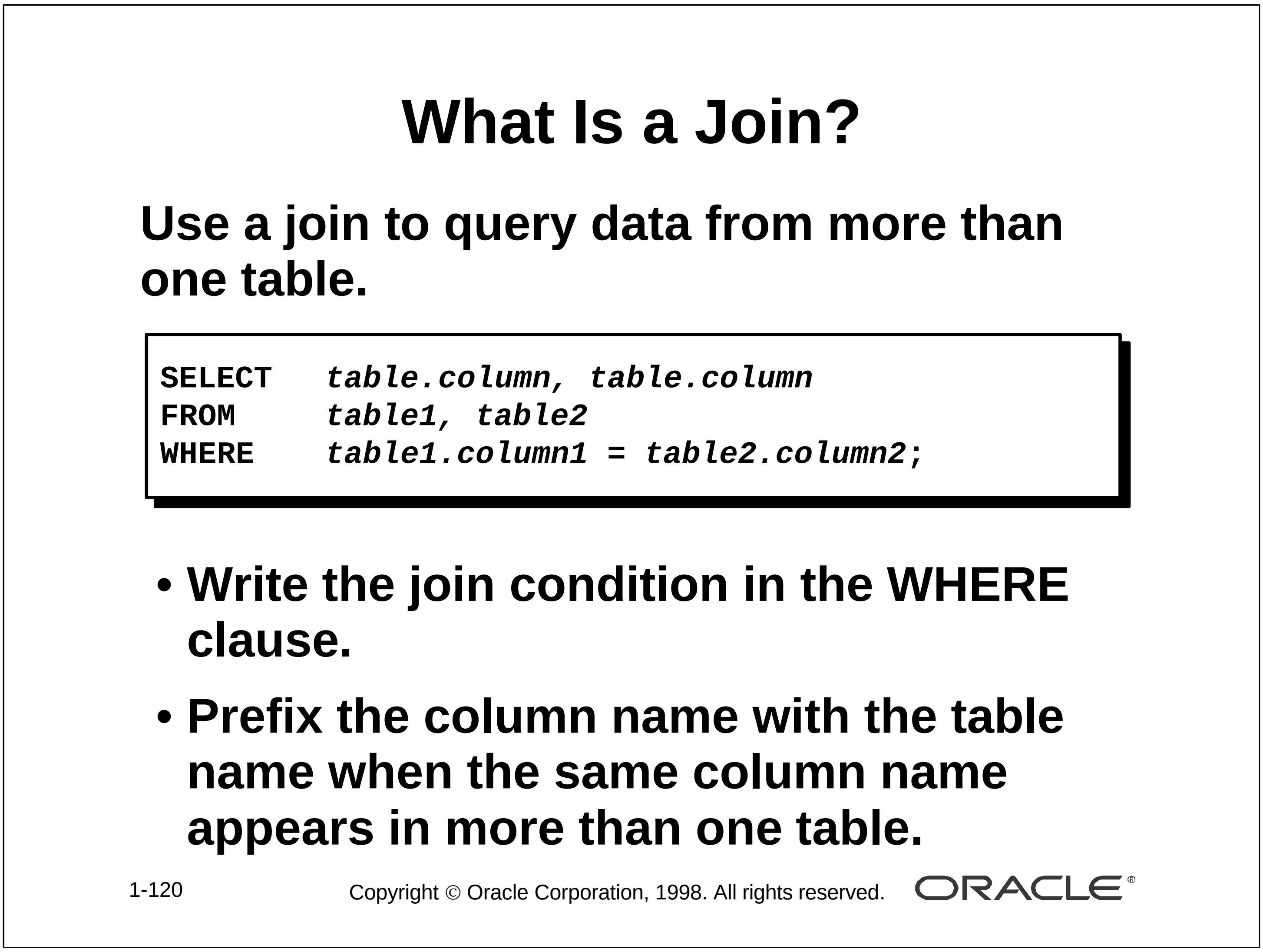
Sometimes you need to use data from more than one table. In the example above, the report displays data from two separate tables.

- EMPNO exists in the EMP table.
- DEPTNO exists in both the EMP and DEPT tables.
- LOC exists in DEPT table.

To produce the report, you need to link EMP and DEPT tables and access data from both of them.

### Class Management Note

In the slide above, the DEPTNO column can come from either the EMP or the DEPT table.



# What Is a Join?

**Use a join to query data from more than one table.**

```
SELECT   table.column, table.column
FROM     table1, table2
WHERE    table1.column1 = table2.column2;
```

- **Write the join condition in the WHERE clause.**
- **Prefix the column name with the table name when the same column name appears in more than one table.**

1-120

ORACLE®

### Defining Joins

When data from more than one table in the database is required, a *join* condition is used. Rows in one table can be joined to rows in another table according to common values existing in corresponding columns, that is, primary and foreign key columns.

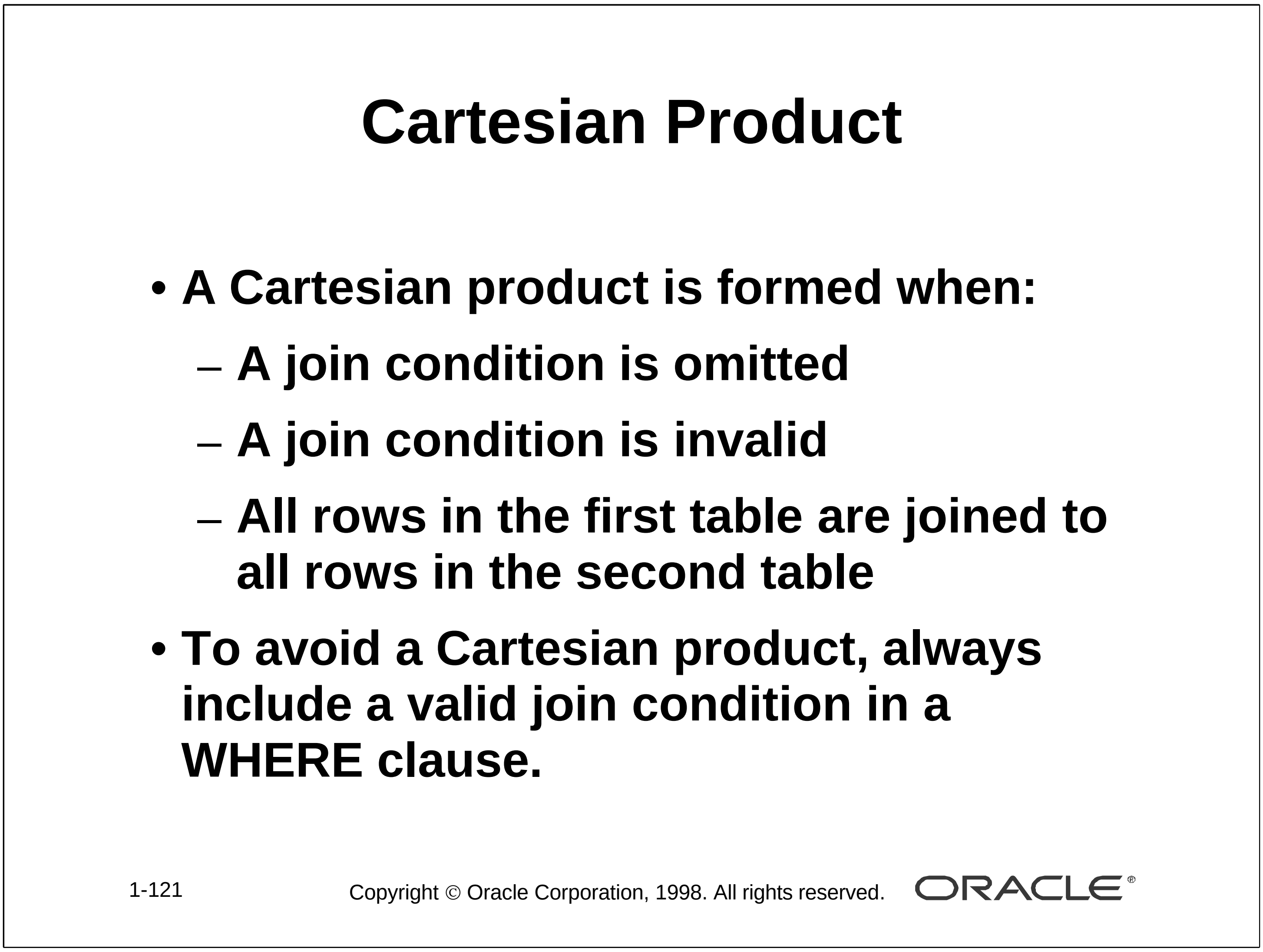
To display data from two or more related tables, write a simple join condition in the WHERE clause. In the syntax:

| | |
|---|---|
| *table.column* | denotes the table and column from which data is retrieved |
| *table1.column1 =*<br>*table2.column2* | is the condition that joins (or relates) the tables together |

### Guidelines

- When writing a SELECT statement that joins tables, precede the column name with the table name for clarity and to enhance database access.
- If the same column name appears in more than one table, the column name must be prefixed with the table name.
- To join *n* tables together, you need a minimum of (*n-1*) join conditions. Therefore, to join four tables, a minimum of three joins are required. This rule may not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row.

For more information, see
*Oracle8 Server SQL Language Reference Manual,* “SELECT.”



# Cartesian Product

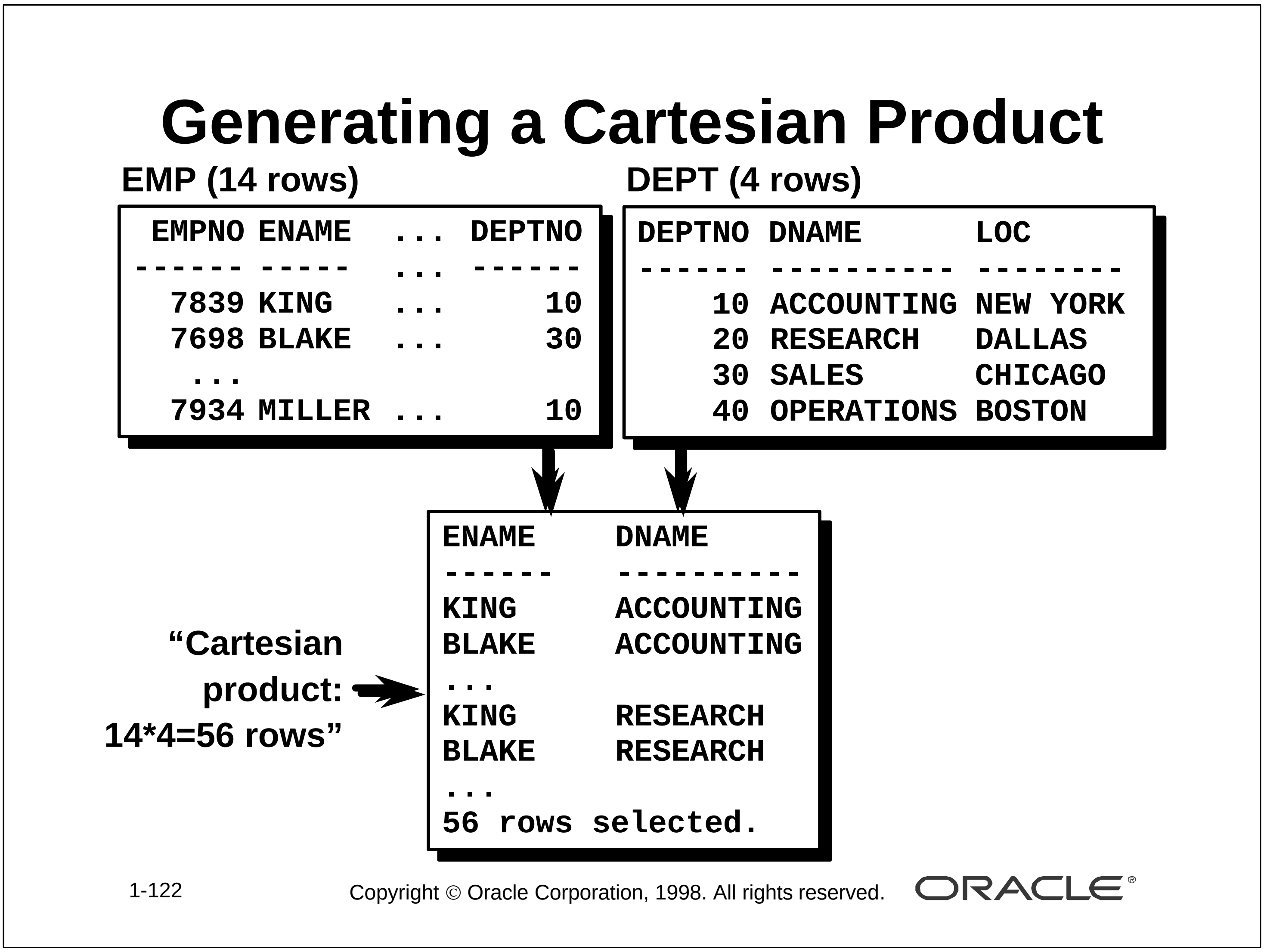
- A Cartesian product is formed when:
  - A join condition is omitted
  - A join condition is invalid
  - All rows in the first table are joined to all rows in the second table
- To avoid a Cartesian product, always include a valid join condition in a WHERE clause.

1-121

### Cartesian Product

When a join condition is invalid or omitted completely, the result is a *Cartesian product* in which all combinations of rows will be displayed. All rows in the first table are joined to all rows in the second table.

A Cartesian product tends to generate a large number of rows, and its result is rarely useful. You should always include a valid join condition in a WHERE clause, unless you have a specific need to combine all rows from all tables.



## Generating a Cartesian Product

EMP (14 rows)

```
 EMPNO ENAME   ...  DEPTNO
------ -----   ...  ------
  7839 KING    ...      10
  7698 BLAKE   ...      30
   ...
  7934 MILLER  ...      10
```

DEPT (4 rows)

```
DEPTNO DNAME      LOC
------ ---------- --------
    10 ACCOUNTING NEW YORK
    20 RESEARCH   DALLAS
    30 SALES      CHICAGO
    40 OPERATIONS BOSTON
```

"Cartesian product: 14*4=56 rows"

```
ENAME     DNAME
------    ----------
KING      ACCOUNTING
BLAKE     ACCOUNTING
...
KING      RESEARCH
BLAKE     RESEARCH
...
56 rows selected.
```

1-122

### Cartesian Product

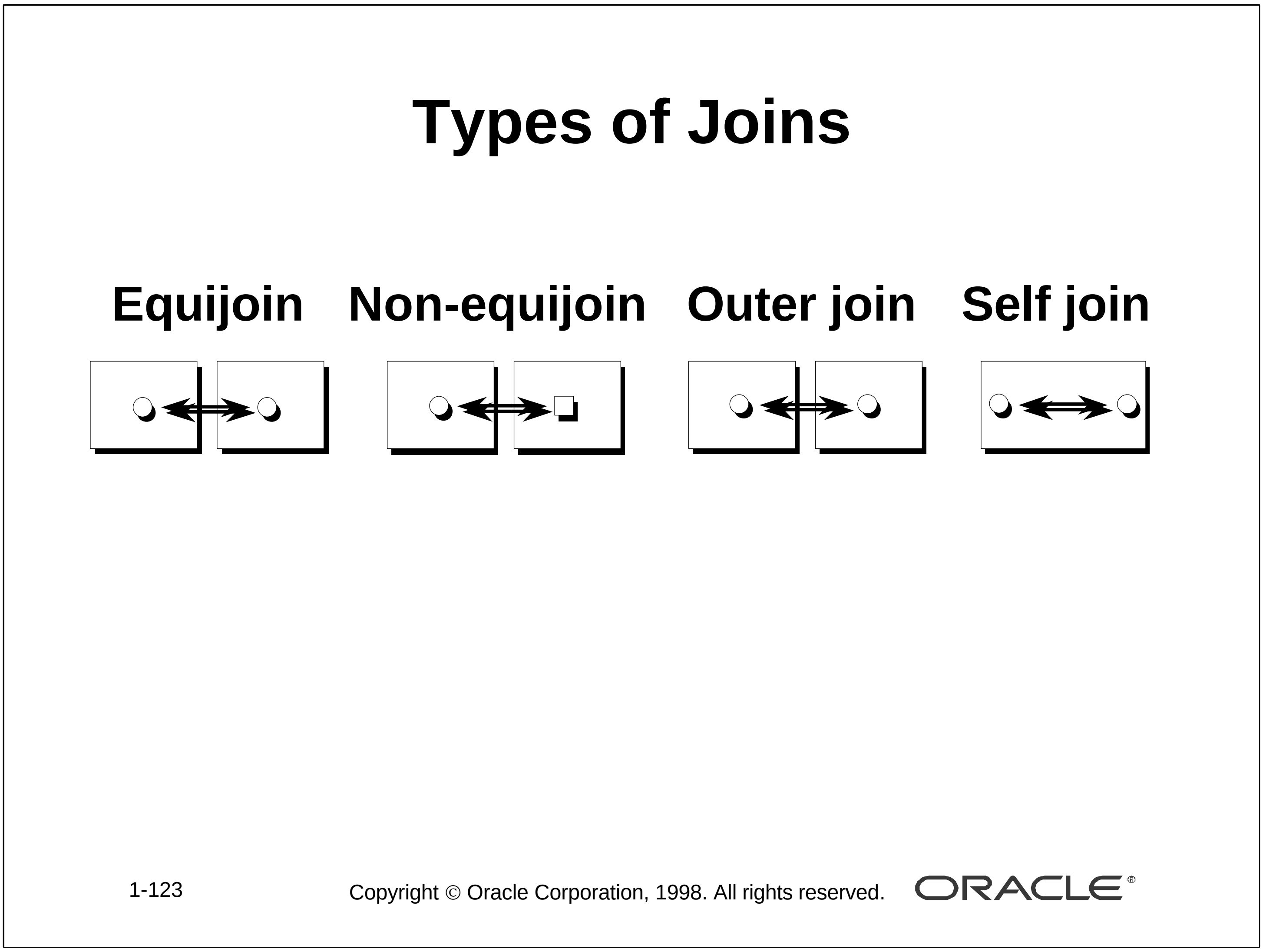
A Cartesian product is generated if a join condition is omitted. The example on the slide displays employee name and department name from EMP and DEPT tables. Because no WHERE clause has been specified, all rows (14 rows) from the EMP table are joined with all rows (4 rows) in the DEPT table, thereby generating 56 rows in the output.

```
SQL> SELECT ename, dname
  2  FROM   emp, dept;
```

```
ENAME      DNAME
---------- --------------
KING       ACCOUNTING
BLAKE      ACCOUNTING
...
KING       RESEARCH
BLAKE      RESEARCH
...
56 rows selected.
```

**Class Management Note**

Demo: *l4cart.sql*



# Types of Joins

Equijoin Non-equijoin Outer join Self join

1-123

### Types of Joins

There are two main types of join conditions:

- Equijoins
- Non-equijoins

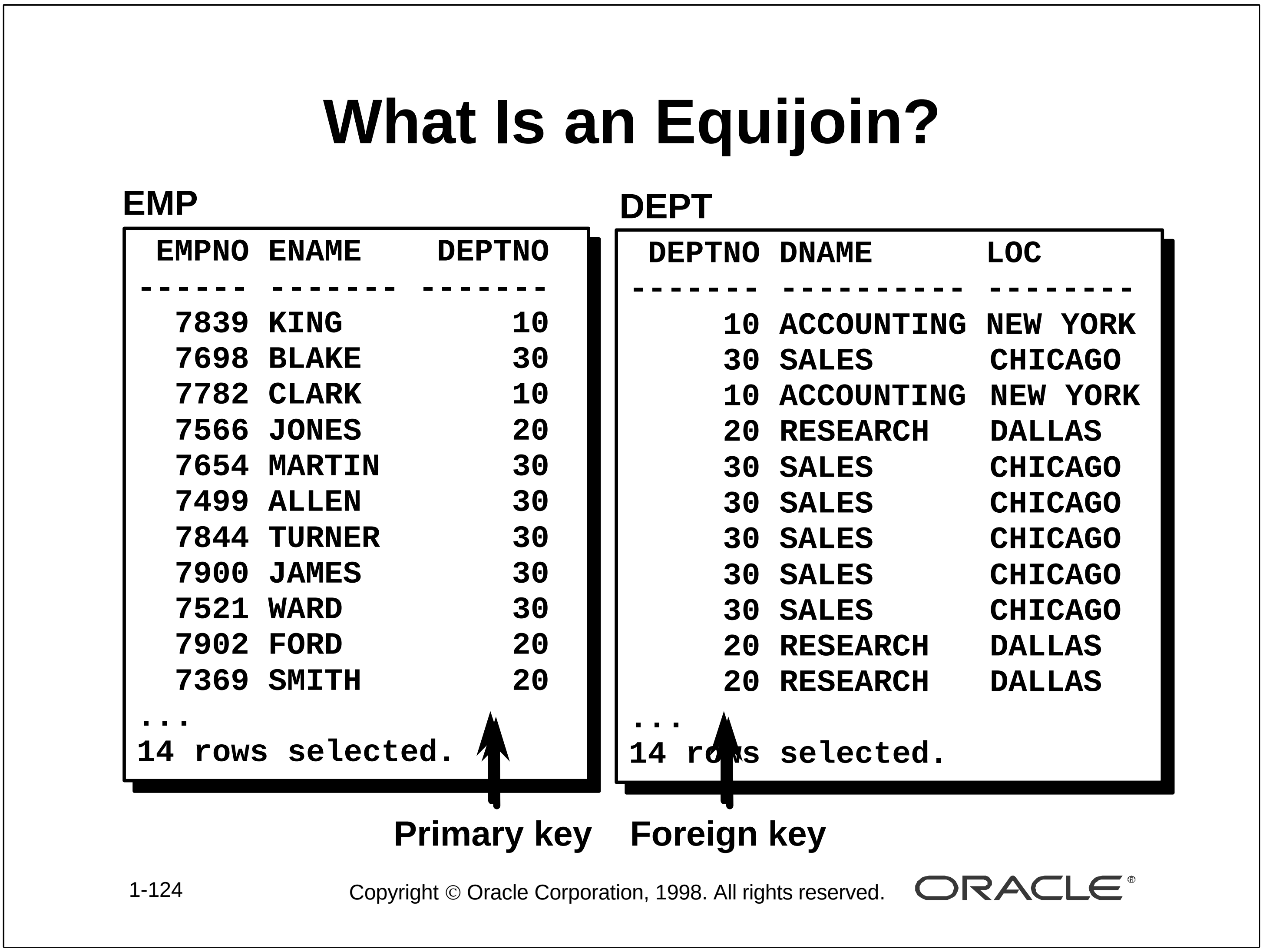
Additional join methods include the following:

- Outer joins
- Self joins
- Set operators

**Note:** Set operators are not covered in this course. They are covered in Advanced SQL and SQL*Plus.

### Class Management Note

Do not get into details of all the types of joins. Explain each join one by one as is done in the following slides.



# What Is an Equijoin?

EMP

```
 EMPNO ENAME    DEPTNO
------ ------- -------
  7839 KING         10
  7698 BLAKE        30
  7782 CLARK        10
  7566 JONES        20
  7654 MARTIN       30
  7499 ALLEN        30
  7844 TURNER       30
  7900 JAMES        30
  7521 WARD         30
  7902 FORD         20
  7369 SMITH        20
...
14 rows selected.
```

DEPT

```
 DEPTNO DNAME      LOC
------- ---------- --------
     10 ACCOUNTING NEW YORK
     30 SALES      CHICAGO
     10 ACCOUNTING NEW YORK
     20 RESEARCH   DALLAS
     30 SALES      CHICAGO
     30 SALES      CHICAGO
     30 SALES      CHICAGO
     30 SALES      CHICAGO
     30 SALES      CHICAGO
     20 RESEARCH   DALLAS
     20 RESEARCH   DALLAS
...
14 rows selected.
```

1-124

### Equijoins

To determine the name of an employee’s department, you compare the value in the DEPTNO column in the EMP table with the DEPTNO values in the DEPT table. The relationship between the EMP and DEPT tables is an *equijoin*—that is, values in the DEPTNO column on both tables must be equal. Frequently, this type of joins involve primary and foreign key complements.

**Note:** Equijoins are also called simple joins or inner joins.

### Class Management Note

Explain the use of decision matrix for simplifying writing joins. For example, if you want to display the name and department number of all the employees who are in the same department as Smith, you can start by making the following decision tree:

| Columns to Display | Originating Table | Condition |
|---|---|---|
| ename<br>dname | emp<br>dept | ename='SMITH'<br>emp.deptno = dept.deptno |

Now, the SQL statement can be easily formulated by looking at the decision matrix. The first column gives the column list in the SELECT statement, the second column gives the tables for the FROM clause, and the third column gives the condition for the WHERE clause.

# Retrieving Records with Equijoins

```
SQL> SELECT   emp.empno, emp.ename, emp.deptno,
  2           dept.deptno, dept.loc
  3  FROM     emp, dept
  4  WHERE    emp.deptno=dept.deptno;
```

```
EMPNO ENAME   DEPTNO DEPTNO LOC
----- ------ ------ ------ ---------
 7839 KING        10     10 NEW YORK
 7698 BLAKE       30     30 CHICAGO
 7782 CLARK       10     10 NEW YORK
 7566 JONES       20     20 DALLAS
...
14 rows selected.
```

### Retrieving Records with Equijoins

In the above example:

- The SELECT clause specifies the column names to retrieve:
  - employee name, employee number, and department number, which are columns in the EMP table
  - department number, department name, and location, which are columns in the DEPT table
- The FROM clause specifies the two tables that the database must access:
  - EMP table
  - DEPT table
- The WHERE clause specifies how the tables are to be joined:

  EMP.DEPTNO=DEPT.DEPTNO

Because the DEPTNO column is common to both tables, it must be prefixed by the table name to avoid ambiguity.

# Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Improve performance by using table prefixes.
- Distinguish columns that have identical names but reside in different tables by using column aliases.

### Qualifying Ambiguous Column Names

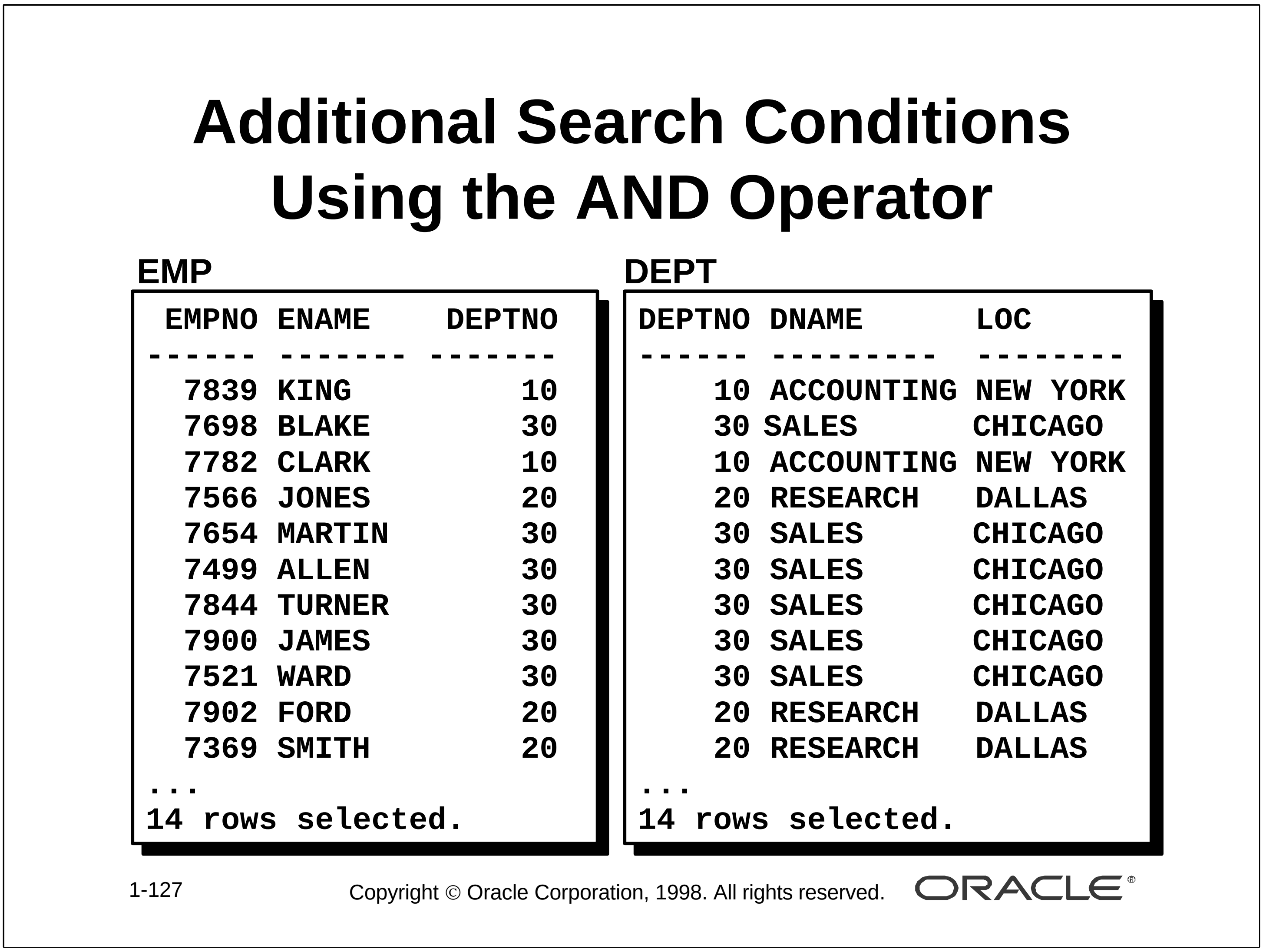
You need to qualify the names of the columns in the WHERE clause with the table name to avoid ambiguity. Without the table prefixes, the DEPTNO column could be from either the DEPT table or the EMP table. It is necessary to add the table prefix to execute your query.

If there are no common column names between the two tables, there is no need to qualify the columns. However, you will gain improved performance by using the table prefix because you tell the Oracle8 Server exactly where to go to find columns.

The requirement to qualify ambiguous column names is also applicable to columns that may be ambiguous in other clauses such as SELECT clause or ORDER BY clause.

### Class Management Note

Demo: *l4loc.sql*



# Additional Search Conditions Using the AND Operator

EMP

```
 EMPNO ENAME    DEPTNO
------ ------- -------
  7839 KING         10
  7698 BLAKE        30
  7782 CLARK        10
  7566 JONES        20
  7654 MARTIN       30
  7499 ALLEN        30
  7844 TURNER       30
  7900 JAMES        30
  7521 WARD         30
  7902 FORD         20
  7369 SMITH        20
...
14 rows selected.
```

DEPT

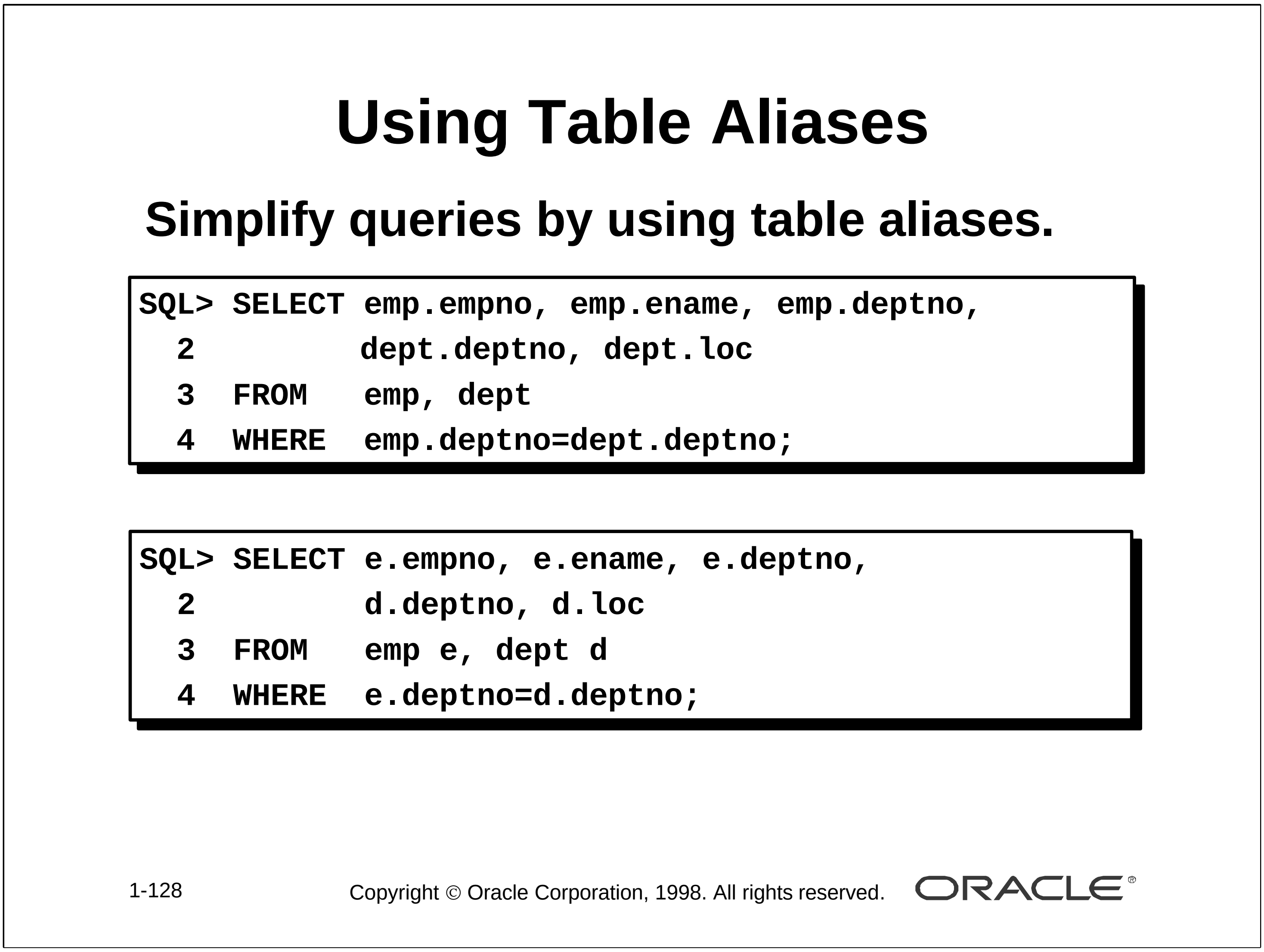
```
DEPTNO DNAME      LOC
------ ---------  --------
    10 ACCOUNTING NEW YORK
    30 SALES      CHICAGO
    10 ACCOUNTING NEW YORK
    20 RESEARCH   DALLAS
    30 SALES      CHICAGO
    30 SALES      CHICAGO
    30 SALES      CHICAGO
    30 SALES      CHICAGO
    30 SALES      CHICAGO
    20 RESEARCH   DALLAS
    20 RESEARCH   DALLAS
...
14 rows selected.
```

### Additional Search Conditions

In addition to the join, you may have additional criteria for your WHERE clause. For example, to display employee King’s employee number, name, department number, and department location, you need an additional condition in the WHERE clause.

```
SQL> SELECT empno, ename, emp.deptno, loc
  2  FROM   emp, dept;
  3  WHERE  emp.deptno = dept.deptno
  4  AND    INITCAP(ename) = 'King';
```

```
    EMPNO ENAME         DEPTNO LOC
--------- ---------- --------- -------------
     7839 KING              10 NEW YORK
```



# Using Table Aliases

## Simplify queries by using table aliases.

```
SQL> SELECT emp.empno, emp.ename, emp.deptno,
  2         dept.deptno, dept.loc
  3  FROM   emp, dept
  4  WHERE  emp.deptno=dept.deptno;
```

```
SQL> SELECT e.empno, e.ename, e.deptno,
  2         d.deptno, d.loc
  3  FROM   emp e, dept d
  4  WHERE  e.deptno=d.deptno;
```

### Table Aliases

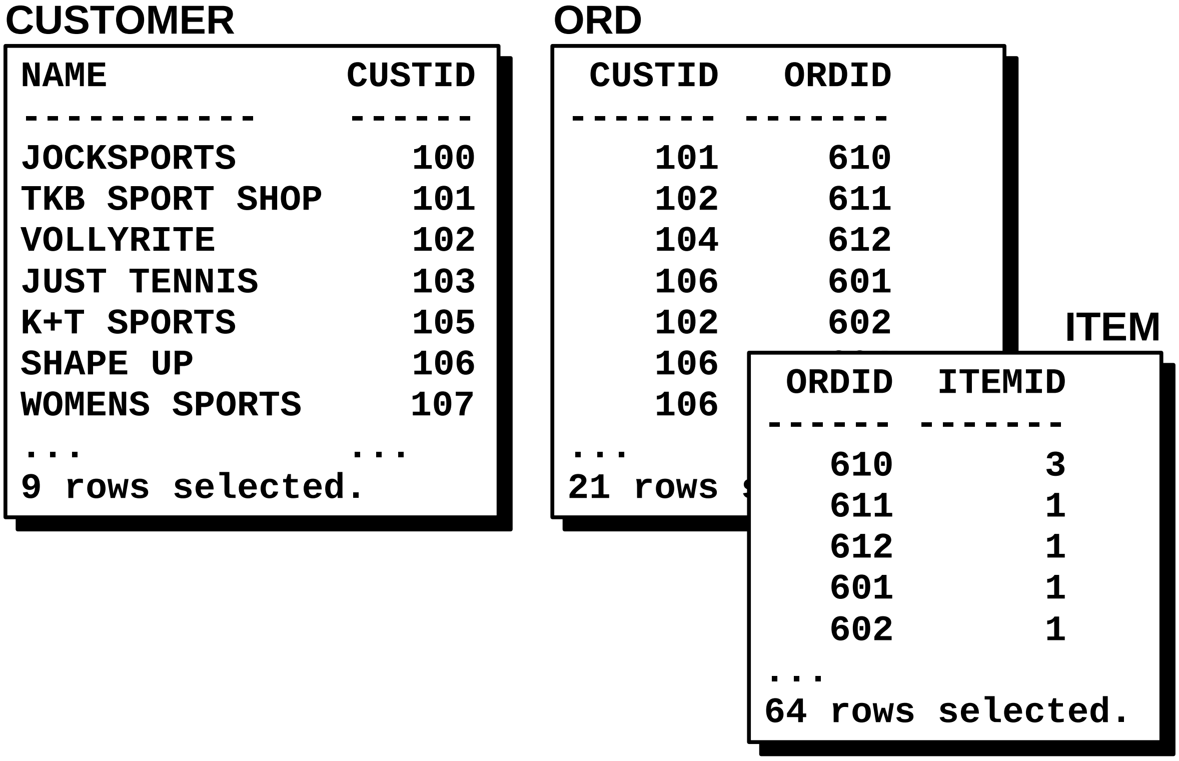
Qualifying column names with table names can be very time consuming, particularly if table names are lengthy. You can use table *aliases* instead of table names. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, therefore using less memory.

Notice how table aliases are identified in the FROM clause in the example. The table name is specified in full, followed by a space and then the table alias. The EMP table has been given an alias of E, whereas the DEPT table has an alias of D.

### Guidelines

- Table aliases can be up to 30 characters in length, but the shorter they are the better.
- If a table alias is used for a particular table name in the FROM clause, then that table alias must be substituted for the table name throughout the SELECT statement.
- Table aliases should be meaningful.
- The table alias is valid only for the current SELECT statement.

# Joining More Than Two Tables



CUSTOMER

```
NAME             CUSTID
-----------      ------
JOCKSPORTS          100
TKB SPORT SHOP      101
VOLLYRITE           102
JUST TENNIS         103
K+T SPORTS          105
SHAPE UP            106
WOMENS SPORTS       107
...              ...
9 rows selected.
```

ORD

```
 CUSTID   ORDID
------- -------
    101     610
    102     611
    104     612
    106     601
    102     602
    106
    106
...
21 rows s
```

ITEM

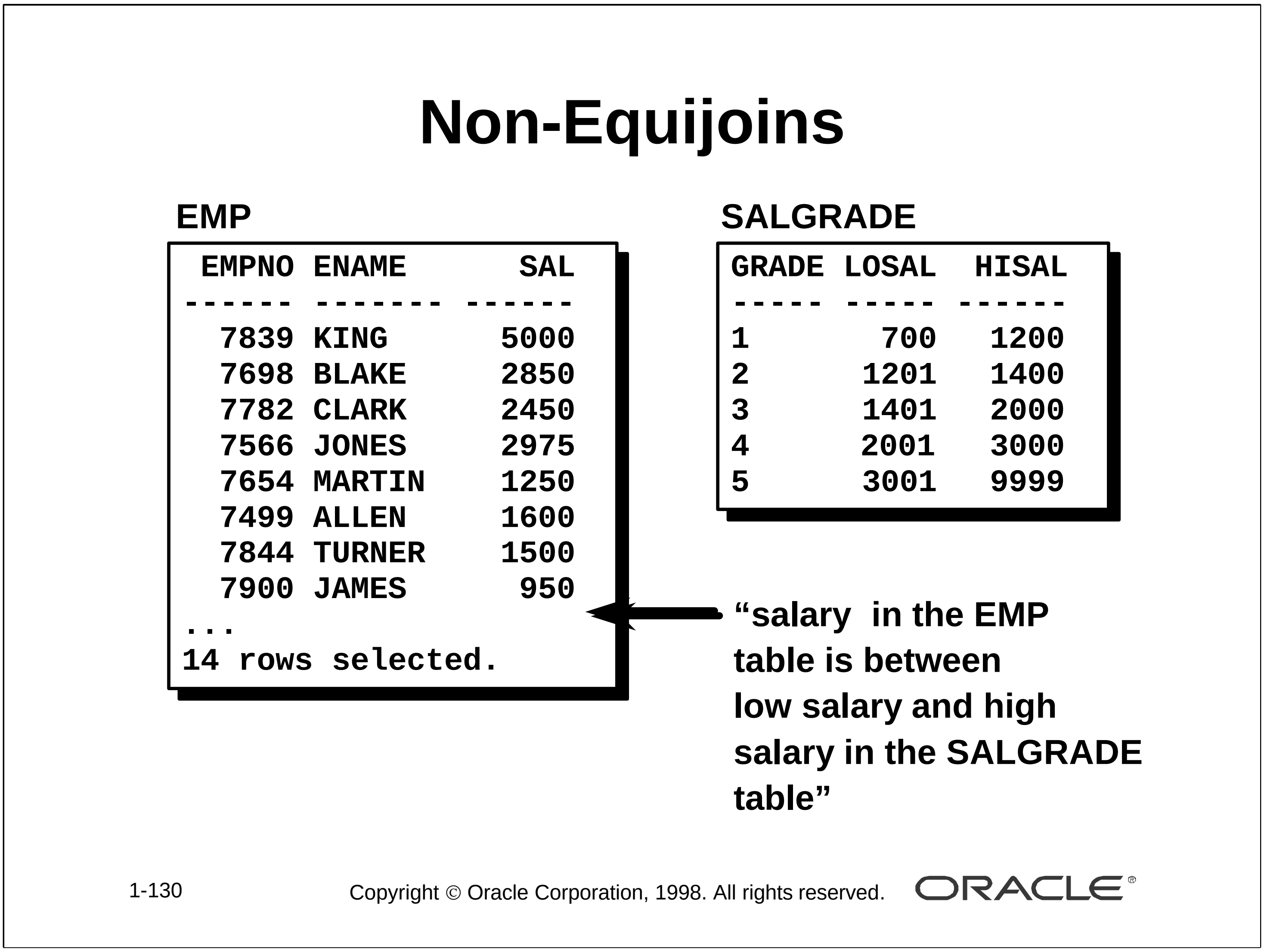
```
 ORDID  ITEMID
------ -------
   610       3
   611       1
   612       1
   601       1
   602       1
...
64 rows selected.
```

### Additional Search Conditions

Sometimes you may need to join more than two tables. For example, to display the name, the orders placed, the item numbers, the total for each item, and the total for each order for customer TKB SPORT SHOP, you will have to join the CUSTOMER, ORD, and ITEM tables.

```
SQL> SELECT c.name, o.ordid, i.itemid, i.itemtot, o.total
  2  FROM   customer c, ord o, item i
  3  WHERE  c.custid = o.custid
  4  AND    o.ordid = i.ordid
  5  AND    c.name = 'TKB SPORT SHOP';
```

```
NAME               ORDID    ITEMID   ITEMTOT     TOTAL
------------ --------- --------- --------- ---------
TKB SPORT SHOP       610         3        58     101.4
TKB SPORT SHOP       610         1        35     101.4
TKB SPORT SHOP       610         2       8.4     101.4
```



# Non-Equijoins

EMP

```
 EMPNO ENAME        SAL
------ ------- ------
  7839 KING       5000
  7698 BLAKE      2850
  7782 CLARK      2450
  7566 JONES      2975
  7654 MARTIN     1250
  7499 ALLEN      1600
  7844 TURNER     1500
  7900 JAMES       950
...
14 rows selected.
```

SALGRADE

```
GRADE LOSAL  HISAL
----- ----- ------
1       700   1200
2      1201   1400
3      1401   2000
4      2001   3000
5      3001   9999
```

**“salary in the EMP table is between low salary and high salary in the SALGRADE table”**

### Non-Equijoins

The relationship between the EMP table and the SALGRADE table is a non-equijoin, meaning that no column in the EMP table corresponds directly to a column in the SALGRADE table. The relationship between the two tables is that the SAL column in the EMP table is between the LOSAL and HISAL column of the SALGRADE table. The relationship is obtained using an operator other than equal (=).

# Retrieving Records with Non-Equijoins

```
SQL>  SELECT   e.ename, e.sal, s.grade
   2  FROM     emp e, salgrade s
   3  WHERE    e.sal
   4  BETWEEN  s.losal AND s.hisal;
```

```
ENAME            SAL     GRADE
---------- --------- ---------
JAMES            950         1
SMITH            800         1
ADAMS           1100         1
...
14 rows selected.
```

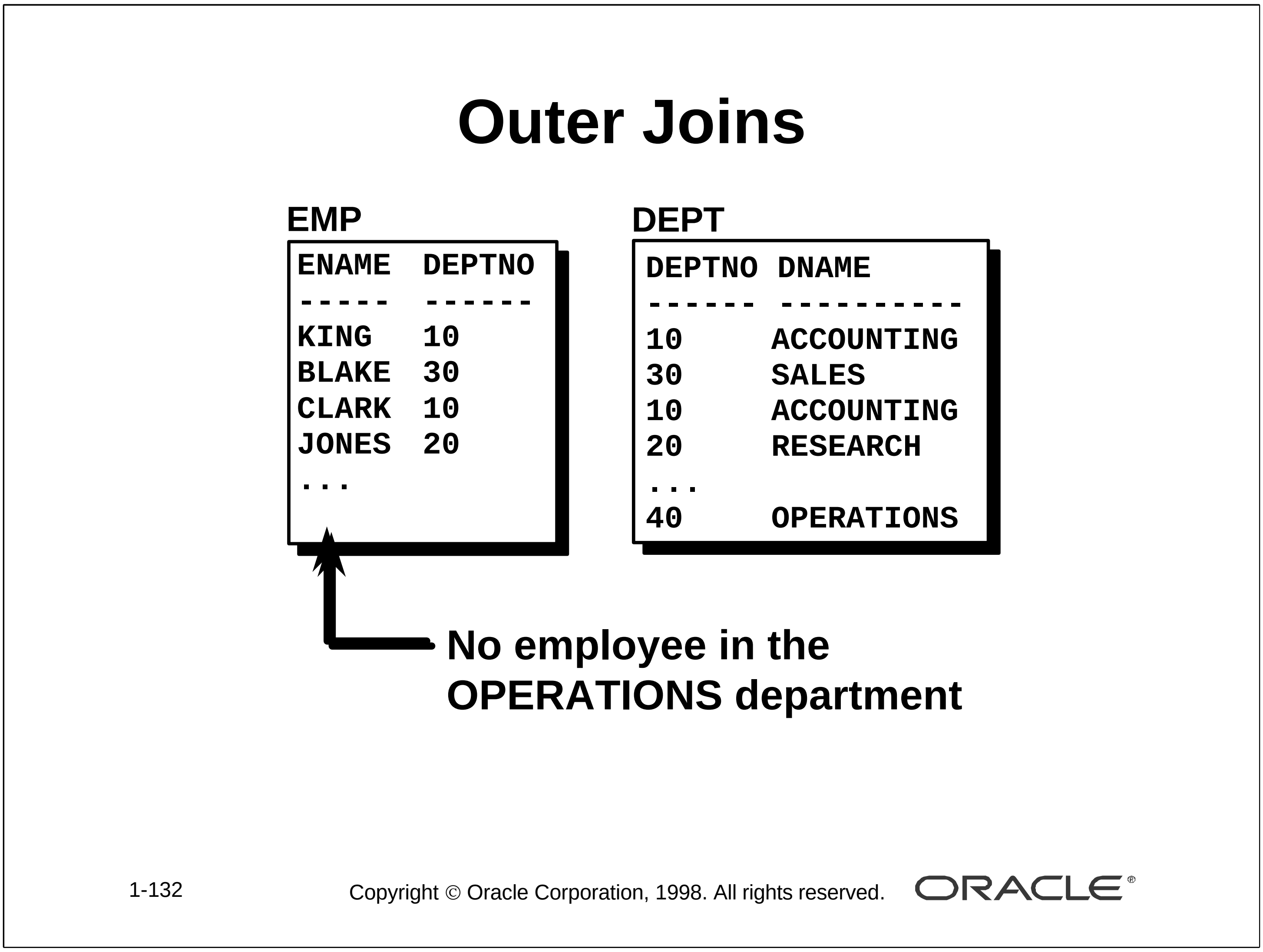
### Non-Equijoins (continued)

The example above creates a non-equijoin to evaluate an employee’s salary grade. The salary must be *between* any pair of the low and high salary ranges.

It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

- None of the rows in the salary grade table contain grades that overlap. That is, the salary value for an employee can only lie between the low salary and high salary values of one of the rows in the salary grade table.
- All of the employees’ salaries lie within the limits provided by the salary grade table. That is, no employee earns less than the lowest value contained in the LOSAL column or more than the highest value contained in the HISAL column

**Note:** Other operators such as <= and >= could be used, but BETWEEN is the simplest. Remember to specify the low value first and the high value last when using BETWEEN. Table aliases have been specified for performance reasons, not because of possible ambiguity.



# Outer Joins

EMP

```
ENAME  DEPTNO
-----  ------
KING   10
BLAKE  30
CLARK  10
JONES  20
...
```

DEPT

```
DEPTNO DNAME
------ ----------
10     ACCOUNTING
30     SALES
10     ACCOUNTING
20     RESEARCH
...
40     OPERATIONS
```

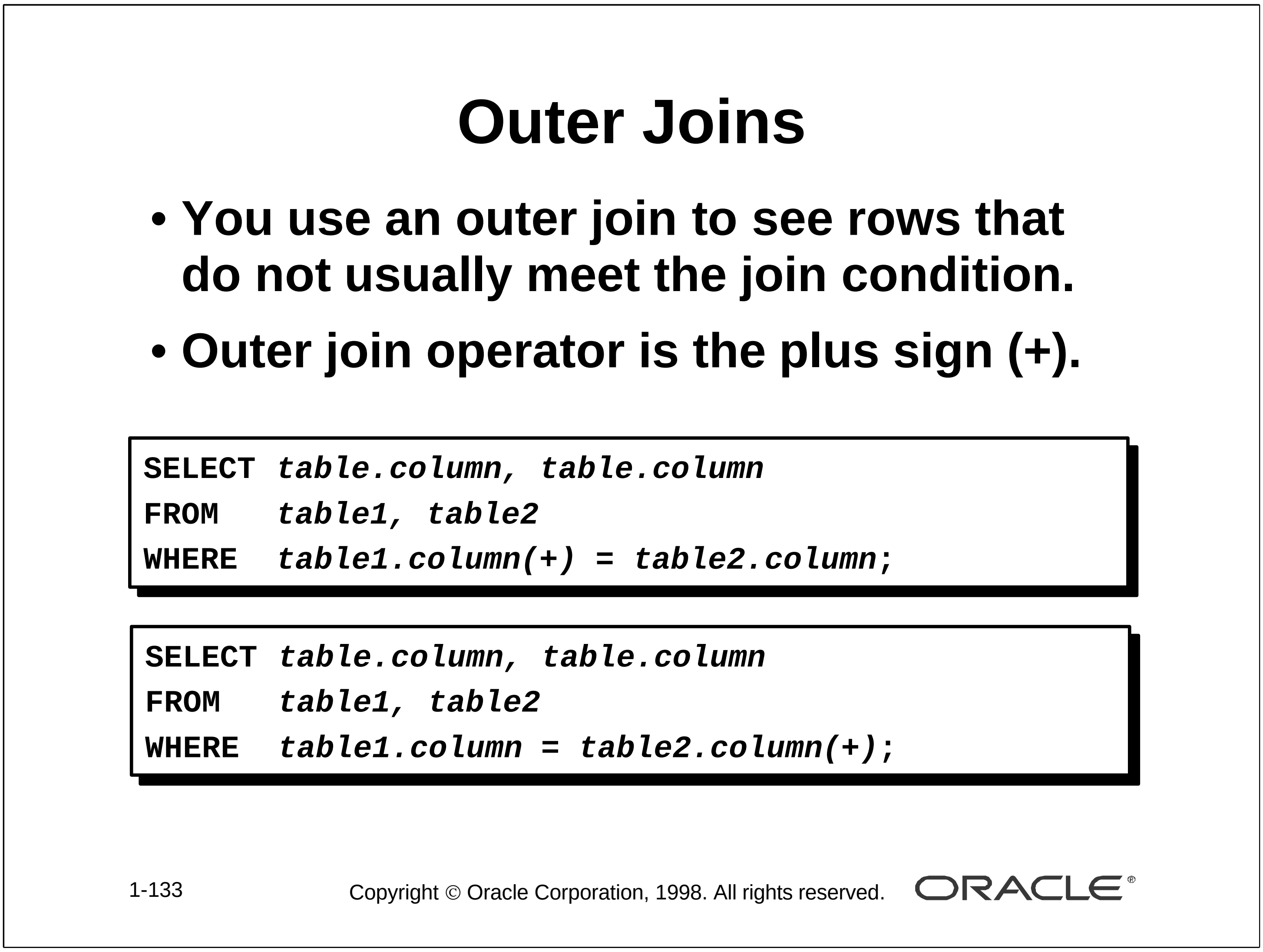
No employee in the OPERATIONS department

### Returning Records with No Direct Match with Outer Joins

If a row does not satisfy a join condition, the row will not appear in the query result. For example, in the equijoin condition of EMP and DEPT tables, department OPERATIONS does not appear because no one works in that department.

```
SQL> SELECT e.ename, e.deptno, d.dname
  2  FROM   emp e, dept d
  3  WHERE  e.deptno = d.deptno;
```

```
ENAME          DEPTNO DNAME
---------- --------- -------------
KING              10 ACCOUNTING
BLAKE             30 SALES
CLARK             10 ACCOUNTING
JONES             20 RESEARCH
...
ALLEN             30 SALES
TURNER            30 SALES
JAMES             30 SALES
...
14 rows selected.
```



# Outer Joins

- You use an outer join to see rows that do not usually meet the join condition.
- Outer join operator is the plus sign (+).

```
SELECT table.column, table.column
FROM   table1, table2
WHERE  table1.column(+) = table2.column;
```

```
SELECT table.column, table.column
FROM   table1, table2
WHERE  table1.column = table2.column(+);
```

1-133

### Returning Records with No Direct Match with Outer Joins

The missing row(s) can be returned if an *outer join* operator is used in the join condition. The operator is a plus sign enclosed in parentheses (+), and it is *placed on the “side” of the join that is deficient in information*. This operator has the effect of creating one or more null rows, to which one or more rows from the nondeficient table can be joined.

In the syntax:

*table1.column* = is the condition that joins (or relates) the tables together.

*table2.column* (+) is the outer join symbol; it can be placed on either side of the WHERE clause condition, but not on both sides. Place the outer join symbol following the name of the table without the matching rows.

### Class Management Note

Demo: *l4ejoin.sql*

# Using Outer Joins

```
SQL> SELECT   e.ename, d.deptno, d.dname
  2  FROM     emp e, dept d
  3  WHERE    e.deptno(+) = d.deptno
  4  ORDER BY e.deptno;
```

```
ENAME         DEPTNO DNAME
---------- --------- ------------
KING              10 ACCOUNTING
CLARK             10 ACCOUNTING
...
                  40 OPERATIONS
15 rows selected.
```

### Returning Records with No Direct Match with Outer Joins

The example above displays numbers and names for all the departments. The OPERATIONS department, which does not have any employees, is also displayed.

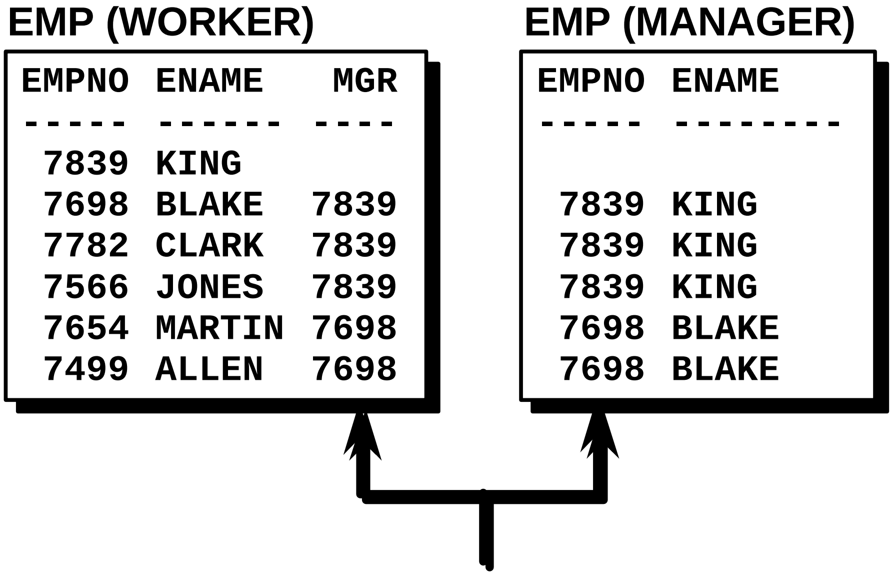
### Outer Join Restrictions

- The outer join operator can appear only on *one* side of the expression—the side that has information missing. It returns those rows from one table that have no direct match in the other table.
- A condition involving an outer join cannot use the IN operator or be linked to another condition by the OR operator.

### Class Management Note

Demo: *l4ojoin.sql*

# Self Joins



EMP (WORKER)

```
EMPNO ENAME    MGR
----- ------ ----
 7839 KING
 7698 BLAKE  7839
 7782 CLARK  7839
 7566 JONES  7839
 7654 MARTIN 7698
 7499 ALLEN  7698
```

EMP (MANAGER)

```
EMPNO ENAME
----- --------

 7839 KING
 7839 KING
 7839 KING
 7698 BLAKE
 7698 BLAKE
```

**“MGR in the WORKER table is equal to EMPNO in the MANAGER table”**

### Joining a Table to Itself

Sometimes you need to join a table to itself. To find the name of each employee’s manager you need to join the EMP table to itself. For example, to find the name of  Blake’s manager, you need to:

- Find Blake in the EMP table by looking at the ENAME column.
- Find the manager number for Blake by looking at the MGR column. Blake’s manager number is 7839.
- Find the name of the manager with EMPNO 7839 by looking at the ENAME column. King’s employee number is 7839. So, King is Blake’s manager.

In this process, you look in the table twice. The first time you look in the table to find Blake in the ENAME column and MGR value of 7839. The second time you look in the EMPNO column to find 7839 and the ENAME column to find King.

# Joining a Table to Itself

```
SQL> SELECT worker.ename||' works for '||manager.ename
  2  FROM   emp worker, emp manager
  3  WHERE  worker.mgr = manager.empno;
```

```
WORKER.ENAME||'WORKSFOR'||MANAG
-------------------------------
BLAKE works for KING
CLARK works for KING
JONES works for KING
MARTIN works for BLAKE
...
13 rows selected.
```

### Joining a Table to Itself (continued)

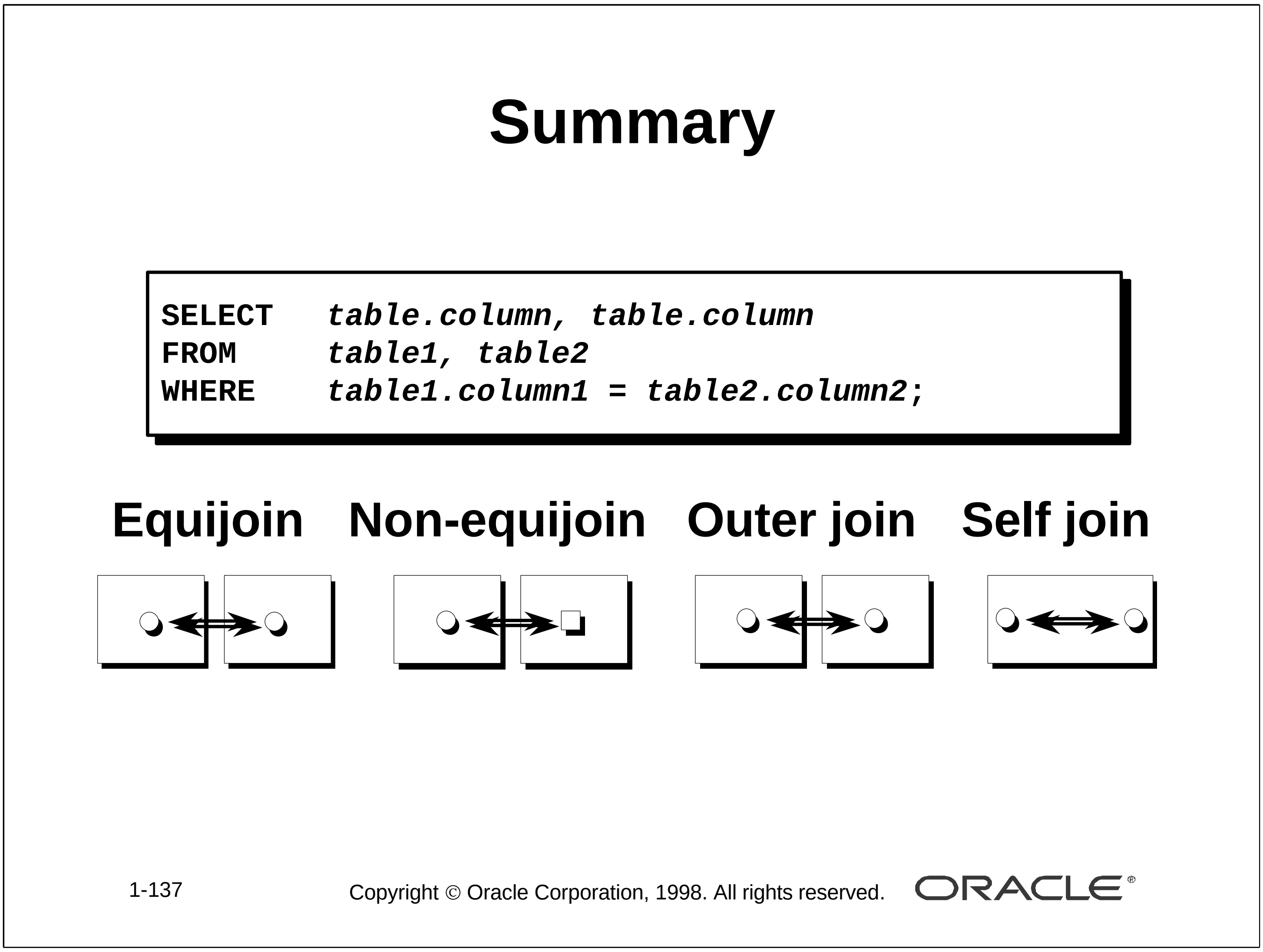
The above example joins the EMP table to itself. To simulate two tables in the FROM clause, there are two aliases, namely WORKER and MANAGER, for the same table, EMP.

In this example, the WHERE clause contains the join that means “where a worker’s manager number matches the employee number for the manager.”

### Class Management Note

Point out the following to the students:

- The column heading in the result of the above query seems meaningless. A meaningful column alias should have been used instead.
- There are only 13 rows in the output, but there are 14 rows in the EMP table. This occurs because employee King, who is the president, does not have a manager.



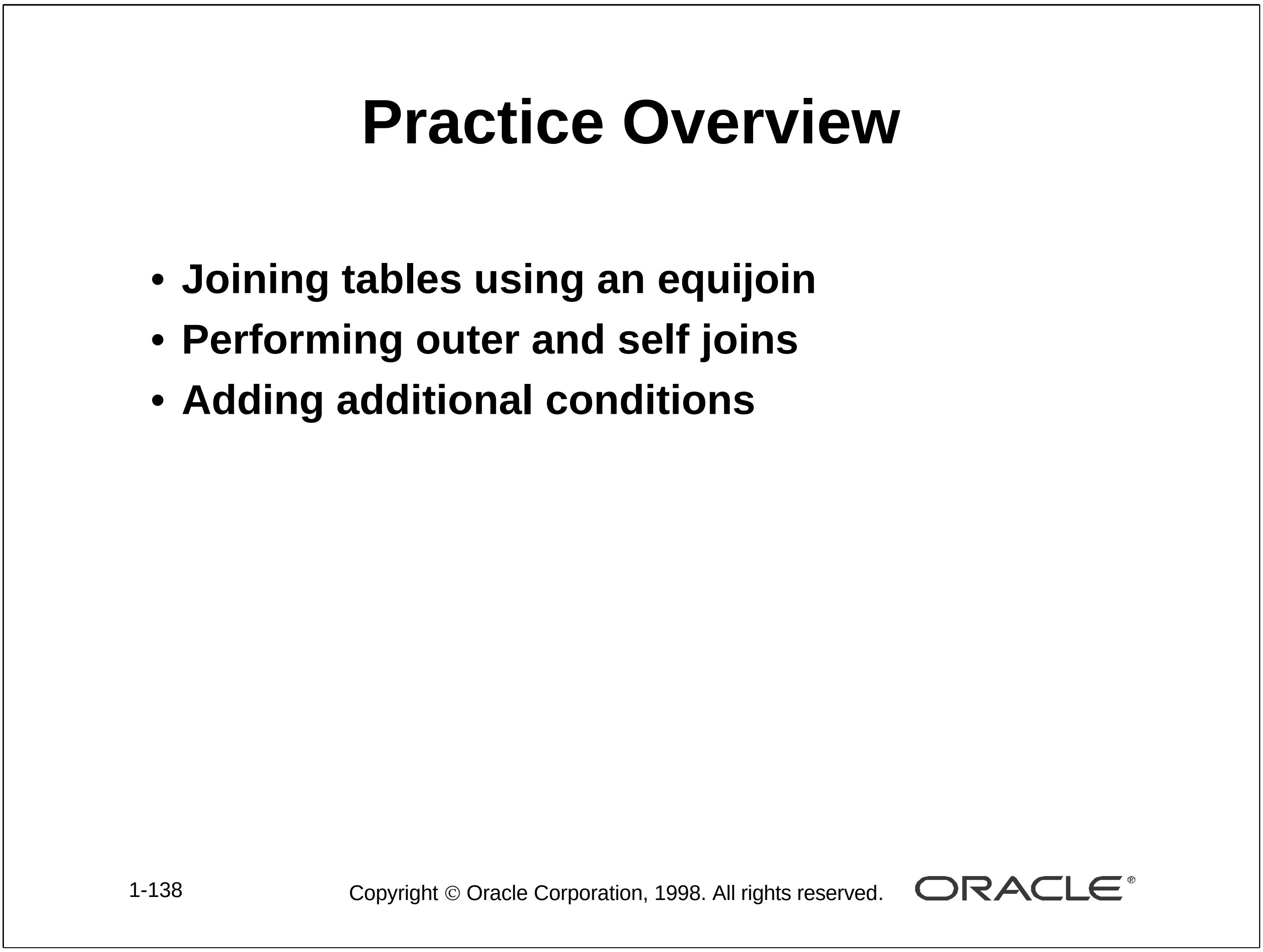
# Summary

```
SELECT   table.column, table.column
FROM     table1, table2
WHERE    table1.column1 = table2.column2;
```

### Summary

There are multiple ways to join tables. The common thread, though, is that you want to link them through a condition in the WHERE clause. The method you choose will be based on the required result and the data structures that you are using.

```
SELECT      table.column, table.column
FROM        table1, table2
WHERE       table1.column1 = table2.column2;
```

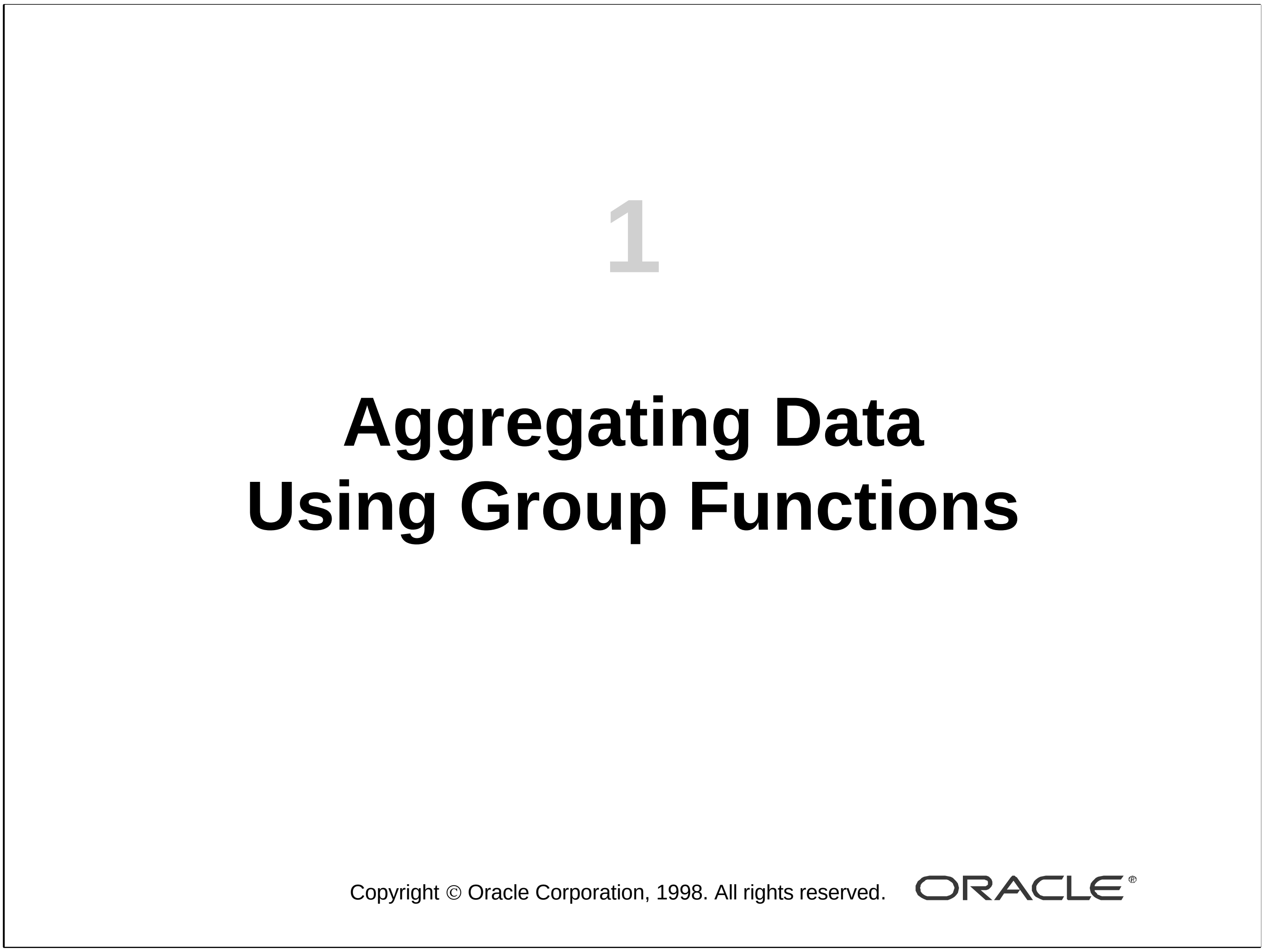


# Practice Overview

- Joining tables using an equijoin
- Performing outer and self joins
- Adding additional conditions

## Practice Overview

This practice is intended to give you practical experience in extracting data from more than one table. You will be required to join and restrict rows in the WHERE clause.



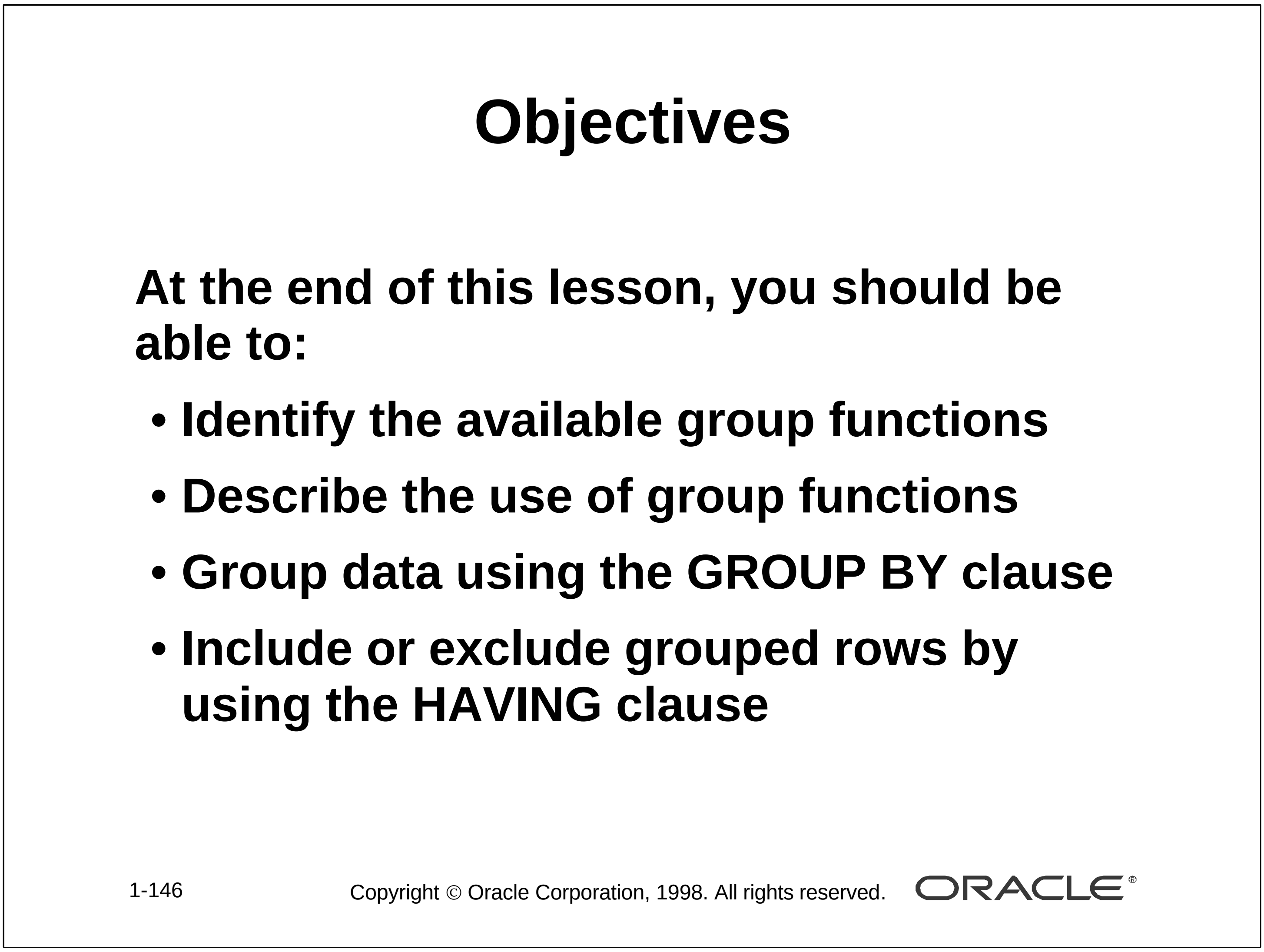
1

# Aggregating Data Using Group Functions

ORACLE®

**Schedule:**

| Timing | Topic |
|---|---|
| 35 minutes | Lecture |
| 40 minutes | Practice |
| 75 minutes | Total |



# Objectives

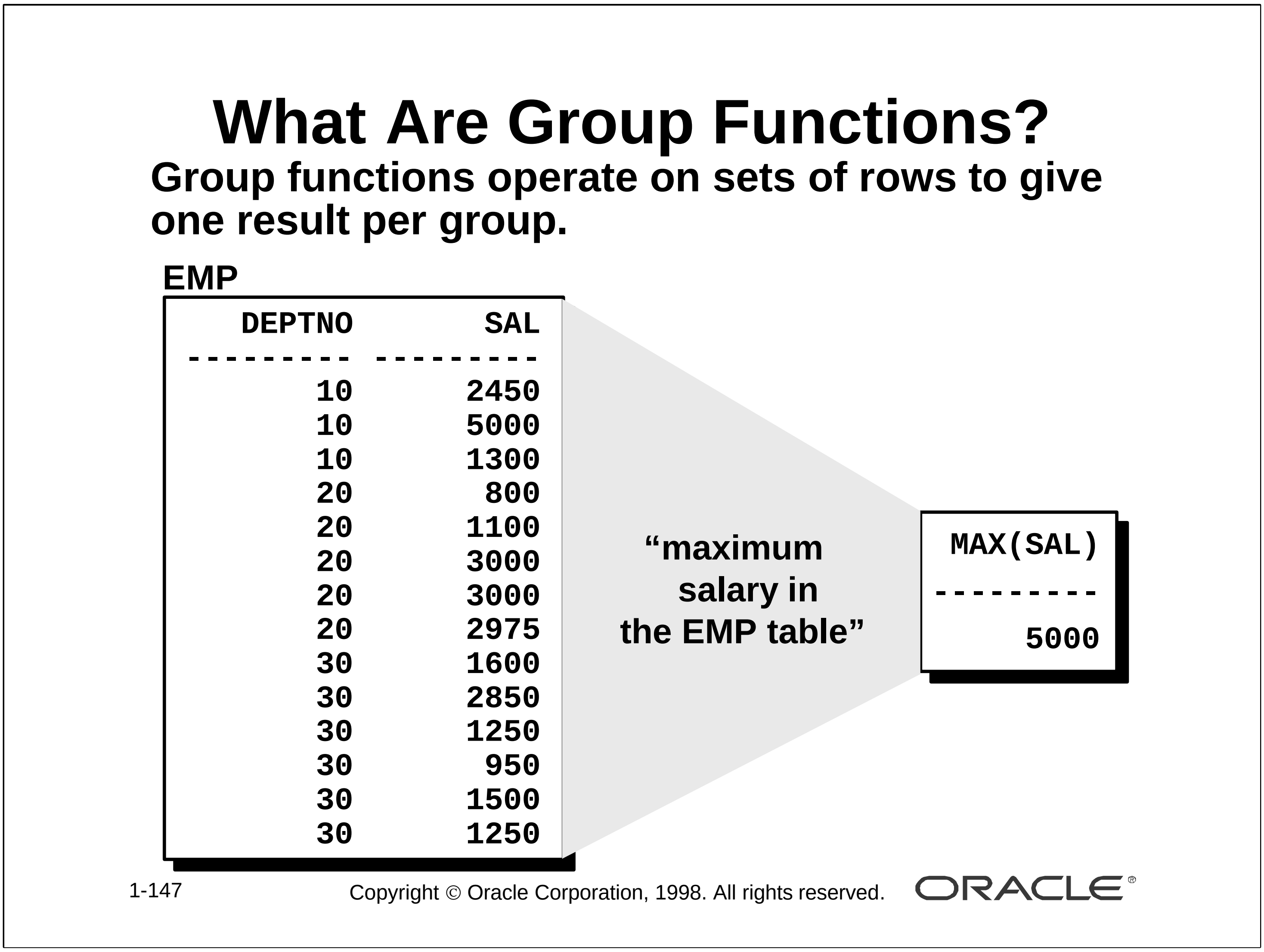
At the end of this lesson, you should be able to:

- Identify the available group functions
- Describe the use of group functions
- Group data using the GROUP BY clause
- Include or exclude grouped rows by using the HAVING clause

1-146

**Lesson Aim**

This lesson further addresses functions. It focuses on obtaining summary information, such as averages, for groups of rows. It discusses how to group rows in a table into smaller sets and how to specify search criteria for groups of rows.



# What Are Group Functions?

## Group functions operate on sets of rows to give one result per group.

EMP

```
   DEPTNO       SAL
--------- ---------
       10      2450
       10      5000
       10      1300
       20       800
       20      1100
       20      3000
       20      3000
       20      2975
       30      1600
       30      2850
       30      1250
       30       950
       30      1500
       30      1250
```

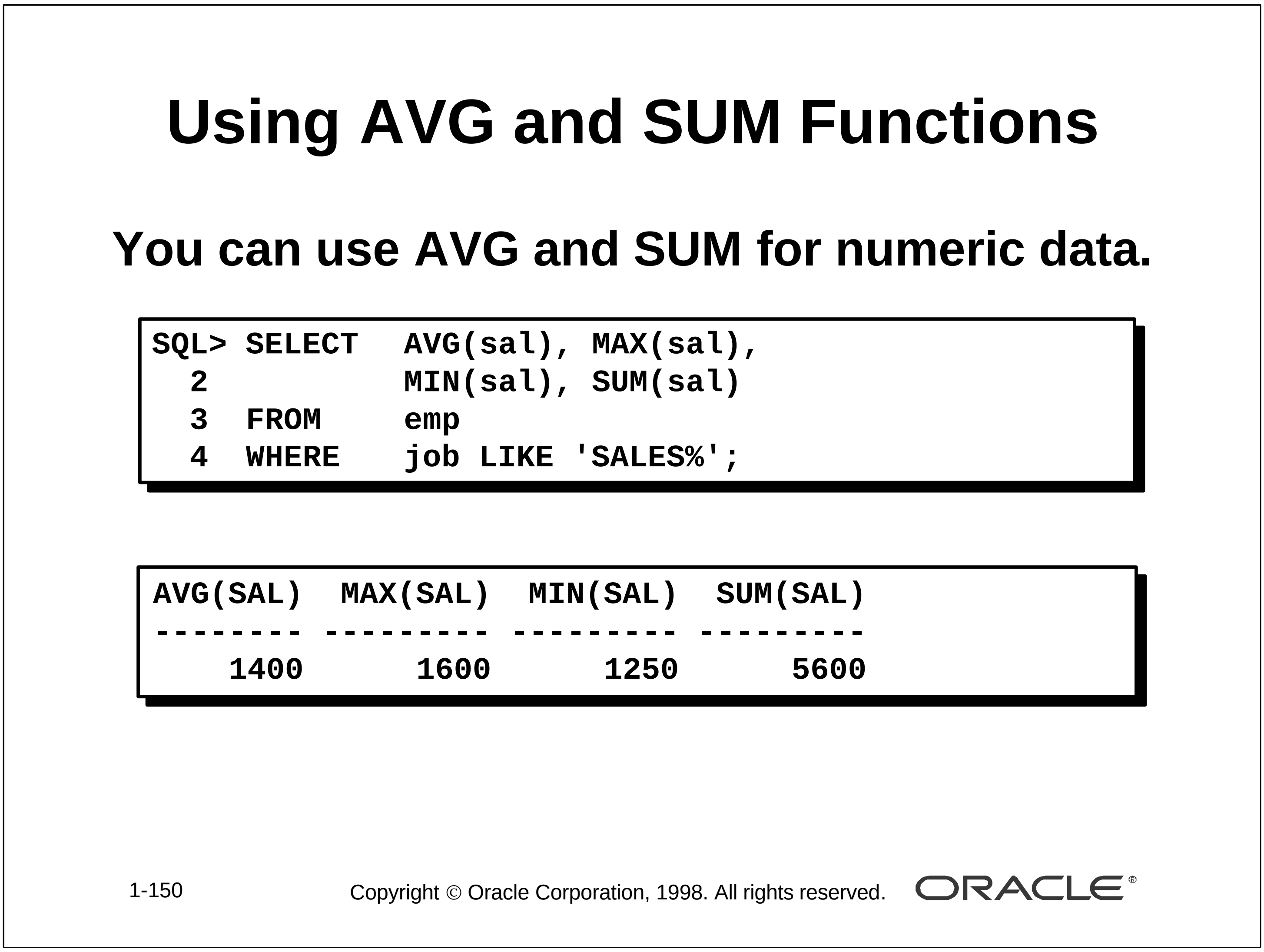
“maximum salary in the EMP table”

```
 MAX(SAL)
---------
     5000
```

1-147

### Group Functions

Unlike single-row functions, group functions operate on sets of rows to give one result per group. These sets may be the whole table or the table split into groups.



# Using AVG and SUM Functions

## You can use AVG and SUM for numeric data.

```
SQL> SELECT   AVG(sal), MAX(sal),
  2           MIN(sal), SUM(sal)
  3  FROM     emp
  4  WHERE    job LIKE 'SALES%';
```

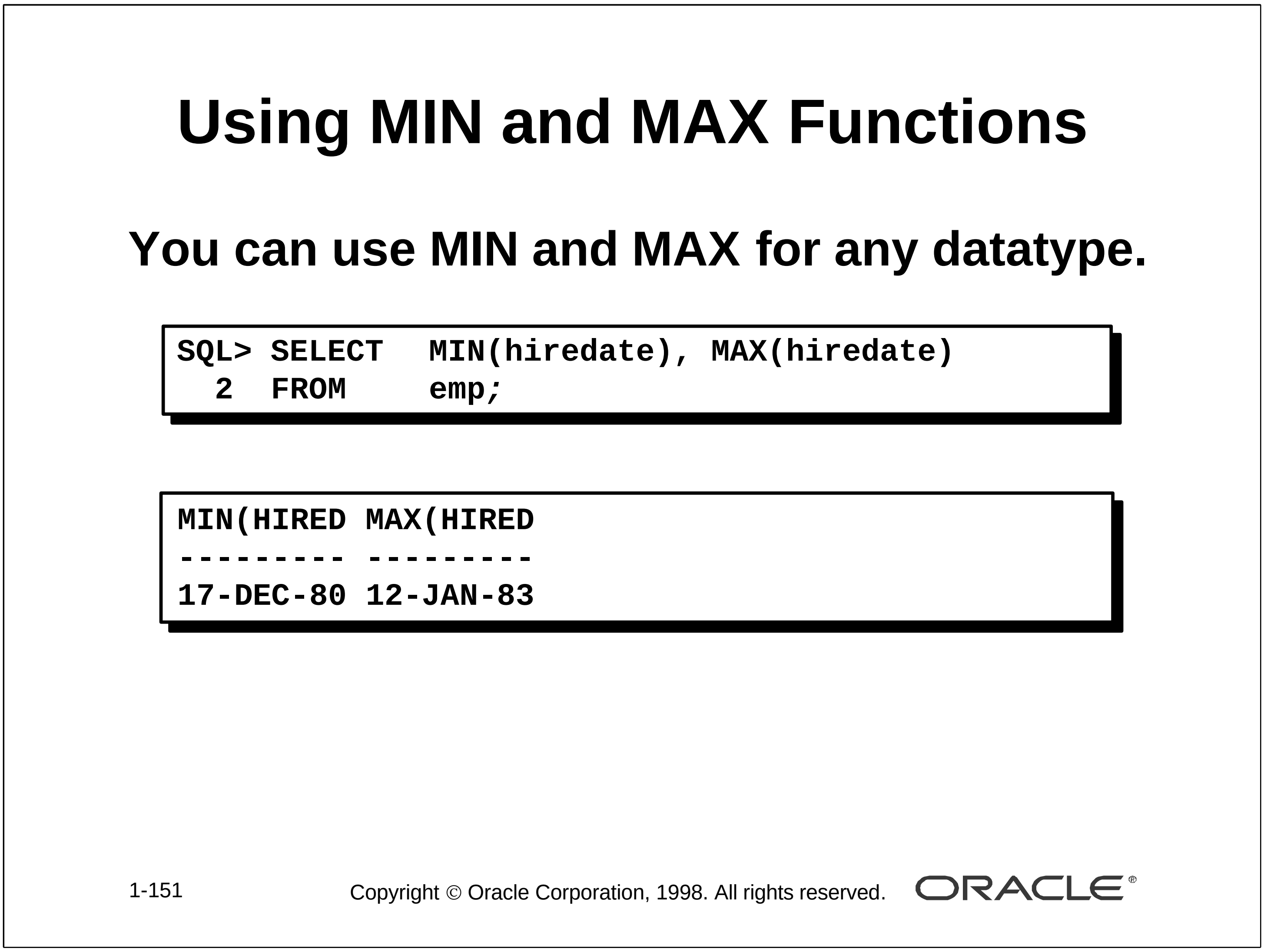
```
AVG(SAL)  MAX(SAL)  MIN(SAL)  SUM(SAL)
-------- --------- --------- ---------
    1400      1600      1250      5600
```

1-150

ORACLE®

### Group Functions

You can use AVG, SUM, MIN, and MAX  functions against columns that can store numeric data. The example above displays the average, highest, lowest, and sum of monthly salaries for all salesmen.



# Using MIN and MAX Functions

## You can use MIN and MAX for any datatype.

```
SQL> SELECT   MIN(hiredate), MAX(hiredate)
  2  FROM     emp;
```

```
MIN(HIRED MAX(HIRED
--------- ---------
17-DEC-80 12-JAN-83
```

1-151

### Group Functions (continued)

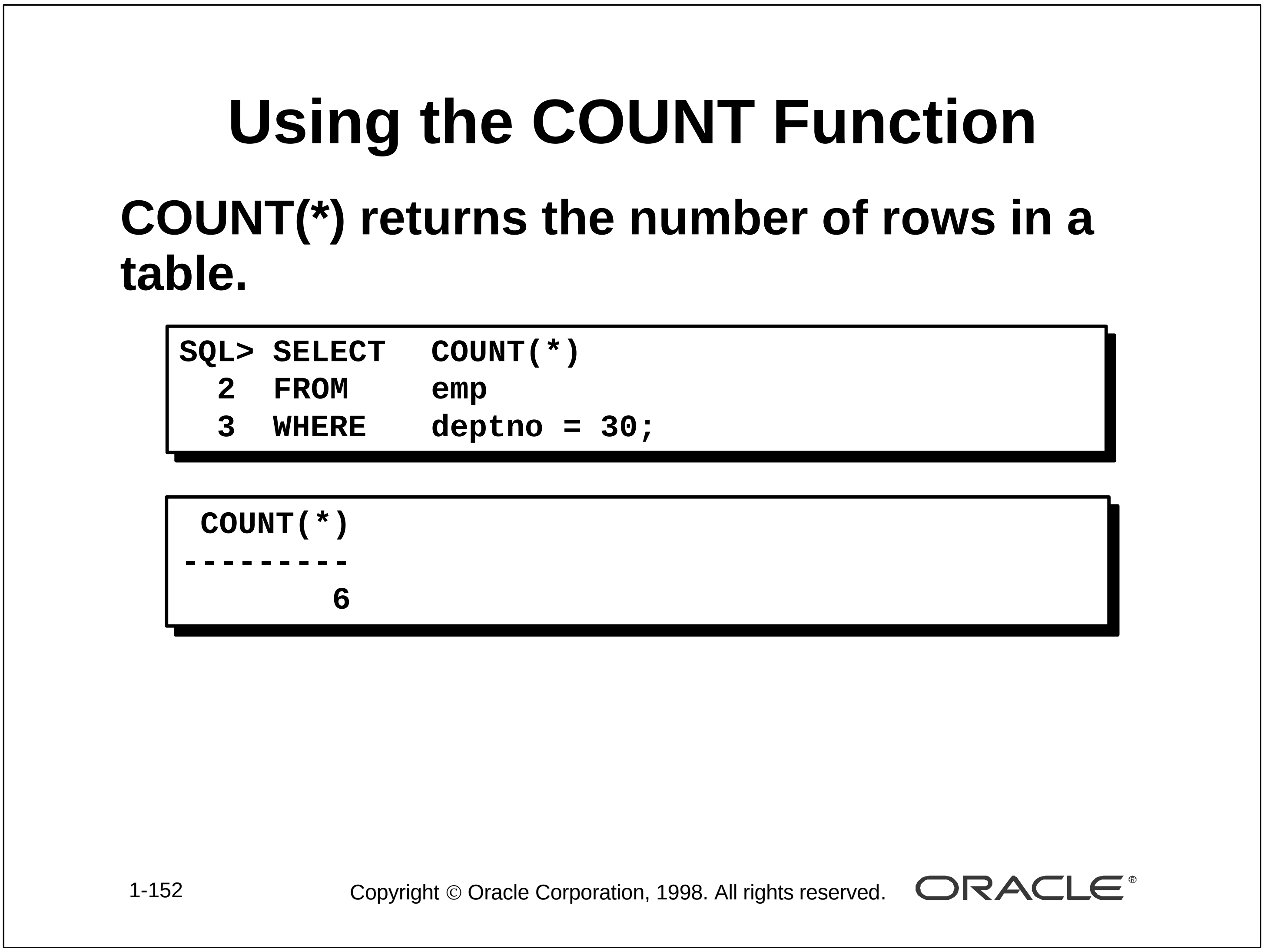
You can use MAX and MIN functions for any datatype. The example above displays the most junior and most senior employee.

The example below displays the employee name that is first and the employee name that is the last in an alphabetized list of all employees.

```
SQL> SELECT     MIN(ename), MAX(ename)
  2  FROM       emp;
```

```
MIN(ENAME) MAX(ENAME)
---------- ----------
ADAMS      WARD
```

**Note**: AVG and SUM functions can be used only with numeric datatypes.



# Using the COUNT Function

## COUNT(*) returns the number of rows in a table.

```
SQL> SELECT   COUNT(*)
  2  FROM     emp
  3  WHERE    deptno = 30;
```

```
 COUNT(*)
---------
        6
```

1-152

ORACLE®

### The COUNT Function

The COUNT function has two formats:

- COUNT(*)
- COUNT(*expr*).

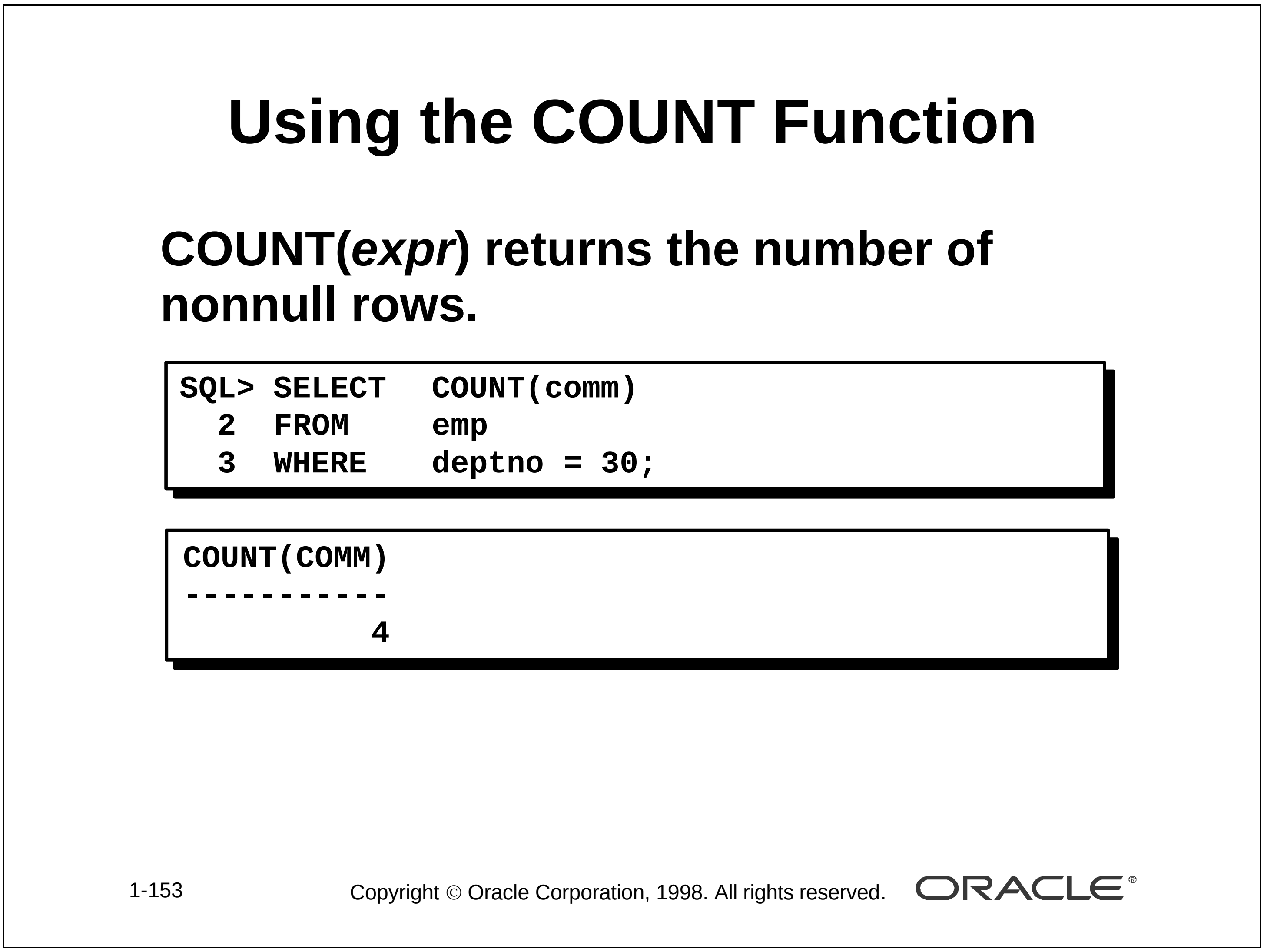
COUNT(*) returns the number of rows in a table, including duplicate rows and rows containing null values.

In contrast, COUNT(*expr*) returns the number of nonnull rows in the column identified by *expr*.

The example above displays the number of employees in department 30.

### Class Management Note

Demo: *l5count1.sql*, *l5count2.sql*



# Using the COUNT Function

## COUNT(*expr*) returns the number of nonnull rows.

```
SQL> SELECT   COUNT(comm)
  2  FROM     emp
  3  WHERE    deptno = 30;
```

```
COUNT(COMM)
-----------
          4
```

1-153

### The COUNT Function (continued)

The example above displays the number of employees in department 30 who can earn a commission. Notice that the result gives the total number of rows to be four because two employees in department 30 cannot earn a commission and contain a null value in the COMM column.

### Example

Display the number of departments in the EMP table.

```
SQL> SELECT    COUNT(deptno)
  2  FROM      emp;
```

```
COUNT(DEPTNO)
-------------
           14
```

Display the number of distinct departments in the EMP table.

```
SQL> SELECT    COUNT(DISTINCT (deptno))
  2  FROM      emp;
```

```
COUNT(DISTINCT(DEPTNO))
-----------------------
                      3
```

# Group Functions and Null Values

## Group functions ignore null values in the column.

```
SQL> SELECT AVG(comm)
  2  FROM   emp;
```

```
 AVG(COMM)
---------
      550
```

### Group Functions and Null Values

All group functions except COUNT (*) ignore null values in the column. In the above example, the average is calculated based *only* on the rows in the table where a valid value is stored in the COMM column. The average is calculated as total commission being paid to all employees divided by the number of employees receiving commission (4).

# Using the NVL Function with Group Functions

## The NVL function forces group functions to include null values.

```
SQL> SELECT AVG(NVL(comm,0))
  2  FROM   emp;
```

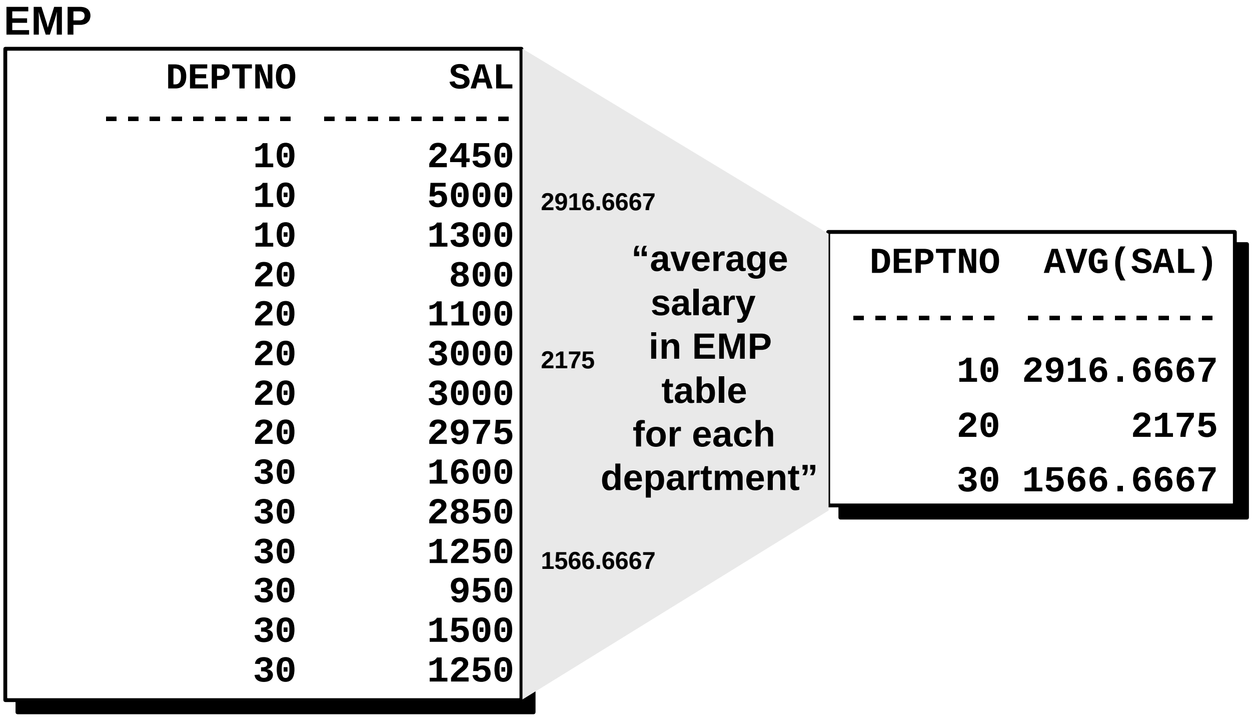
```
AVG(NVL(COMM,0))
----------------
       157.14286
```

1-155

### Group Functions and Null Values (continued)

The NVL function forces group functions to include null values. In the above example, the average is calculated based on *all* rows in the table regardless of whether null values are stored in the COMM column. The average is calculated as total commission being paid to all employees divided by the total number of employees in the company (14).

# Creating Groups of Data



EMP

```
   DEPTNO       SAL
--------- ---------
       10      2450
       10      5000
       10      1300
       20       800
       20      1100
       20      3000
       20      3000
       20      2975
       30      1600
       30      2850
       30      1250
       30       950
       30      1500
       30      1250
```

2916.6667

2175

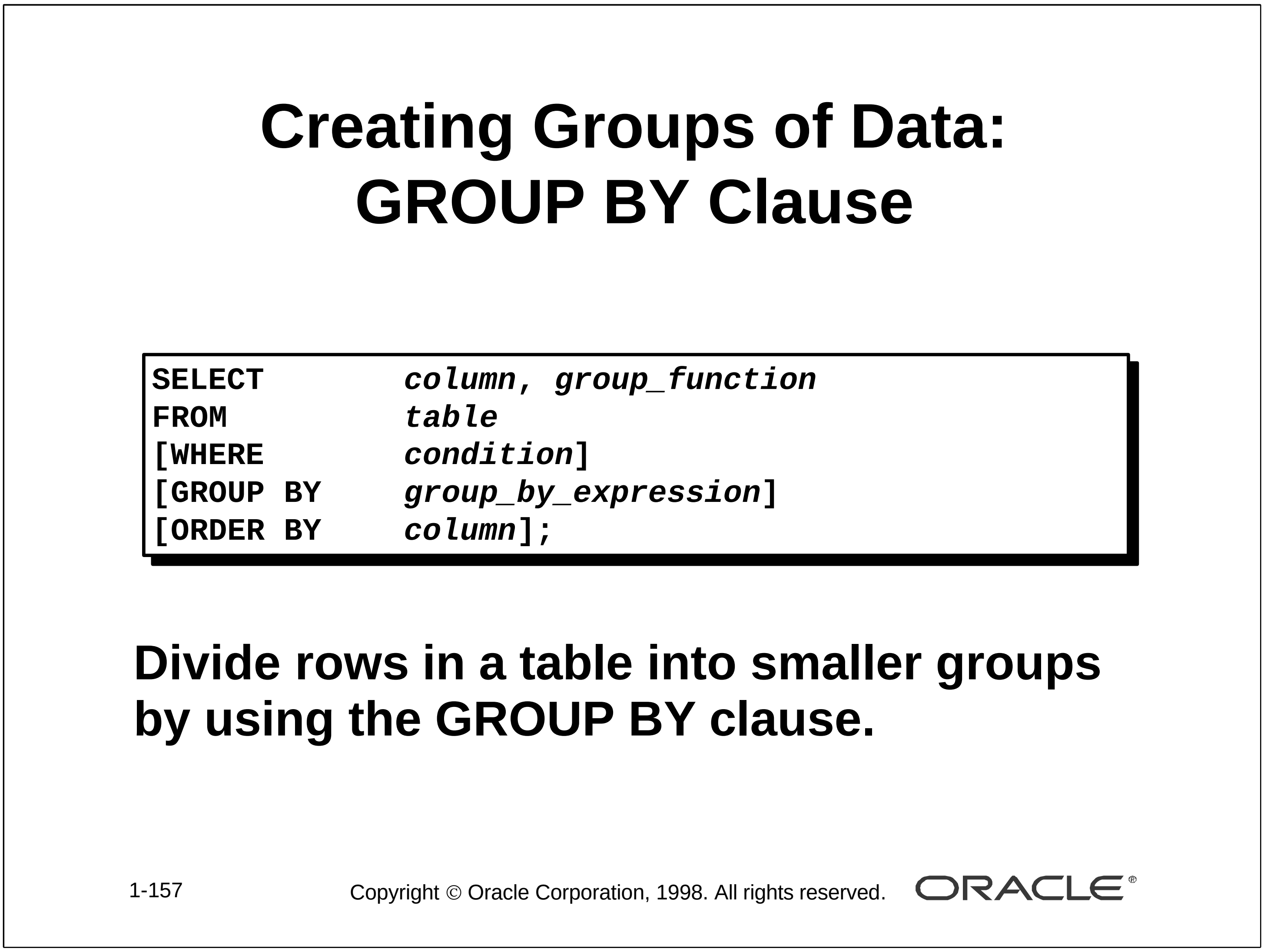
1566.6667

"average salary in EMP table for each department"

```
 DEPTNO  AVG(SAL)
------- ---------

     10 2916.6667

     20      2175

     30 1566.6667
```

### Groups of Data

Until now, all group functions have treated the table as one large group of information. At times, you need to divide the table of information into smaller groups. This can be done by using the GROUP BY clause.



# Creating Groups of Data: GROUP BY Clause

```
SELECT        column, group_function
FROM          table
[WHERE        condition]
[GROUP BY     group_by_expression]
[ORDER BY     column];
```

**Divide rows in a table into smaller groups by using the GROUP BY clause.**

1-157

ORACLE®

### The GROUP BY Clause

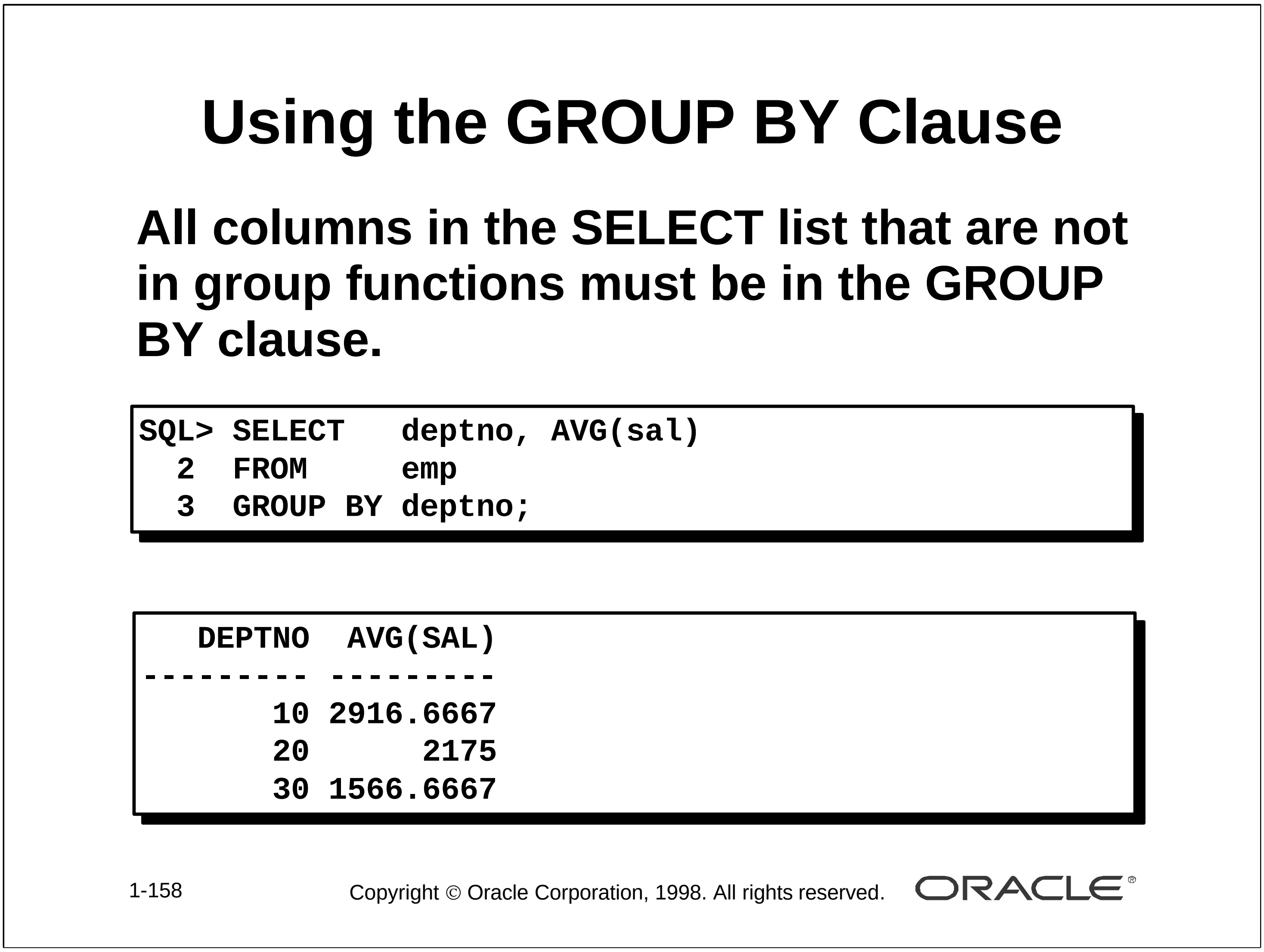
You can use the GROUP BY clause to divide the rows in a table into groups. You can then use the group functions to return summary information for each group.

In the syntax:

| | |
|---|---|
| *group_by_expression* | specifies columns whose values determine the basis for grouping rows |

### Guidelines

- If you include a group function in a SELECT clause, you cannot select individual results as well *unless* the individual column appears in the GROUP BY clause. You will receive an error message if you fail to include the column list.
- Using a WHERE clause, you can preexclude rows before dividing them into groups.
- You must include the *columns* in the GROUP BY clause.
- You cannot use the column alias in the GROUP BY clause.
- By default, rows are sorted by ascending order of the columns included in the GROUP BY list. You can override this by using the ORDER BY clause.



# Using the GROUP BY Clause

## All columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

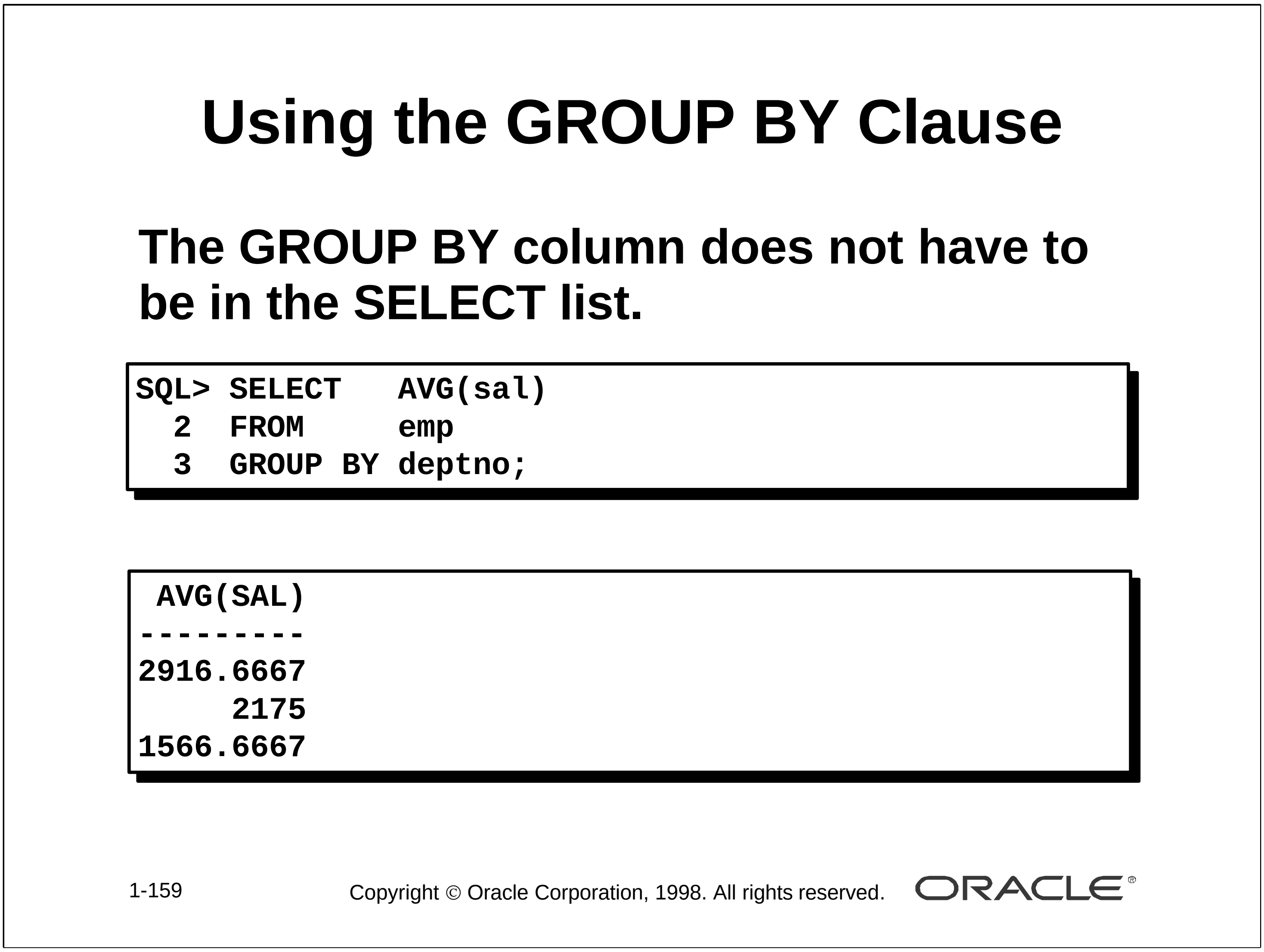
```
SQL> SELECT   deptno, AVG(sal)
  2  FROM     emp
  3  GROUP BY deptno;
```

```
   DEPTNO  AVG(SAL)
--------- ---------
       10 2916.6667
       20      2175
       30 1566.6667
```

### The GROUP BY Clause (continued)

When using the GROUP BY clause, make sure that all columns in the SELECT list that are not in the group functions are included in the GROUP BY clause. The above example displays the department number and the average salary for each department. Here is how the SELECT statement above, containing a GROUP BY clause, is evaluated:

- The SELECT clause specifies the columns to be retrieved:
  - Department number column in the EMP table
  - The average of all the salaries in the group you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMP table.
- The WHERE clause specifies the rows to be retrieved. Since there is no WHERE clause, by default all rows are retrieved.
- The GROUP BY clause specifies how the rows should be grouped. The rows are being grouped by department number, so the AVG function that is being applied to the salary column will calculate the *average salary for each department.*



# Using the GROUP BY Clause

## The GROUP BY column does not have to be in the SELECT list.

```
SQL> SELECT   AVG(sal)
  2  FROM     emp
  3  GROUP BY deptno;
```

```
 AVG(SAL)
---------
2916.6667
     2175
1566.6667
```

1-159

### The GROUP BY Clause (continued)

The GROUP BY column does not have to be in the SELECT clause. For example, the above SELECT statement displays the average salaries for each department without displaying the respective department numbers. However, without the department numbers, the results do not look meaningful.

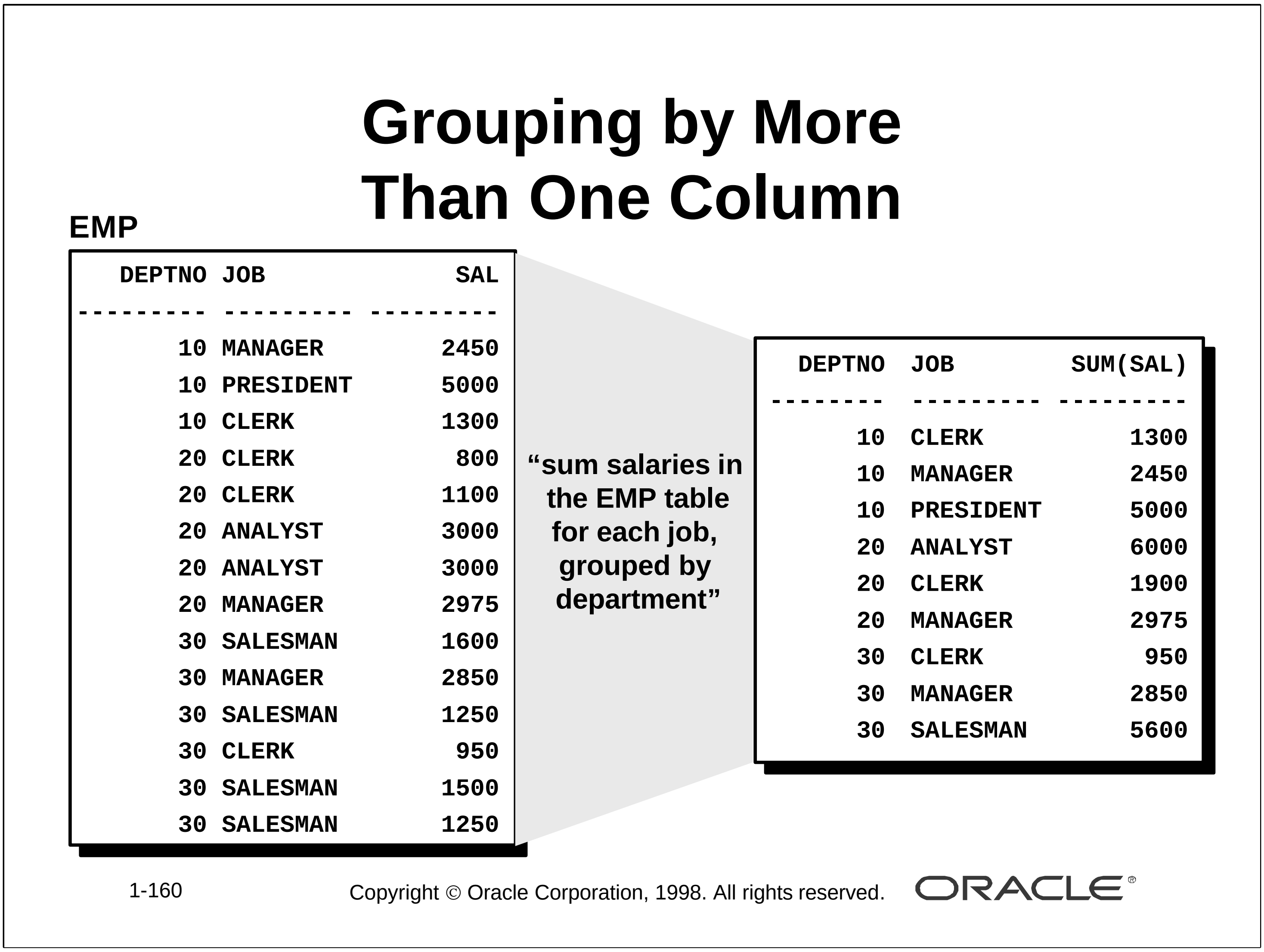
You can use the group function in the ORDER BY clause.

```
SQL> SELECT     deptno, AVG(sal)
  2  FROM       emp
  3  GROUP BY   deptno
  4  ORDER BY    AVG(sal);
```

```
      DEPTNO     AVG(SAL)
  ---------- ------------
          30    1566.6667
          20         2175
          10    2916.6667
```

**Class Management Note**

Demonstrate the query with and without the DEPTNO in the SELECT statement.



# Grouping by More Than One Column

EMP

```
   DEPTNO JOB             SAL
--------- --------- ---------
       10 MANAGER        2450
       10 PRESIDENT      5000
       10 CLERK          1300
       20 CLERK           800
       20 CLERK          1100
       20 ANALYST        3000
       20 ANALYST        3000
       20 MANAGER        2975
       30 SALESMAN       1600
       30 MANAGER        2850
       30 SALESMAN       1250
       30 CLERK           950
       30 SALESMAN       1500
       30 SALESMAN       1250
```

“sum salaries in the EMP table for each job, grouped by department”

```
  DEPTNO  JOB        SUM(SAL)
--------  --------- ---------
      10  CLERK          1300
      10  MANAGER        2450
      10  PRESIDENT      5000
      20  ANALYST        6000
      20  CLERK          1900
      20  MANAGER        2975
      30  CLERK           950
      30  MANAGER        2850
      30  SALESMAN       5600
```

1-160

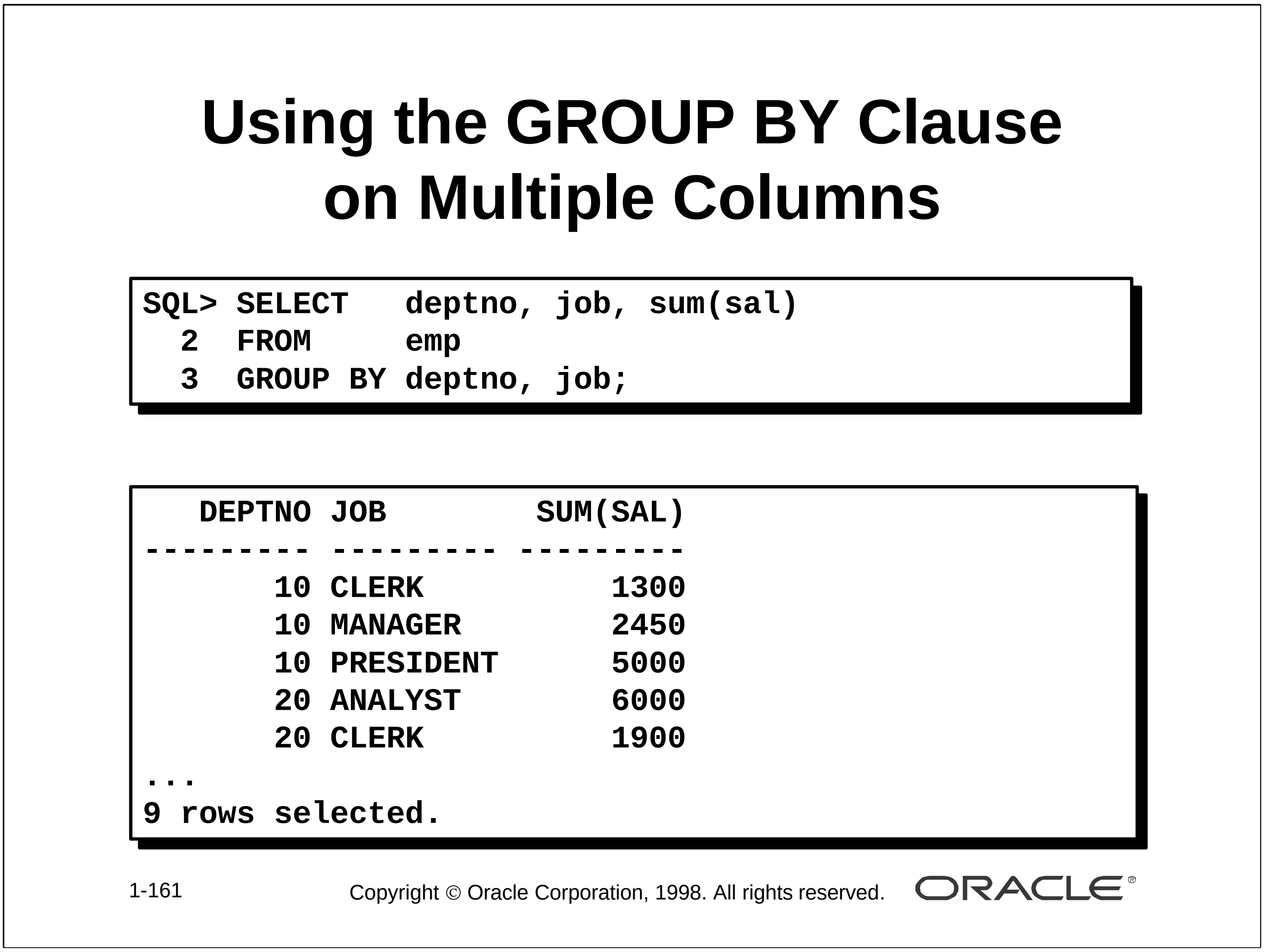
### Groups Within Groups

Sometimes there is a need to see results for groups within groups. The slide above shows a report that displays the total salary being paid to each job title, within each department.

The EMP table is grouped first by department number and then within that grouping it is grouped by job title. For example, the two clerks in department 20 are grouped together and a single result (total salary) is produced for all salesmen within the group.

### Class Management Note

Demo: *l5order1.sql,l5order2.sql*



# Using the GROUP BY Clause on Multiple Columns

```
SQL> SELECT   deptno, job, sum(sal)
  2  FROM     emp
  3  GROUP BY deptno, job;
```

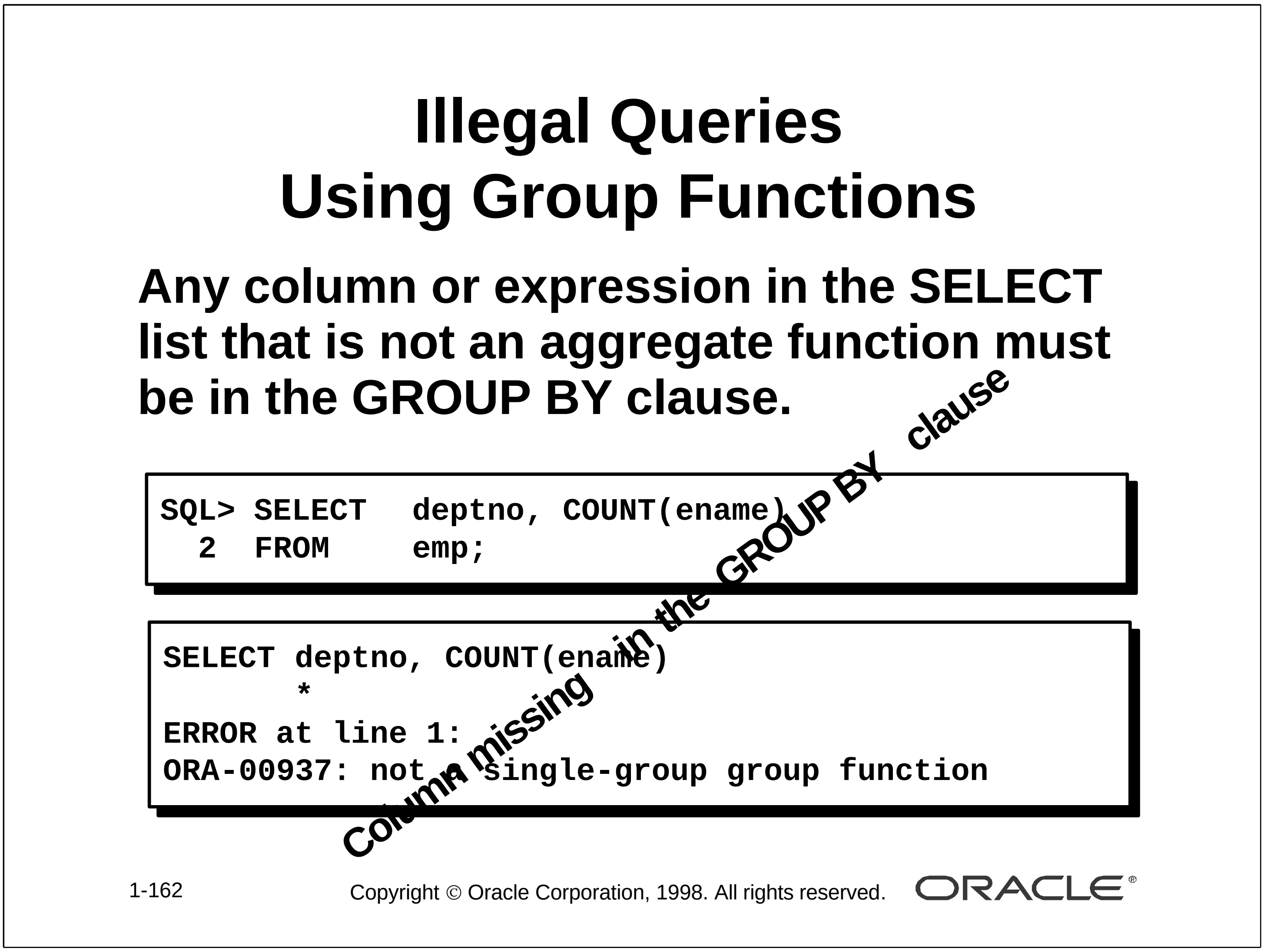
```
   DEPTNO JOB        SUM(SAL)
--------- --------- ---------
       10 CLERK          1300
       10 MANAGER        2450
       10 PRESIDENT      5000
       20 ANALYST        6000
       20 CLERK          1900
...
9 rows selected.
```

### Groups Within Groups (continued)

You can return summary results for groups and subgroups by listing more than one GROUP BY column. You can determine the default sort order of the results by the order of the columns in the GROUP BY clause. Here is how the SELECT statement above, containing a GROUP BY clause, is evaluated:

- The SELECT clause specifies the column to be retrieved:
  - Department number in the EMP table
  - Job title in the EMP table
  - The sum of all the salaries in the group you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMP table.
- The GROUP BY clause specifies how you must group the rows:
  - First, the rows are grouped by department number.
  - Second, within the department number groups, the rows are grouped by job title.

So the SUM function is being applied to the salary column for all job titles within each department number group.



# Illegal Queries Using Group Functions

**Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause.**

```
SQL> SELECT   deptno, COUNT(ename)
  2  FROM     emp;
```

```
SELECT deptno, COUNT(ename)
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

Column missing in the GROUP BY clause

1-162

ORACLE®

### Illegal Queries Using Group Functions

Whenever you use a mixture of individual items (DEPTNO) and group functions (COUNT) in the same SELECT statement, you must include a GROUP BY clause that specifies the individual items (in this case, DEPTNO). If the GROUP BY clause is missing, then the error message “not a single-group group function” appears and an asterisk (*) points to the offending column. You can correct the above error by adding the GROUP BY clause.

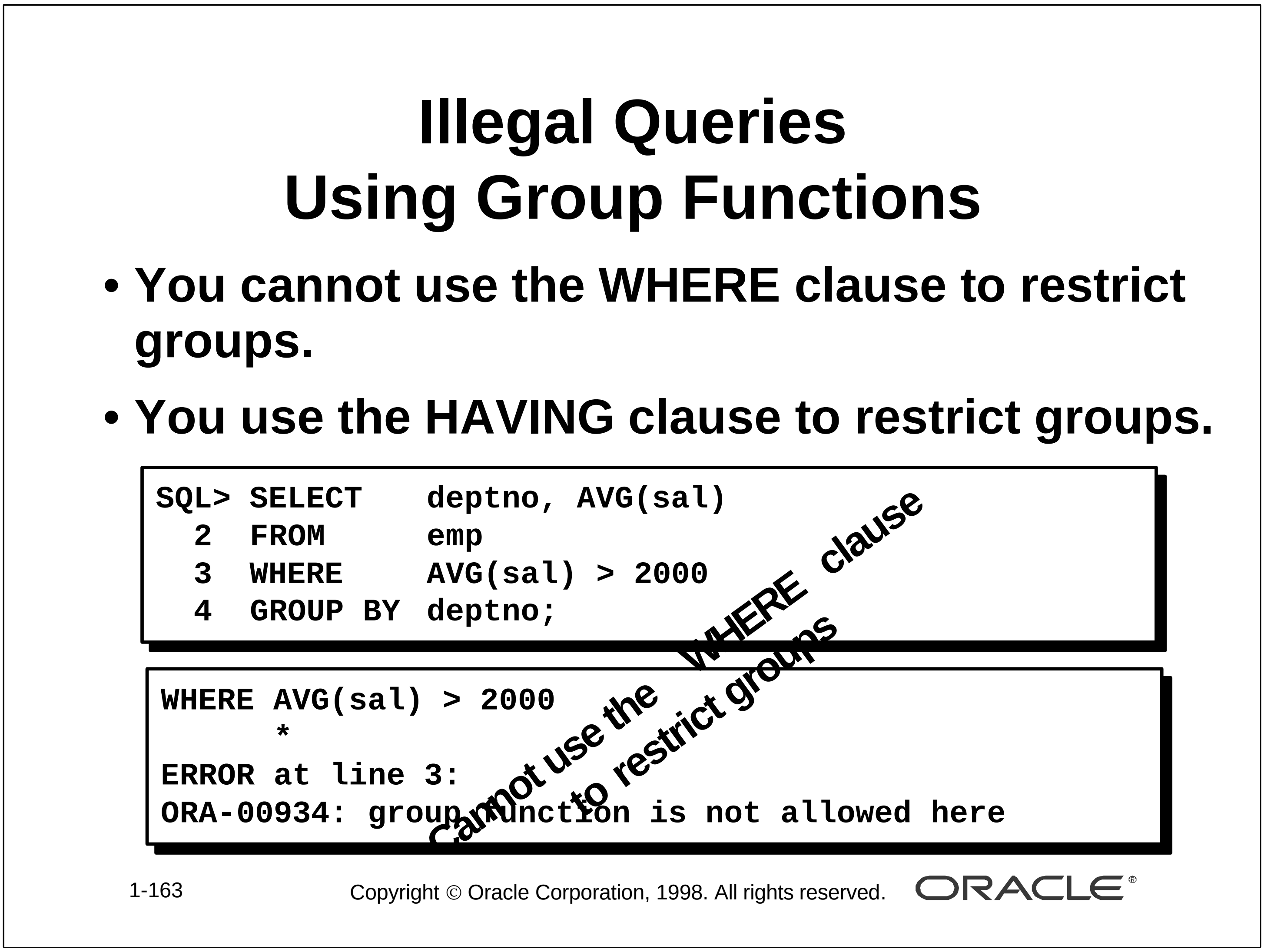
```
SQL> SELECT    deptno, COUNT(ename)
  2  FROM      emp
  3  GROUP BY  deptno;
```

```
    DEPTNO COUNT(ENAME)
---------- ------------
        10          3
        20          5
        30          6
```

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause.

**Class Management Note**

Demo: *l5error.sql*



# Illegal Queries Using Group Functions

- You cannot use the WHERE clause to restrict groups.
- You use the HAVING clause to restrict groups.

```
SQL> SELECT    deptno, AVG(sal)
  2  FROM      emp
  3  WHERE     AVG(sal) > 2000
  4  GROUP BY  deptno;
```

```
WHERE AVG(sal) > 2000
          *
ERROR at line 3:
ORA-00934: group function is not allowed here
```

Cannot use the WHERE clause to restrict groups

## Illegal Queries Using Group Functions (continued)

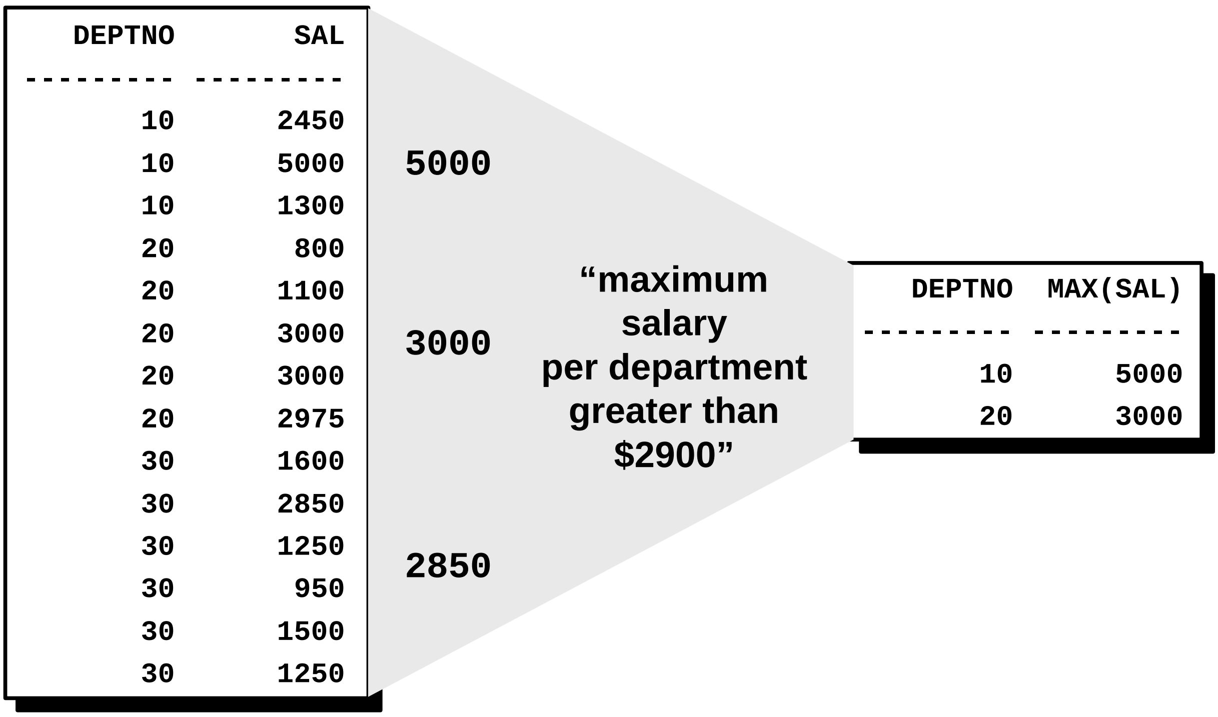
The WHERE clause cannot be used to restrict groups. The above SELECT statement results in an error because it uses the WHERE clause to restrict the display of average salaries of those departments that have an average salary of greater than $2000.

You can correct the above error by using the HAVING clause to restrict groups.

```
SQL> SELECT    deptno, AVG(sal)
  2  FROM      emp
  3  GROUP BY  deptno
  4  HAVING    AVG(sal) > 2000;
```

```
    DEPTNO       AVG(SAL)
---------- --------------
        10      2916.6667
        20           2175
```

# Excluding Group Results



EMP

```
   DEPTNO       SAL
--------- ---------
       10      2450
       10      5000
       10      1300
       20       800
       20      1100
       20      3000
       20      3000
       20      2975
       30      1600
       30      2850
       30      1250
       30       950
       30      1500
       30      1250
```

“maximum salary per department greater than $2900”

```
   DEPTNO  MAX(SAL)
--------- ---------
       10      5000
       20      3000
```

### Restricting Group Results

In the same way that you use the WHERE clause to restrict the rows that you select, you use the HAVING clause to restrict groups. To find the maximum salary of each department, but show only the departments that have a maximum salary of more than $2900, you need to do the following two things:

1. Find the average salary for each department by grouping by department number.
2. Restrict the groups to those departments with a maximum salary greater than $2900.

# Excluding Group Results: HAVING Clause

## Use the HAVING clause to restrict groups

– Rows are grouped.

– The group function is applied.

– Groups matching the HAVING clause are displayed.

```
SELECT        column, group_function
FROM          table
[WHERE        condition]
[GROUP BY     group_by_expression]
[HAVING       group_condition]
[ORDER BY     column];
```

### The HAVING Clause

You use the HAVING clause to specify which groups are to be displayed. Therefore, you further restrict the groups on the basis of aggregate information.

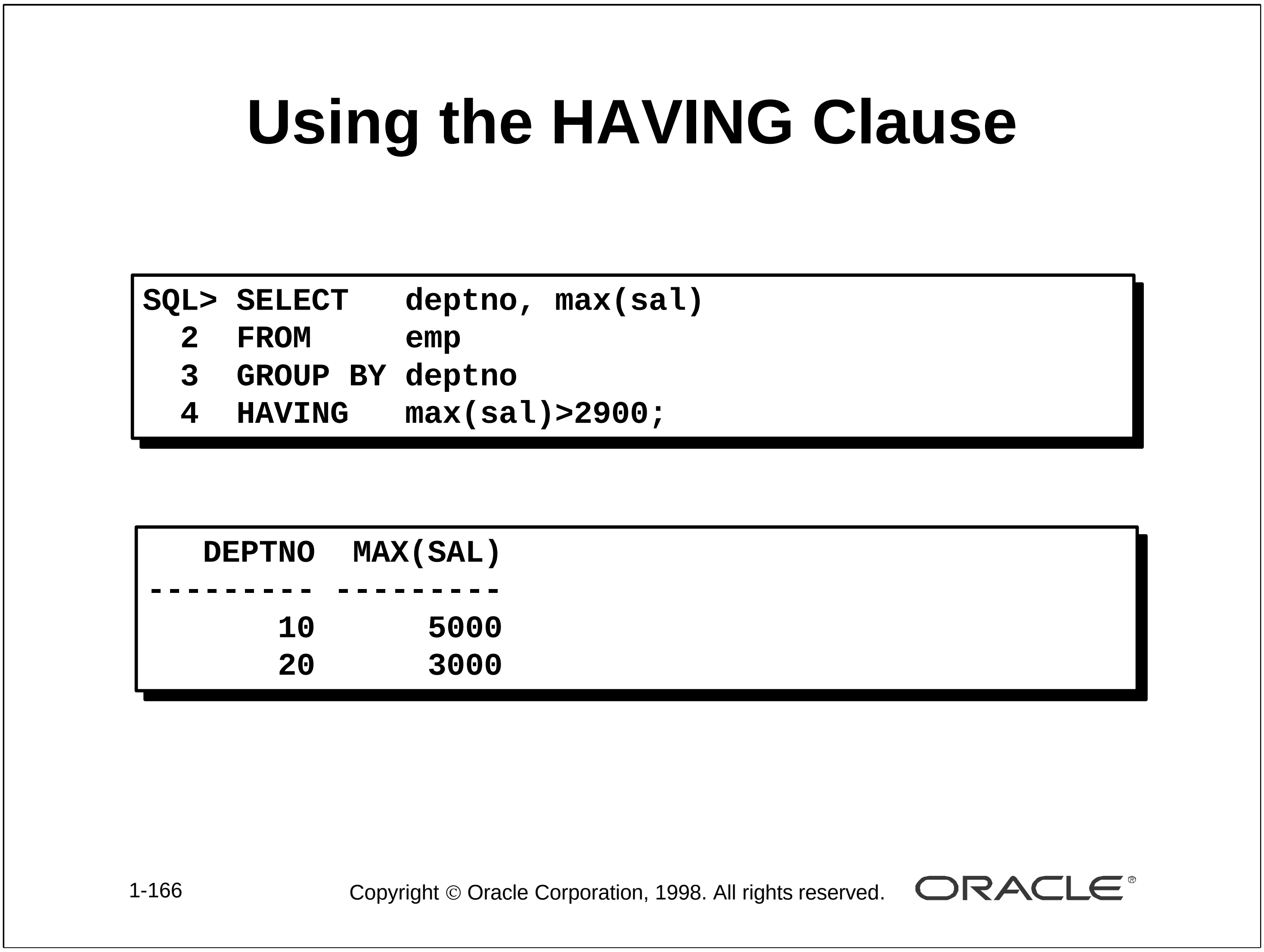
In the syntax:

| | |
|---|---|
| *group_condition* | restricts the groups of rows returned to those groups for which the specified condition is TRUE |

The Oracle8 Server performs the following steps when you use the HAVING clause:

- Rows are grouped.
- The group function is applied to the group.
- The groups that match the criteria in the HAVING clause are displayed.

The HAVING clause can precede the GROUP BY clause, but it is recommended that you place the GROUP BY clause first because it is more logical. Groups are formed and group functions are calculated before the HAVING clause is applied to the groups in the SELECT list.



# Using the HAVING Clause

```
SQL> SELECT   deptno, max(sal)
  2  FROM     emp
  3  GROUP BY deptno
  4  HAVING   max(sal)>2900;
```

```
   DEPTNO  MAX(SAL)
--------- ---------
       10      5000
       20      3000
```

1-166

### The HAVING Clause (continued)

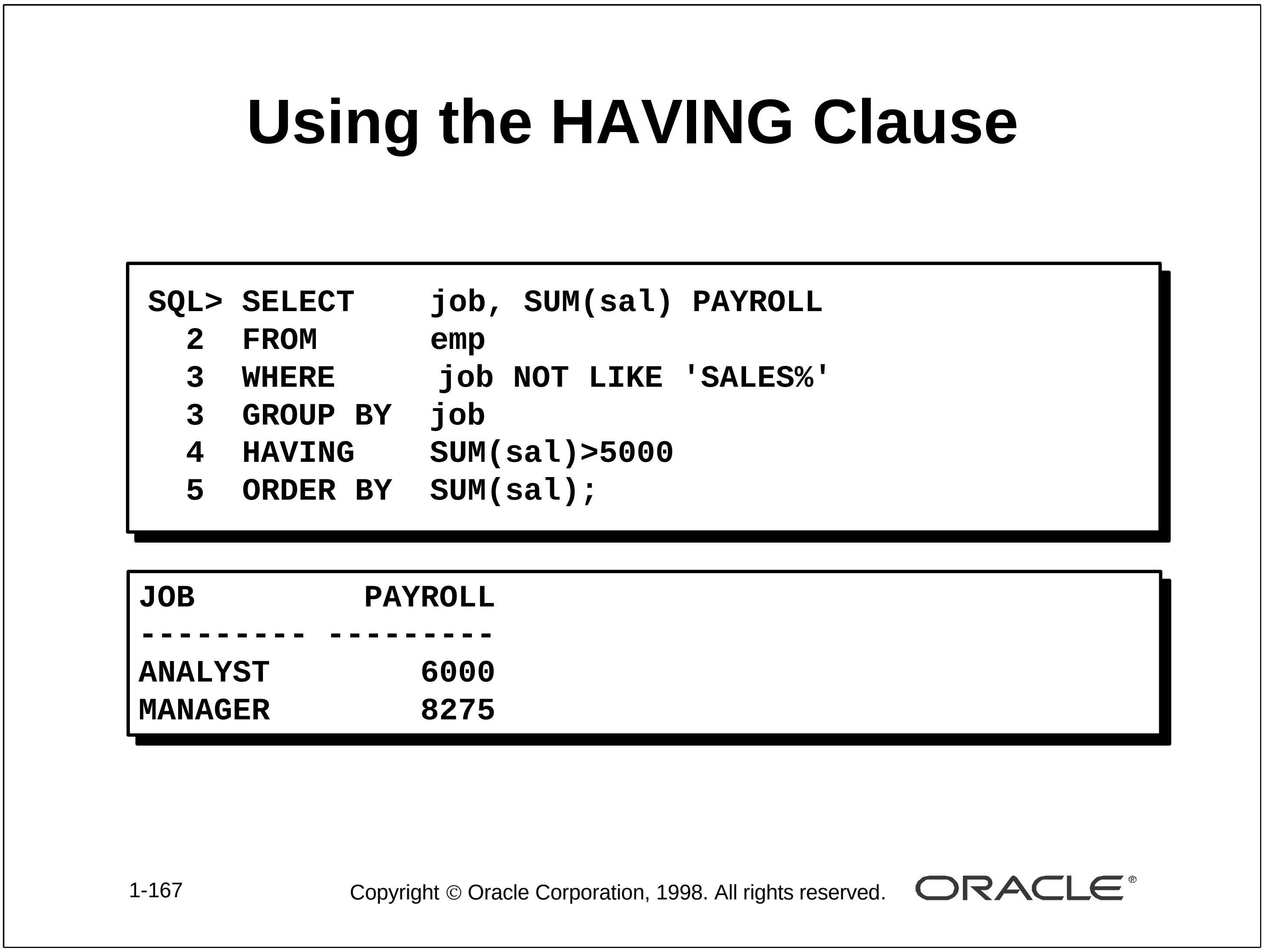
The above example displays department numbers and maximum salary for those departments whose maximum salary is greater than $2900.

You can use the GROUP BY clause without using a group function in the SELECT list.

If you restrict rows based on the result of a group function, you must have a GROUP BY clause as well as the HAVING clause.

```
SQL> SELECT     deptno
  2  FROM       emp
  3  GROUP BY   deptno
  4  HAVING     MAX(sal) > 2900;
```

```
   DEPTNO
---------
       10
       20
```



# Using the HAVING Clause

```
SQL> SELECT    job, SUM(sal) PAYROLL
  2  FROM      emp
  3  WHERE      job NOT LIKE 'SALES%'
  3  GROUP BY  job
  4  HAVING    SUM(sal)>5000
  5  ORDER BY  SUM(sal);
```

```
JOB         PAYROLL
--------- ---------
ANALYST        6000
MANAGER        8275
```

ORACLE®

### The HAVING Clause (continued)

The above example displays the job title and total monthly salary for each job title with a total payroll exceeding $5000. The example excludes salesmen and sorts the list by the total monthly salary.

### Class Management Note

Demo: *l5job1.sql*, *l5job2.sql*

# Nesting Group Functions

## Display the maximum average salary.

```
SQL> SELECT   max(avg(sal))
  2  FROM     emp
  3  GROUP BY deptno;
```

```
MAX(AVG(SAL))
-------------
    2916.6667
```

### Nesting Group Functions

Group functions can be nested. The above example displays the maximum average salary.

# Summary

```
SELECT        column, group_function
FROM          table
[WHERE        condition]
[GROUP BY     group_by_expression]
[HAVING       group_condition]
[ORDER BY     column];
```

### Summary

Seven group functions are available in SQL:

- AVG
- COUNT
- MAX
- MIN
- SUM
- STDDEV
- VARIANCE

You can create subgroups by using the GROUP BY clause. Groups can be excluded using the HAVING clause.

Place the HAVING and GROUP BY clauses after the WHERE clause in a statement. Place the ORDER BY clause last.

Oracle8 Server evaluates the clauses in the following order:

- If the statement contains a WHERE clause, the server establishes the candidate rows.
- The server identifies the groups specified in the GROUP BY clause.
- The HAVING clause further restricts result groups that do not meet the group criteria in the HAVING clause.

# Practice Overview

- Showing different queries that use group functions
- Grouping by rows to achieve more than one result
- Excluding groups by using the HAVING clause

### Practice Overview

At the end of this practice, you should be familiar with using group functions and selecting groups of data.

### Paper-Based Questions

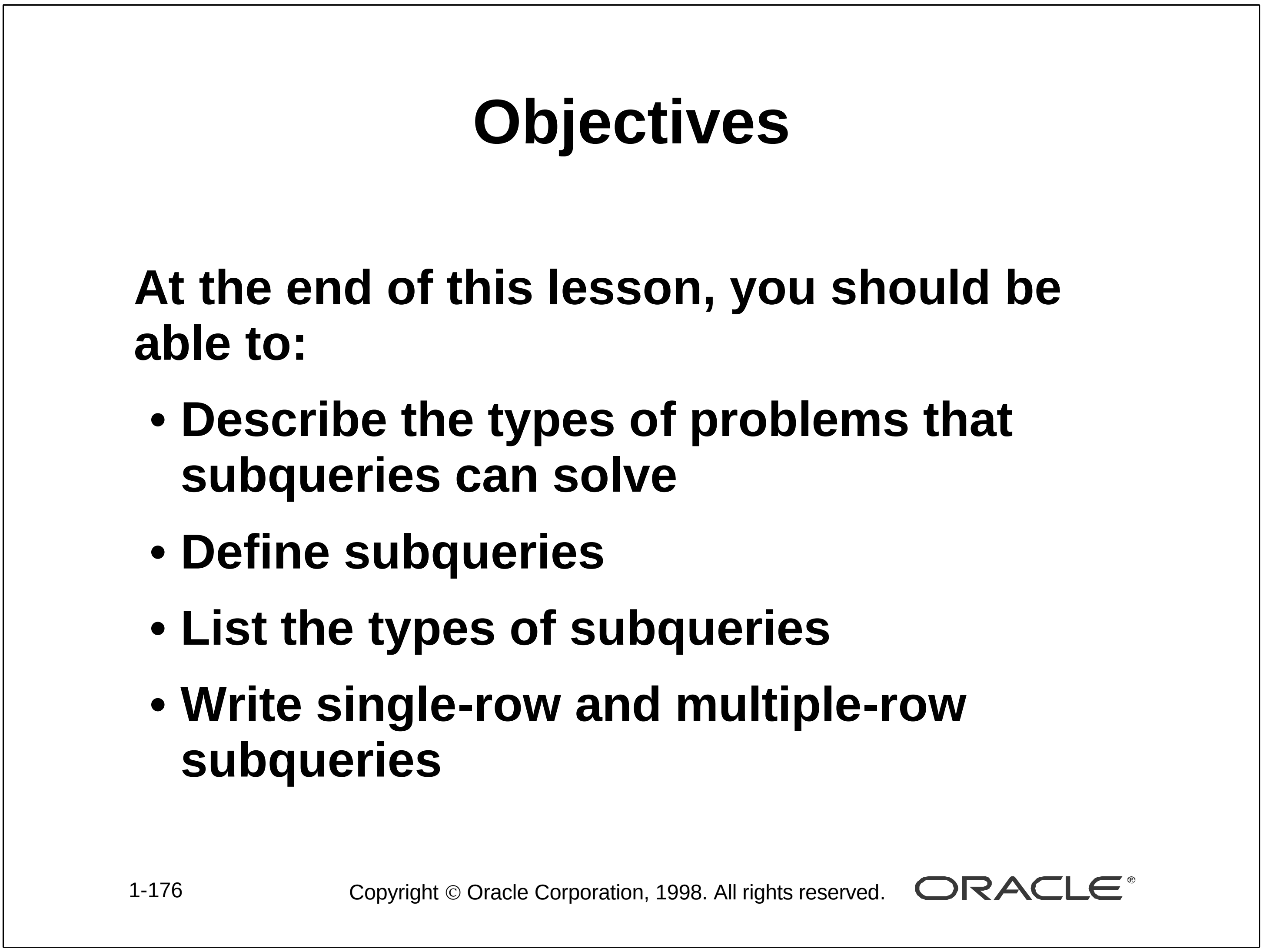
For questions 1-3 circle either True or False.

**Note:** Column aliases are used for the queries.

1

# Subqueries

| Schedule: | Timing | Topic |
|---|---|---|
| | 25 minutes | Lecture |
| | 30 minutes | Practice |
| | 55 minutes | Total |



# Objectives

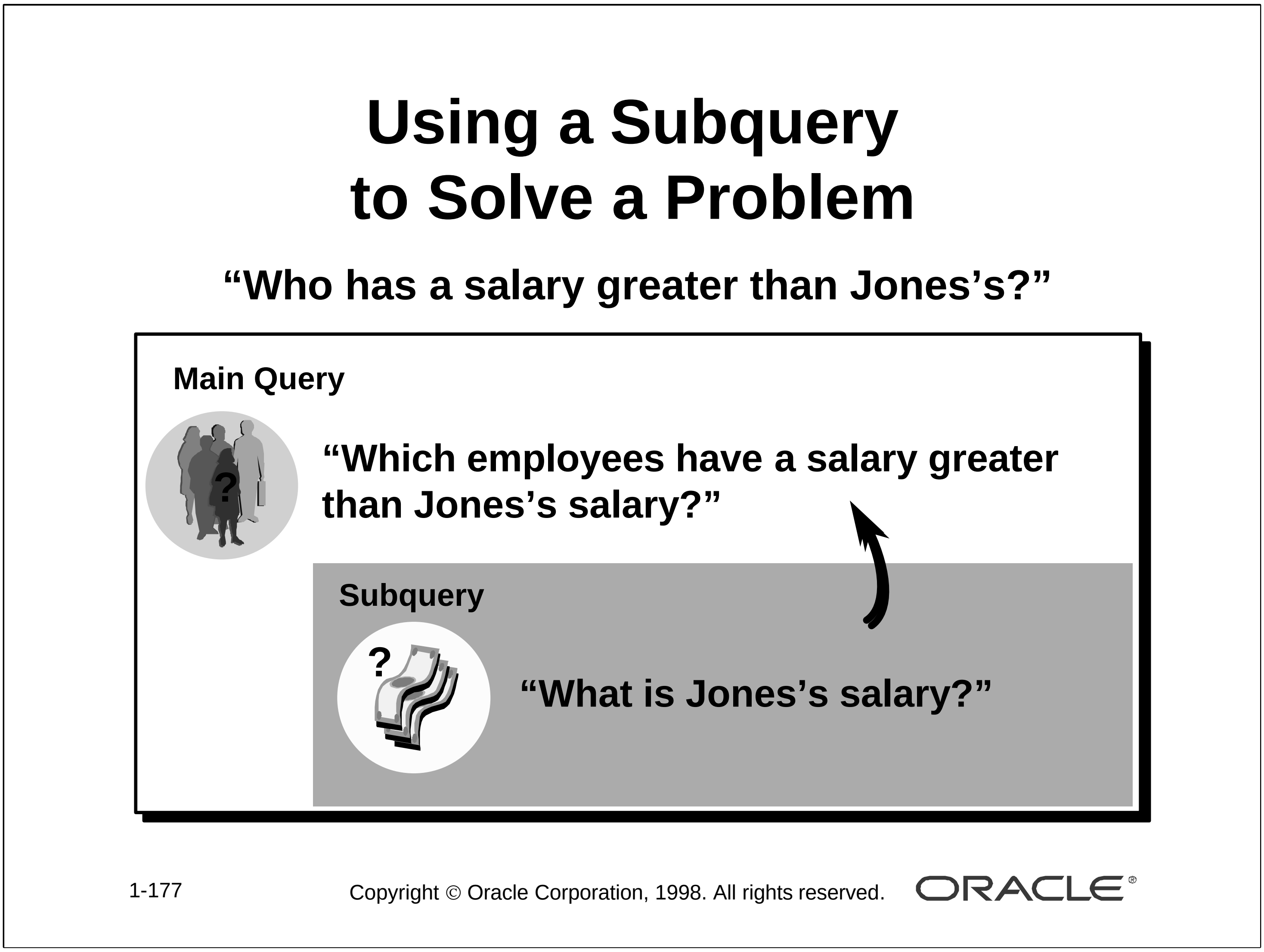
At the end of this lesson, you should be able to:

- Describe the types of problems that subqueries can solve
- Define subqueries
- List the types of subqueries
- Write single-row and multiple-row subqueries

1-176

### Lesson Aim

In this lesson you will learn about more advanced features of the SELECT statement. You can write subqueries in the WHERE clause of another SQL statement to obtain values based on an unknown conditional value. This lesson covers single-row subqueries and multiple-row subqueries.



# Using a Subquery to Solve a Problem

“Who has a salary greater than Jones’s?”

Main Query

“Which employees have a salary greater than Jones’s salary?”

Subquery

“What is Jones’s salary?”

1-177

### Using a Subquery to Solve a Problem

Suppose you want to write a query to find out who earns a salary greater than Jones’s salary.

To solve this problem, you need *two* queries: one query to find what Jones earns and a second query to find who earns more than that amount.

You can solve this problem by combining the two queries, placing one query *inside* the other query.

An inner query or the *subquery* returns a value that is used by the outer query or the main query. Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search value in the second query.

# Subqueries

```
SELECT   select_list
FROM     table
WHERE    expr operator
                     (SELECT      select_list
                      FROM        table);
```

- **The subquery (inner query) executes once before the main query.**
- **The result of the subquery is used by the main query (outer query).**

### Subqueries

A subquery is a SELECT statement that is embedded in a clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

You can place the subquery in a number of SQL clauses:

- WHERE clause
- HAVING clause
- FROM clause

In the syntax:

*operator* includes a comparison operator such as >, =, or IN

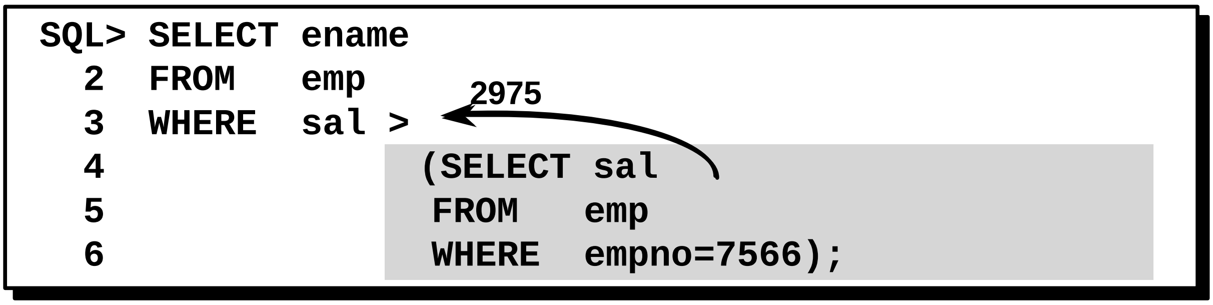
**Note:** Comparison operators fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL) .

The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement. The subquery generally executes first, and its output is used to complete the query condition for the main or outer query.

### Class Management Note

Additionally, subqueries can be placed in the CREATE VIEW statement, CREATE TABLE statement, UPDATE clause, INTO clause of an INSERT statement, and SET clause of an UPDATE statement.

# Using a Subquery



```
SQL> SELECT ename
  2  FROM   emp
  3  WHERE  sal >
  4                (SELECT sal
  5                 FROM   emp
  6                 WHERE  empno=7566);
```

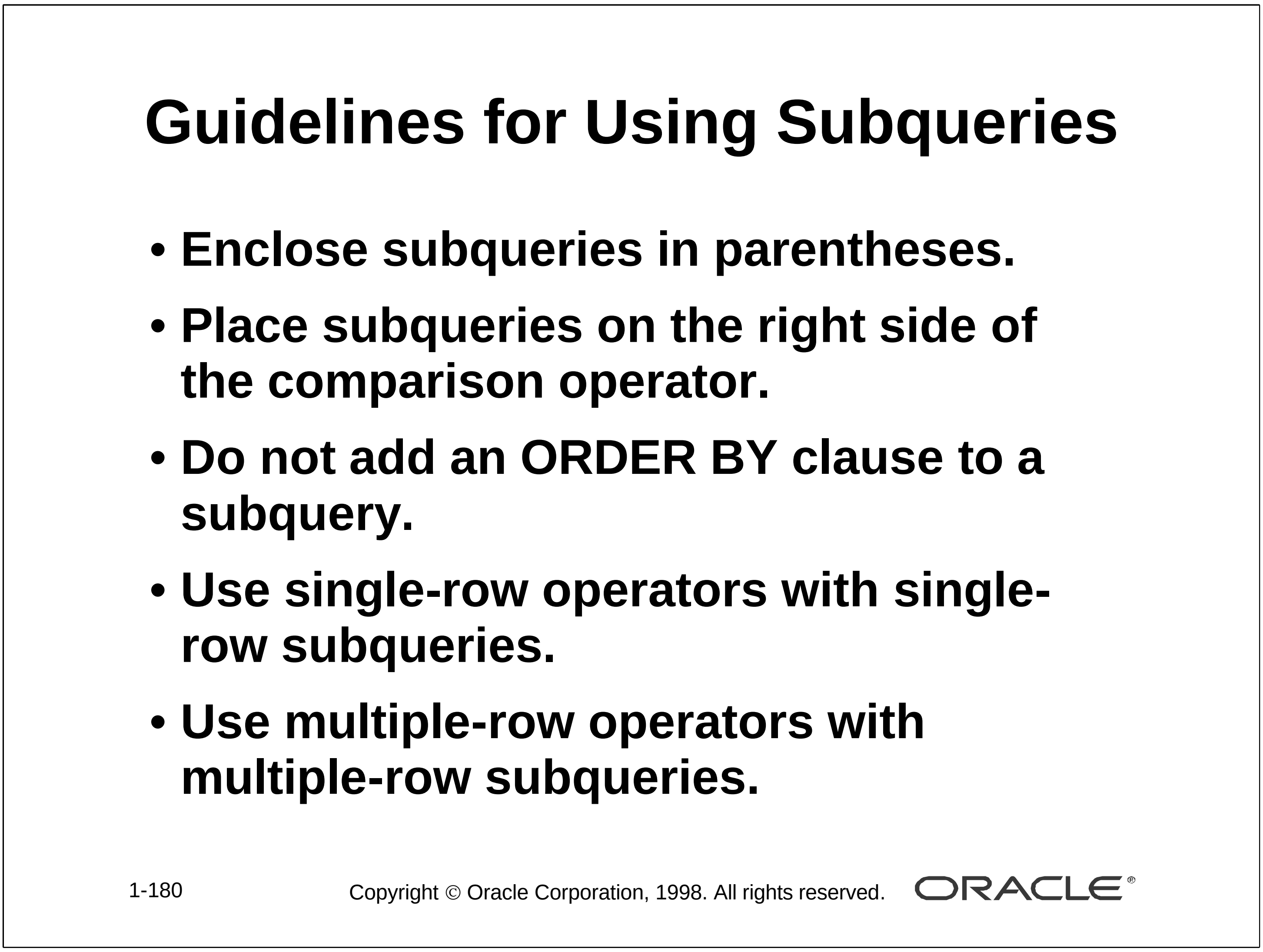
```
ENAME
----------
KING
FORD
SCOTT
```

### Using a Subquery

In the slide, the inner query determines the salary of employee 7566. The outer query takes the result of the inner query and uses this result to display all the employees who earn more than this amount.

### Class Management Note

Execute the subquery (inner query) on its own first to show the value that the subquery returns. Then execute the outer query using the result returned by the inner query. Finally, execute the entire query (containing the subquery) and show that the result is the same.



# Guidelines for Using Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison operator.
- Do not add an ORDER BY clause to a subquery.
- Use single-row operators with single-row subqueries.
- Use multiple-row operators with multiple-row subqueries.

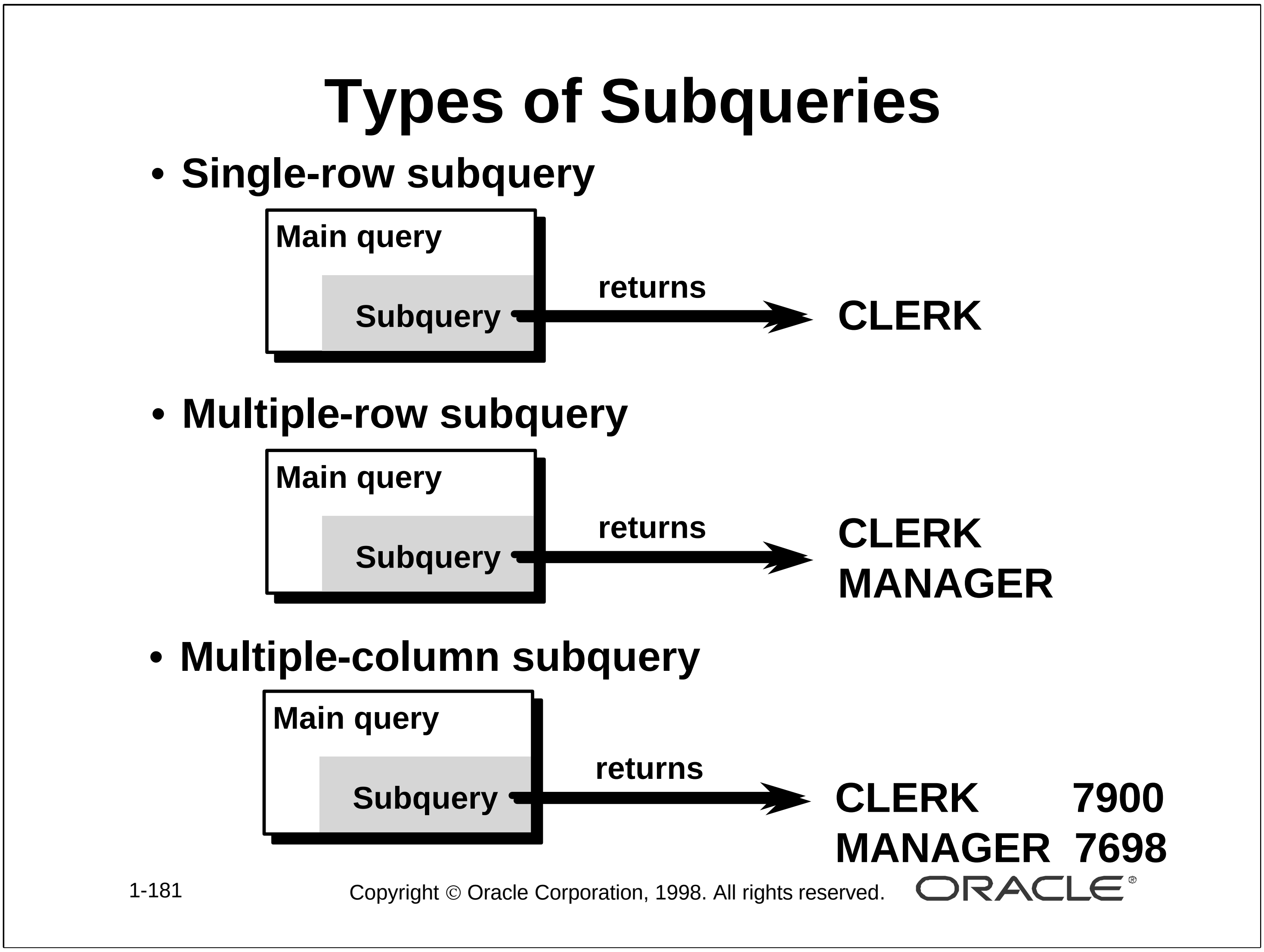
1-180

### Guidelines for Using Subqueries

- A subquery must be enclosed in parentheses.
- A subquery must appear on the right side of the comparison operator.
- Subqueries cannot contain an ORDER BY clause. You can have only one ORDER BY clause for a SELECT statement, and if specified it must be the last clause in the main SELECT statement.
- Two classes of comparison operators are used in subqueries: single-row operators and multiple-row operators.

### Class Management Note

A subquery can execute multiple times in correlated subqueries, which are not included in this course. Refer students to the *Advanced SQL and SQL*Plus* course for this topic.
Students may ask how many subqueries can be written. The Oracle8 Server imposes no limit on the number of subqueries. The limit is related to the buffer size that the query uses.



## Types of Subqueries

- Single-row subquery

Main query

Subquery

returns

CLERK

- Multiple-row subquery

Main query

Subquery

returns

CLERK
MANAGER

- Multiple-column subquery

Main query

Subquery

returns

| | |
|---|---|
| CLERK | 7900 |
| MANAGER | 7698 |

1-181

**Types of Subqueries**

- Single-row subqueries: Queries that return only one row from the inner SELECT statement
- Multiple-row subqueries: Queries that return more than one row from the inner SELECT statement
- Multiple-column subqueries: Queries that return more than one column from the inner SELECT statement

# Single-Row Subqueries

- Return only one row
- Use single-row comparison operators

| Operator | Meaning |
|---|---|
| = | Equal to |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |
| <> | Not equal to |

### Single Row Subqueries

A *single-row subquery* is one that returns one row from the inner SELECT statement. This type of subquery uses a single-row operator. The slide gives a list of single-row operators.

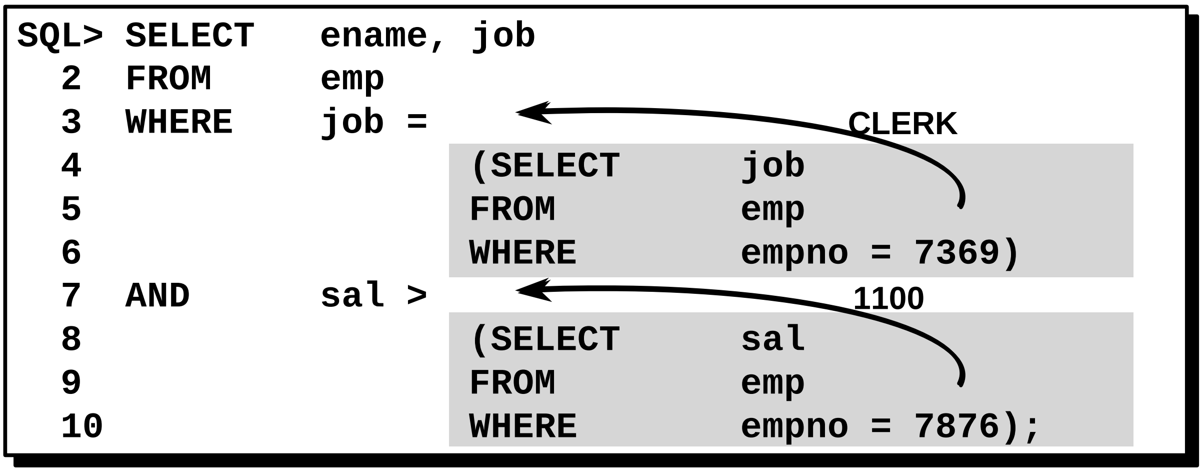
### Example

Display the employees whose job title is the same as that of employee 7369.

```
SQL> SELECT   ename, job
  2  FROM     emp
  3  WHERE    job =
  4               (SELECT  job
  5                FROM    emp
  6                WHERE   empno = 7369);
```

```
ENAME      JOB
---------- ---------
JAMES      CLERK
SMITH      CLERK
ADAMS      CLERK
MILLER     CLERK
```

# Executing Single-Row Subqueries



```
SQL> SELECT   ename, job
  2  FROM     emp
  3  WHERE    job =
  4                 (SELECT    job
  5                 FROM       emp
  6                 WHERE      empno = 7369)
  7  AND      sal >
  8                 (SELECT    sal
  9                 FROM       emp
 10                 WHERE      empno = 7876);
```

```
ENAME      JOB
---------- ---------
MILLER     CLERK
```

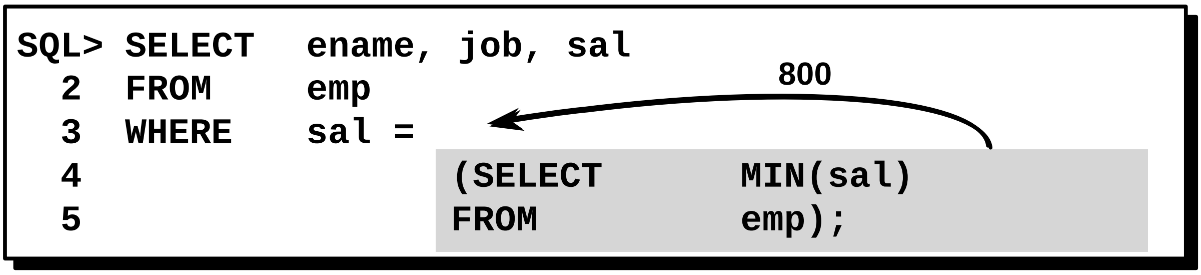
### Executing Single Row Subqueries

A SELECT statement can be considered as a query block. The example above displays employees whose job title is the same as that of employee 7369 and whose salary is greater than that of employee 7876.

The example consists of three query blocks: the outer query and two inner queries. The inner query blocks are executed first, producing the query results: CLERK and 1100, respectively. The outer query block is then processed and uses the values returned by the inner queries to complete its search conditions.

Both the inner queries return single values (CLERK and 1100, respectively), so this SQL statement is called a single-row subquery.

# Using Group Functions in a Subquery



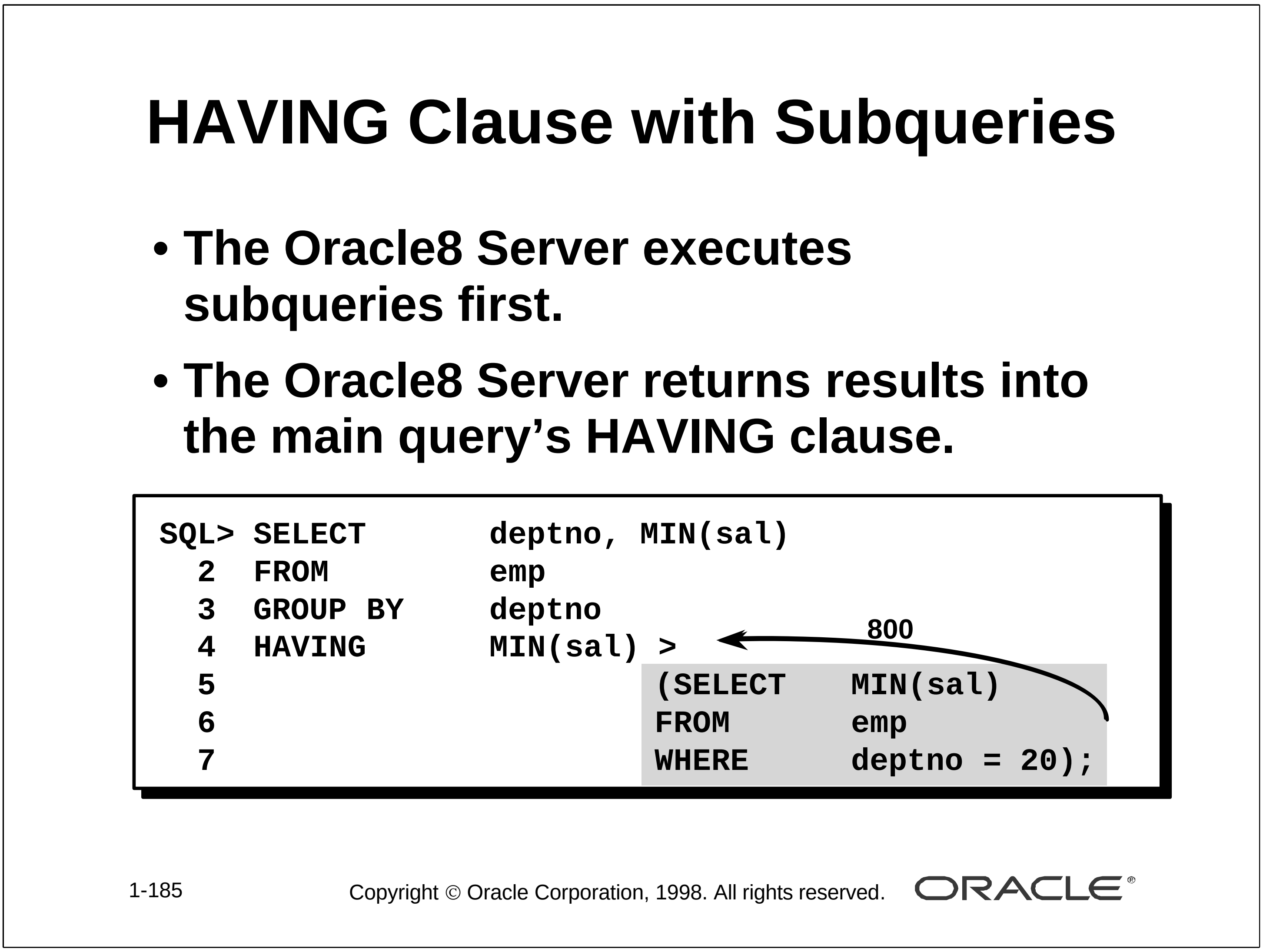
```
SQL> SELECT   ename, job, sal
  2  FROM     emp
  3  WHERE    sal =
  4                   (SELECT      MIN(sal)
  5                   FROM         emp);
```

```
ENAME      JOB               SAL
---------- --------- ---------
SMITH      CLERK             800
```

### Using Group Functions in a Subquery

You can display data from a main query by using a group function in a subquery to return a single row. The subquery is in parentheses and is placed after the comparison operator.

The example above displays the employee name, job title, and salary of all employees whose salary is equal to the minimum salary. The MIN group function returns a single value (800) to the outer query.



# HAVING Clause with Subqueries

- The Oracle8 Server executes subqueries first.
- The Oracle8 Server returns results into the main query’s HAVING clause.

```
                                                    800
SQL> SELECT       deptno, MIN(sal)
  2  FROM         emp
  3  GROUP BY     deptno
  4  HAVING       MIN(sal) >
  5                         (SELECT   MIN(sal)
  6                         FROM      emp
  7                         WHERE     deptno = 20);
```

### HAVING Clause With Subqueries

You can use subqueries not only in the WHERE clause, but also in the HAVING clause. The Oracle8 Server executes the subquery, and the results are returned into the main query’s HAVING clause.

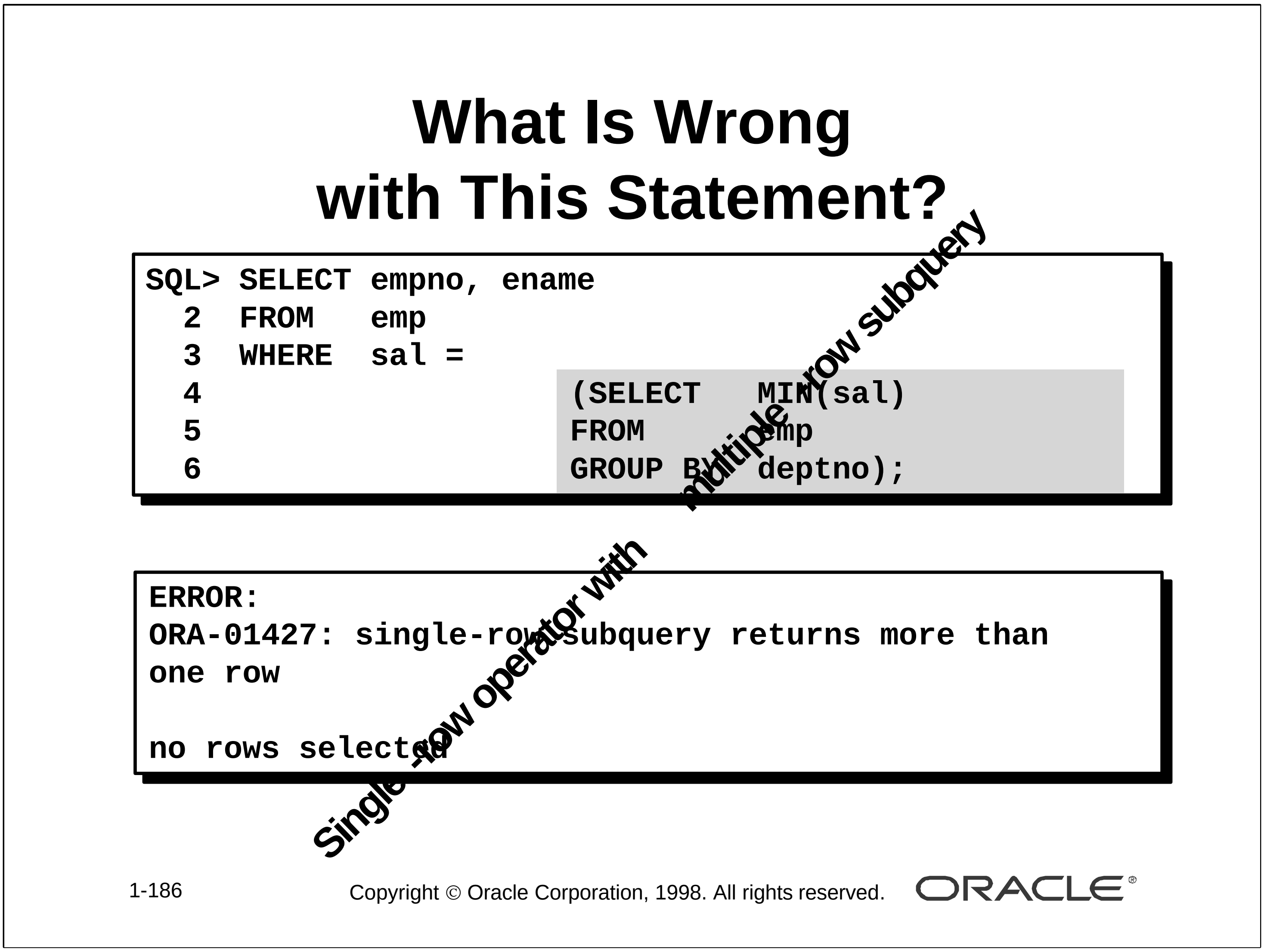
The SQL statement on the slide displays all the departments that have a minimum salary greater than that of department 20.

```
   DEPTNO   MIN(SAL)
--------- ---------
       10      1300
       30       950
```

**Example**

Find the job with the lowest average salary.

```
SQL> SELECT   job, AVG(sal)
  2  FROM     emp
  3  GROUP BY job
  4  HAVING   AVG(sal) = (SELECT  MIN(AVG(sal))
                          FROM    EMP
                         GROUP BY job);
```



# What Is Wrong with This Statement?

```
SQL> SELECT empno, ename
  2  FROM   emp
  3  WHERE  sal =
  4                      (SELECT   MIN(sal)
  5                      FROM      emp
  6                      GROUP BY  deptno);
```

```
ERROR:
ORA-01427: single-row subquery returns more than
one row

no rows selected
```

Single-row operator with multiple-row subquery

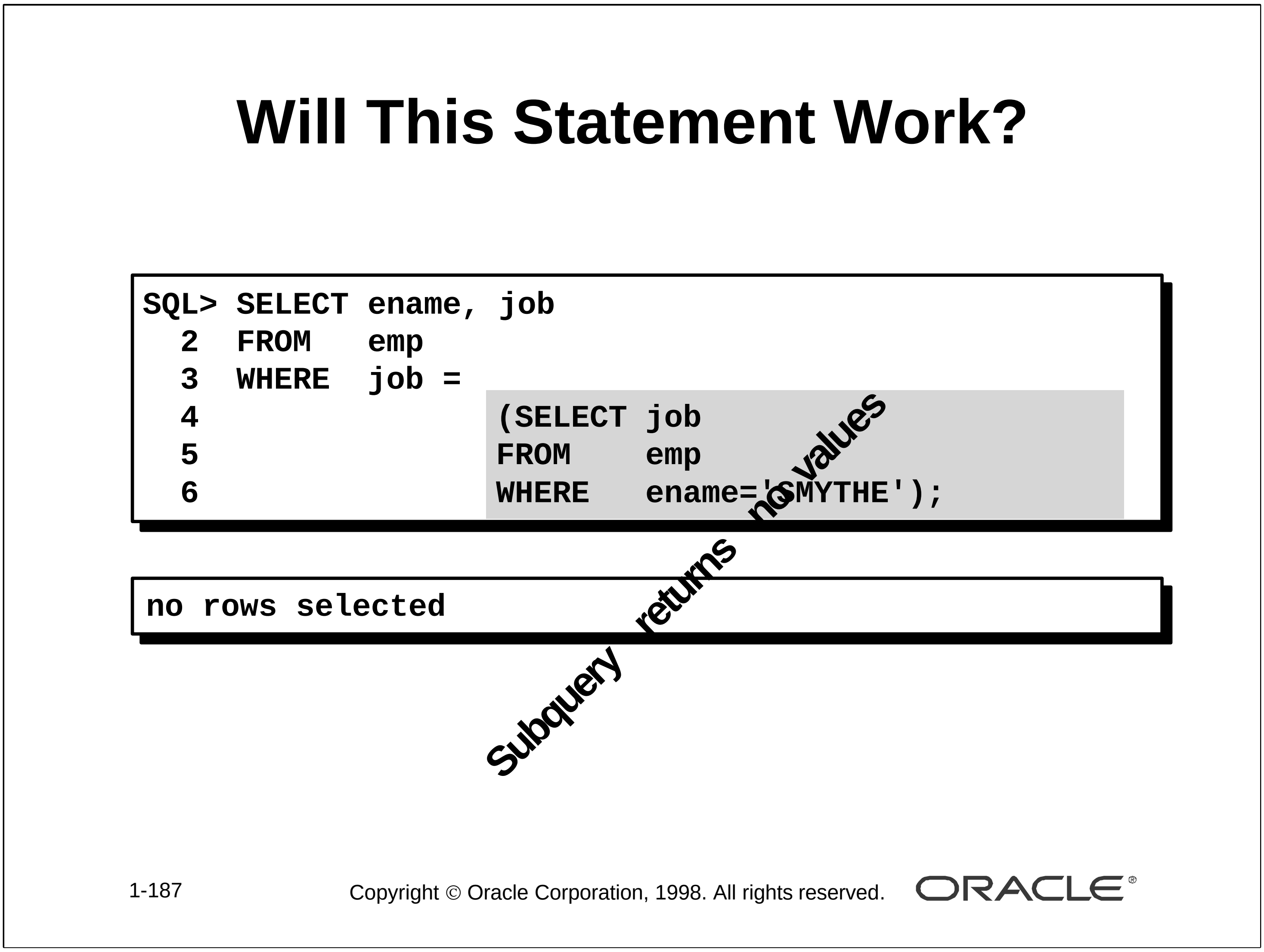
## Errors with Subqueries

One common error with subqueries is more than one row returned for a single-row subquery.

In the SQL statement above, the subquery contains a GROUP BY (deptno) clause, which implies that the subquery will return multiple rows, one for each group it finds. In this case, the result of the subquery will be 800, 1300, and 950.

The outer query takes the results of the subquery (800, 950, 1300) and uses these results in its WHERE clause. The WHERE clause contains an equal (=) operator, a single-row comparison operator expecting only one value. The = operator cannot accept more than one value from the subquery and hence generates the error.

To correct this error, change the = operator to IN.



# Will This Statement Work?

```
SQL> SELECT ename, job
  2  FROM   emp
  3  WHERE  job =
  4                 (SELECT job
  5                 FROM    emp
  6                 WHERE   ename='SMYTHE');
```

```
no rows selected
```

Subquery returns no values

1-187

**Errors with Subqueries (continued)**

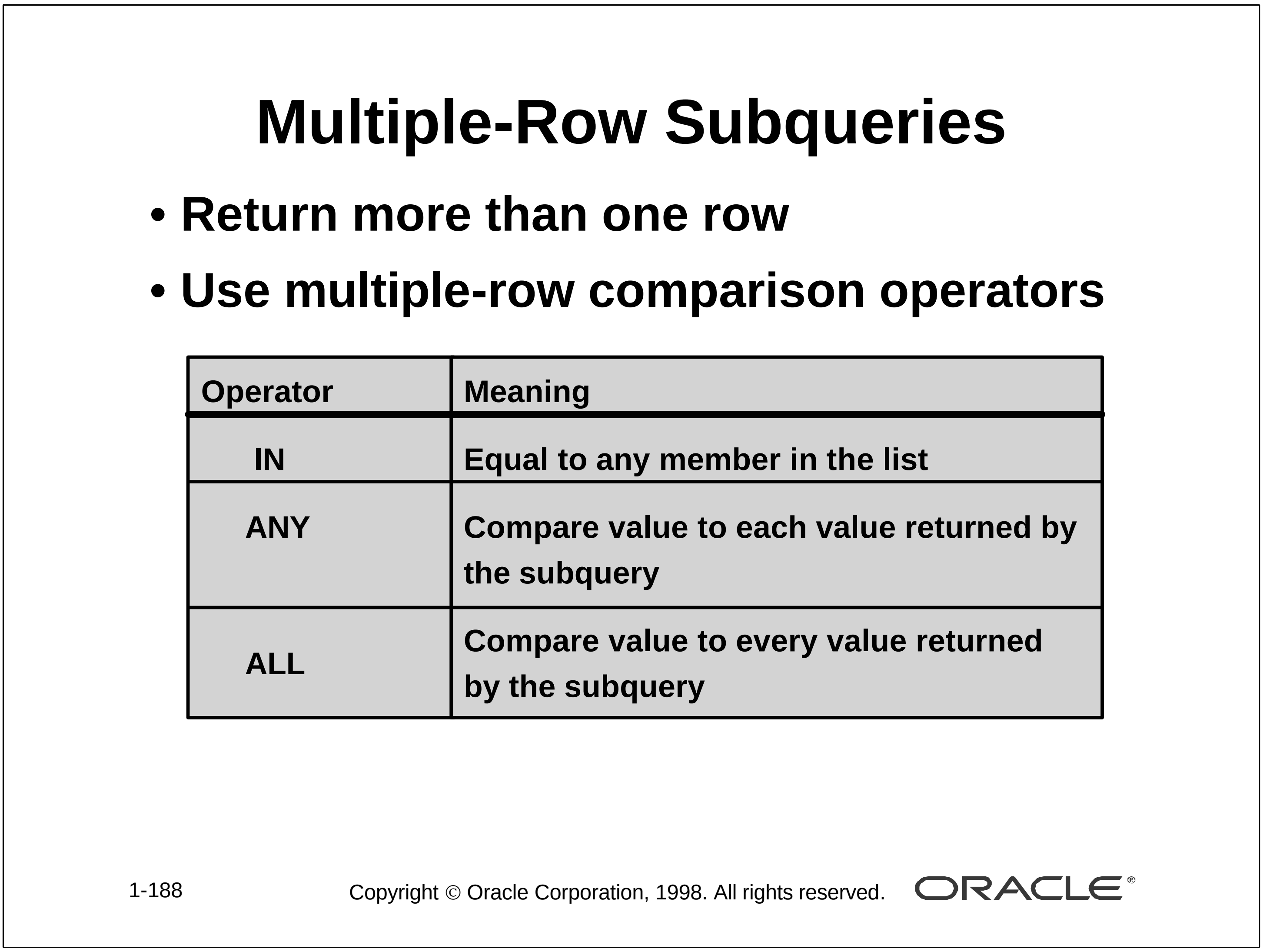
Another common error with subqueries is no rows being returned by the inner query.

In the SQL statement above, the subquery contains a WHERE (ename='SMYTHE') clause. Presumably, the intention is to find the employee whose name is Smythe. The statement seems to be correct but selects no rows when executed.

The problem is that Smythe is misspelled. There is no employee named Smythe. So the subquery returns no rows. The outer query takes the results of the subquery (null) and uses these results in its WHERE clause. The outer query finds no employee with a job title equal to null and so returns no rows.

**Class Management Note**

Earlier versions of SQL*Plus will generate an error ORA-1246 instead of the output shown above. You may want to show the correct SQL statement by changing the spelling of Smythe to Smith.



# Multiple-Row Subqueries

- Return more than one row
- Use multiple-row comparison operators

| Operator | Meaning |
|---|---|
| IN | Equal to any member in the list |
| ANY | Compare value to each value returned by the subquery |
| ALL | Compare value to every value returned by the subquery |

### Multiple-Row subqueries

Subqueries that return more than one row are called *multiple-row subqueries*. You use a multiple-row operator, instead of a single-row operator, with a multiple row subquery. The multiple-row operator expects one or more values.

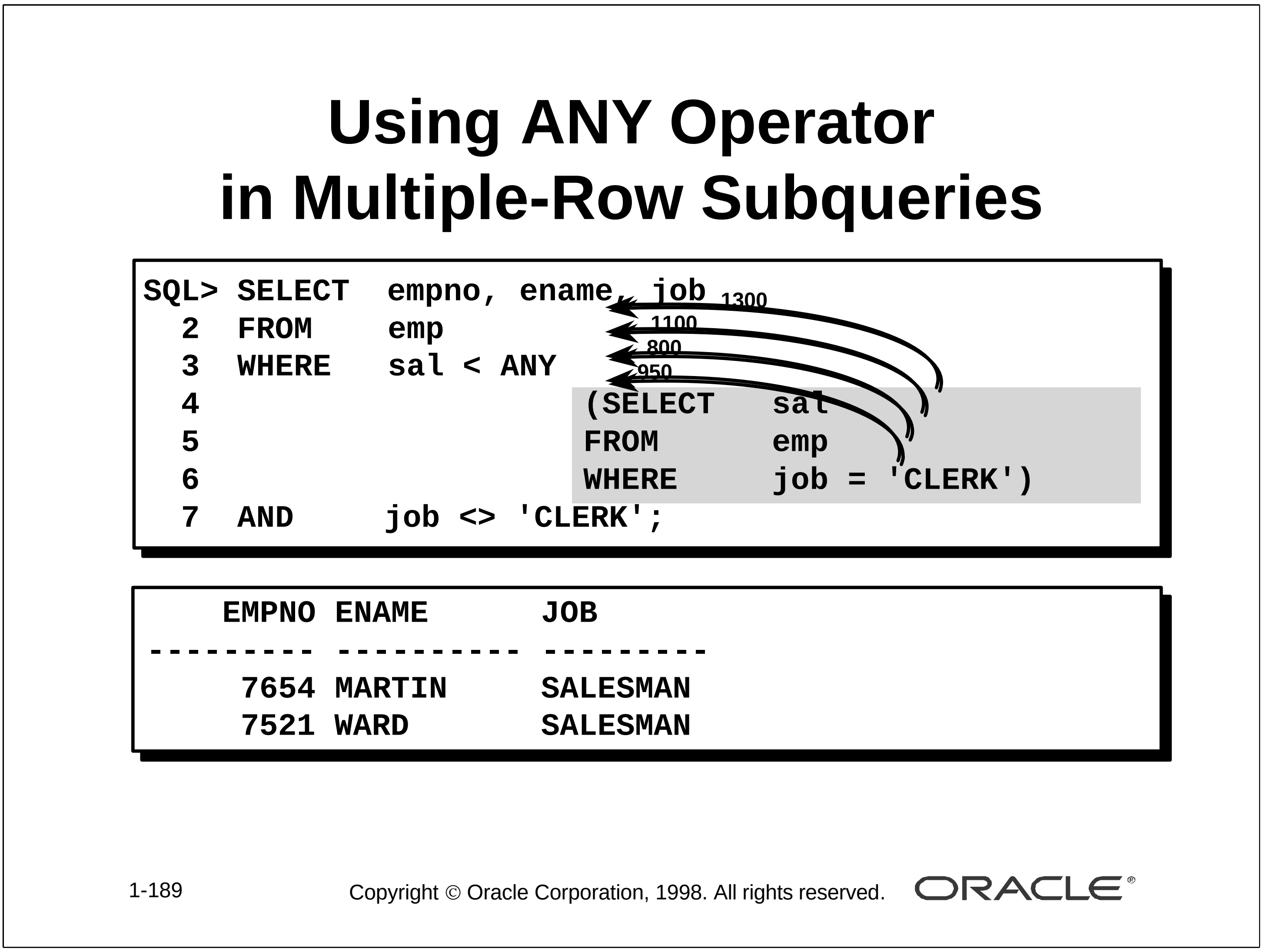
### Example

Find the employees who earn the same salary as the minimum salary for departments.

```
SQL> SELECT    ename, sal, deptno
  2  FROM      emp
  3  WHERE     sal IN (SELECT   MIN(sal)
  4  FROM     emp
  5  GROUP BY deptno);
```

The inner query is executed first, producing a query result containing three rows: 800, 950, 1300. The main query block is then processed and uses the values returned by the inner query to complete its search condition. In fact, the main query would look like the following to the Oracle8 Server:

```
SQL> SELECT   ename, sal, deptno
  2  FROM     emp
  3  WHERE    sal IN (800, 950, 1300);
```



# Using ANY Operator in Multiple-Row Subqueries

```
SQL> SELECT  empno, ename, job
  2  FROM    emp
  3  WHERE   sal < ANY
  4                    (SELECT   sal
  5                    FROM      emp
  6                    WHERE     job = 'CLERK')
  7  AND     job <> 'CLERK';
```

```
    EMPNO ENAME      JOB
--------- ---------- ---------
     7654 MARTIN     SALESMAN
     7521 WARD       SALESMAN
```

1-189

ORACLE®

### Multiple Row Subqueries

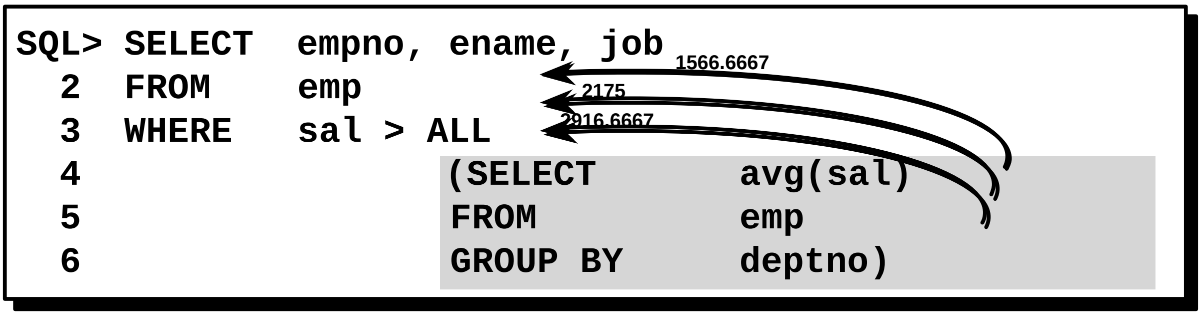
The ANY operator (and its synonym SOME operator) compares a value to *each* value returned by a subquery. The example above displays employees whose salary is less than any clerk and who are not clerks. The maximum salary that a clerk earns is $1300. The SQL statement displays all the employees who are not clerks but earn less than $1300.

<ANY means less than the maximum. >ANY means more than the minimum. =ANY is equivalent to IN.

### Class Management Note

When using SOME or ANY, you often use the DISTINCT keyword to prevent rows from being selected several times.

# Using ALL Operator in Multiple-Row Subqueries



```
SQL> SELECT  empno, ename, job
  2  FROM    emp
  3  WHERE   sal > ALL
  4                    (SELECT      avg(sal)
  5                     FROM        emp
  6                     GROUP BY    deptno)
```

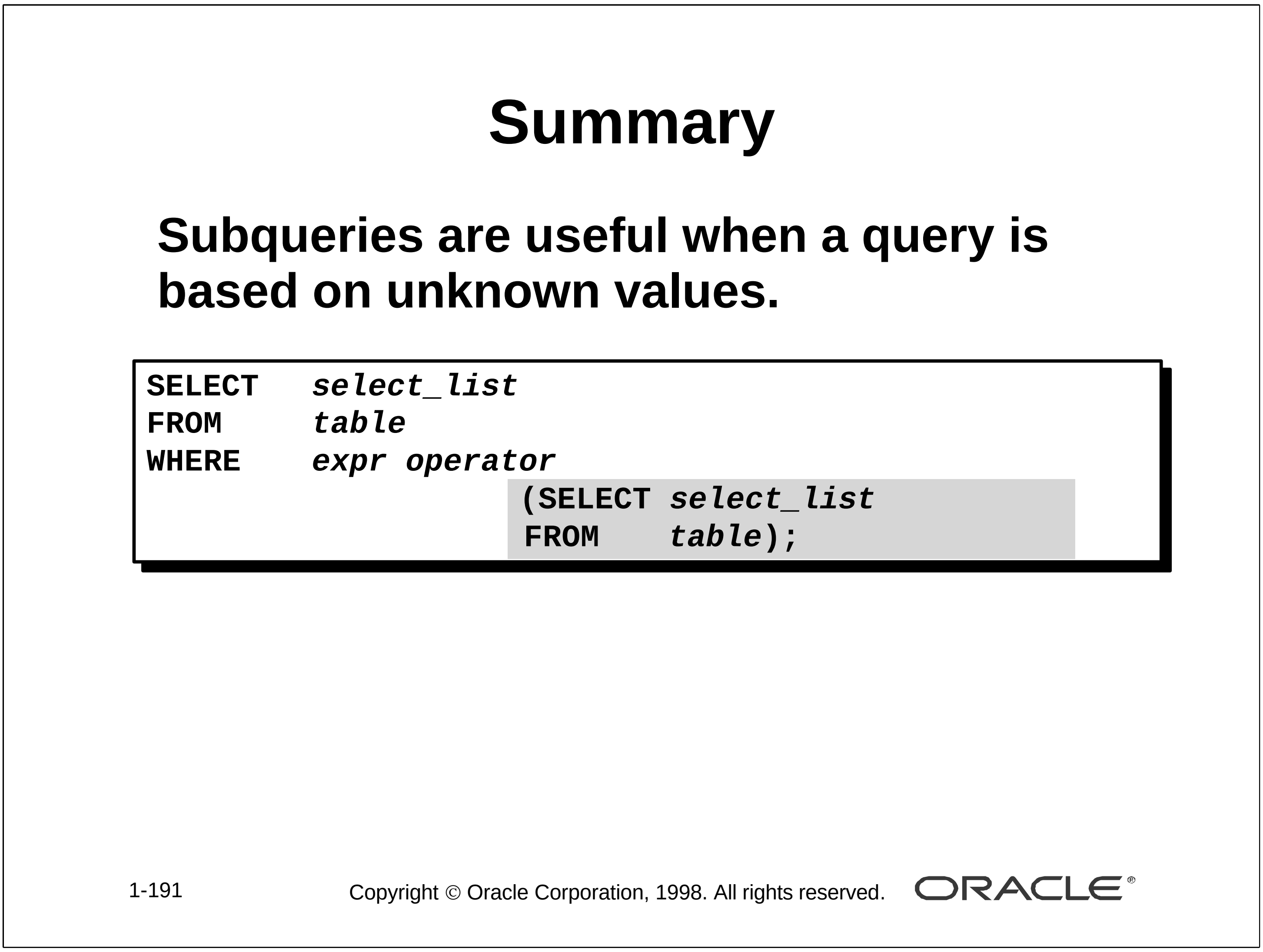
```
    EMPNO ENAME      JOB
--------- ---------- ---------
     7839 KING       PRESIDENT
     7566 JONES      MANAGER
     7902 FORD       ANALYST
     7788 SCOTT      ANALYST
```

### Multiple Row Subqueries

The ALL operator compares a value to *every* value returned by a subquery. The example above displays employees whose salary is greater than the average salaries of all the departments. The highest average salary of a department is $2916.66, so the query returns those employees whose salary is greater than $2916.66.

>ALL means more than the maximum and <ALL means less than the minimum.

The NOT operator can be used with IN, ANY, and ALL operators.



# Summary

## Subqueries are useful when a query is based on unknown values.

```
SELECT   select_list
FROM     table
WHERE    expr operator
                    (SELECT select_list
                     FROM   table);
```

1-191

### Summary

A subquery is a SELECT statement that is embedded in a clause of another SQL statement. Subqueries are useful when a query is based on unknown criteria.

Subqueries have the following characteristics:

- Can pass one row of data to a main statement that contains a single-row operator, such as =, <>, >, >=, <, or <=
- Can pass multiple rows of data to a main statement that contains a multiple-row operator, such as IN
- Are processed first by the Oracle8 Server, and the WHERE or HAVING clause uses the results
- Can contain group functions

# Practice Overview

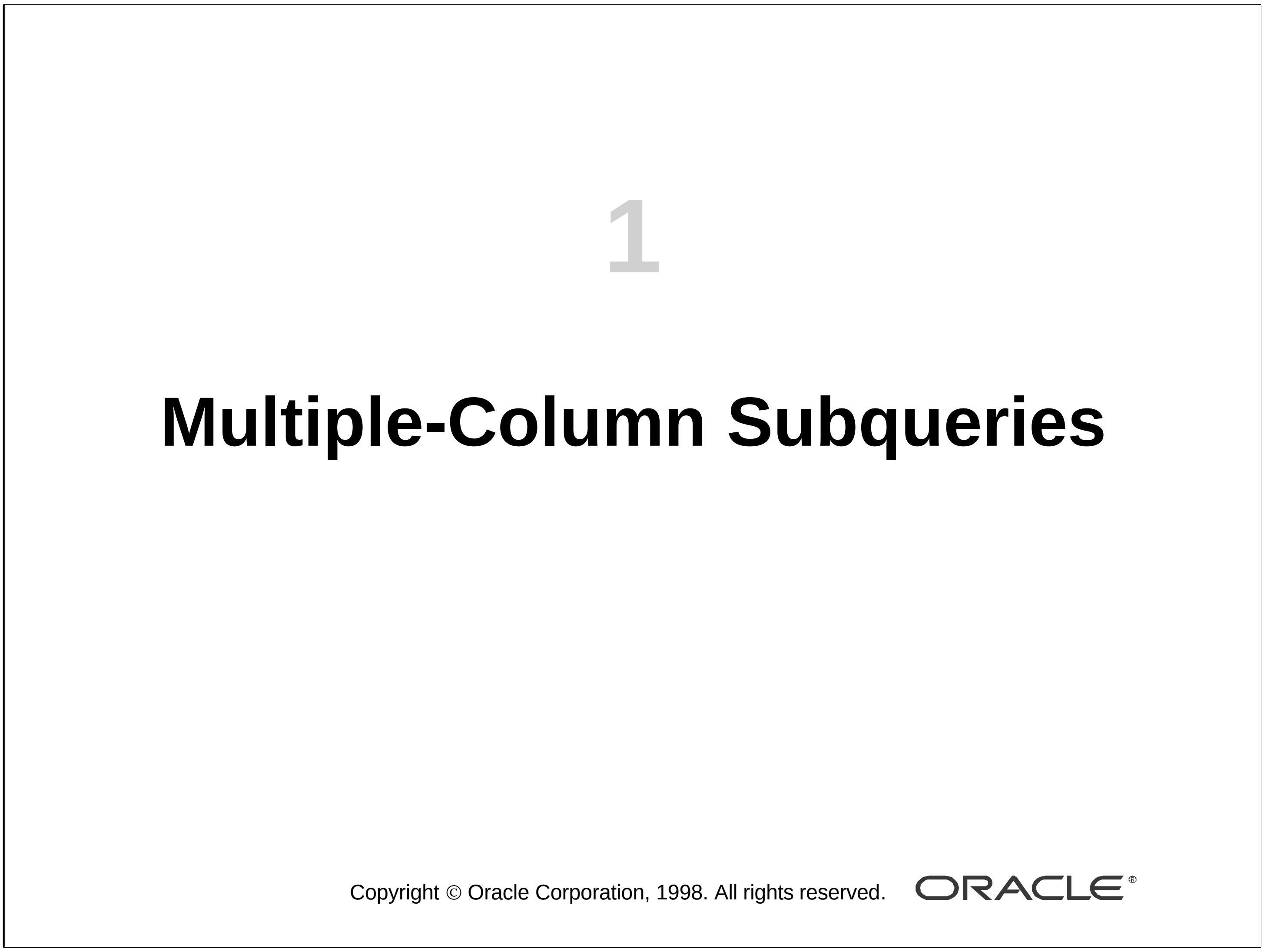
- Creating subqueries to query values based on unknown criteria
- Using subqueries to find out what values exist in one set of data and not in another

### Practice Overview

In this practice, you will write complex queries using nested SELECT statements.

### Paper-Based Questions

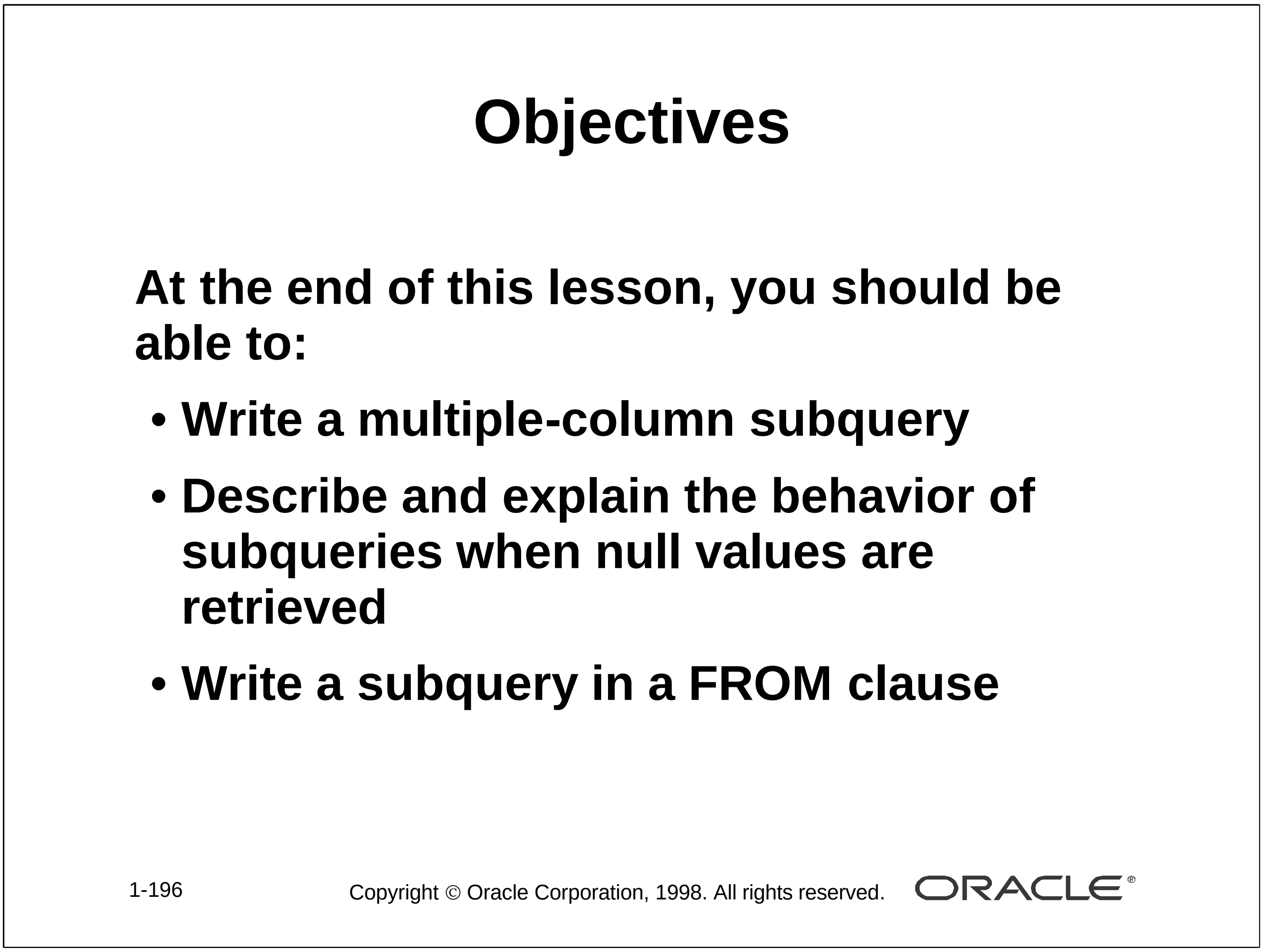
You may want to consider creating the inner query first for these questions. Make sure that it runs and produces the data you anticipate before coding the outer query.



1

# Multiple-Column Subqueries

| Schedule: | Timing | Topic |
|---|---|---|
| | 20 minutes | Lecture |
| | 20 minutes | Practice |
| | 40 minutes | Total |



# Objectives

At the end of this lesson, you should be able to:

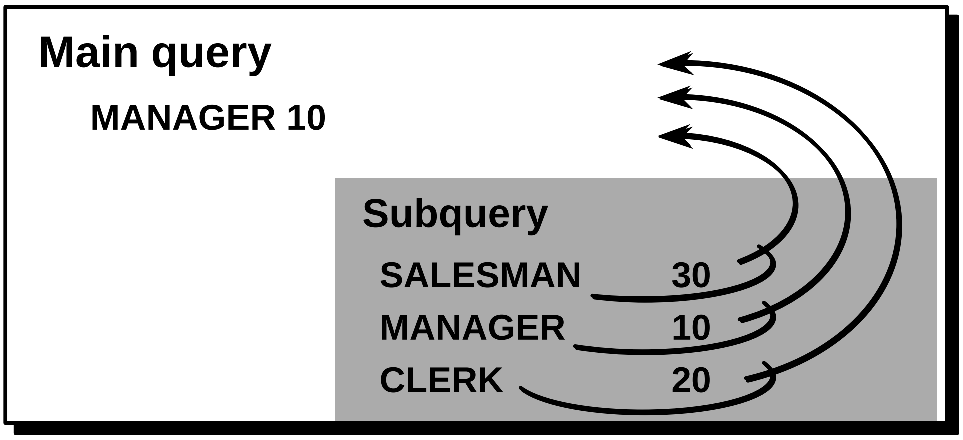
- Write a multiple-column subquery
- Describe and explain the behavior of subqueries when null values are retrieved
- Write a subquery in a FROM clause

1-196

**Lesson Aim**

In this lesson, you will learn how to write multiple-column subqueries and subqueries in the FROM clause of a SELECT statement.

# Multiple-Column Subqueries



| Main query compares | to | Values from a multiple-row and multiple-column subquery |
|---|---|---|
| MANAGER 10 | | SALESMAN 30<br>MANAGER 10<br>CLERK 20 |

1-197

### Multiple-Column Subqueries

So far you have written single-row subqueries and multiple-row subqueries where only one column was compared in the WHERE clause or HAVING clause of the SELECT statement. If you want to compare two or more columns, you must write a compound WHERE clause using logical operators. Multiple-column subqueries enable you to combine duplicate WHERE conditions into a single WHERE clause.

### Syntax

```
SELECT   column, column, ...
FROM     table
WHERE    (column, column, ...) IN
                             (SELECT column, column, ...
                              FROM   table
                              WHERE  condition);
```

# Using Multiple-Column Subqueries

**Display the name, department number, salary, and commission of any employee whose salary and commission matches both the commission and salary of any employee in department 30.**

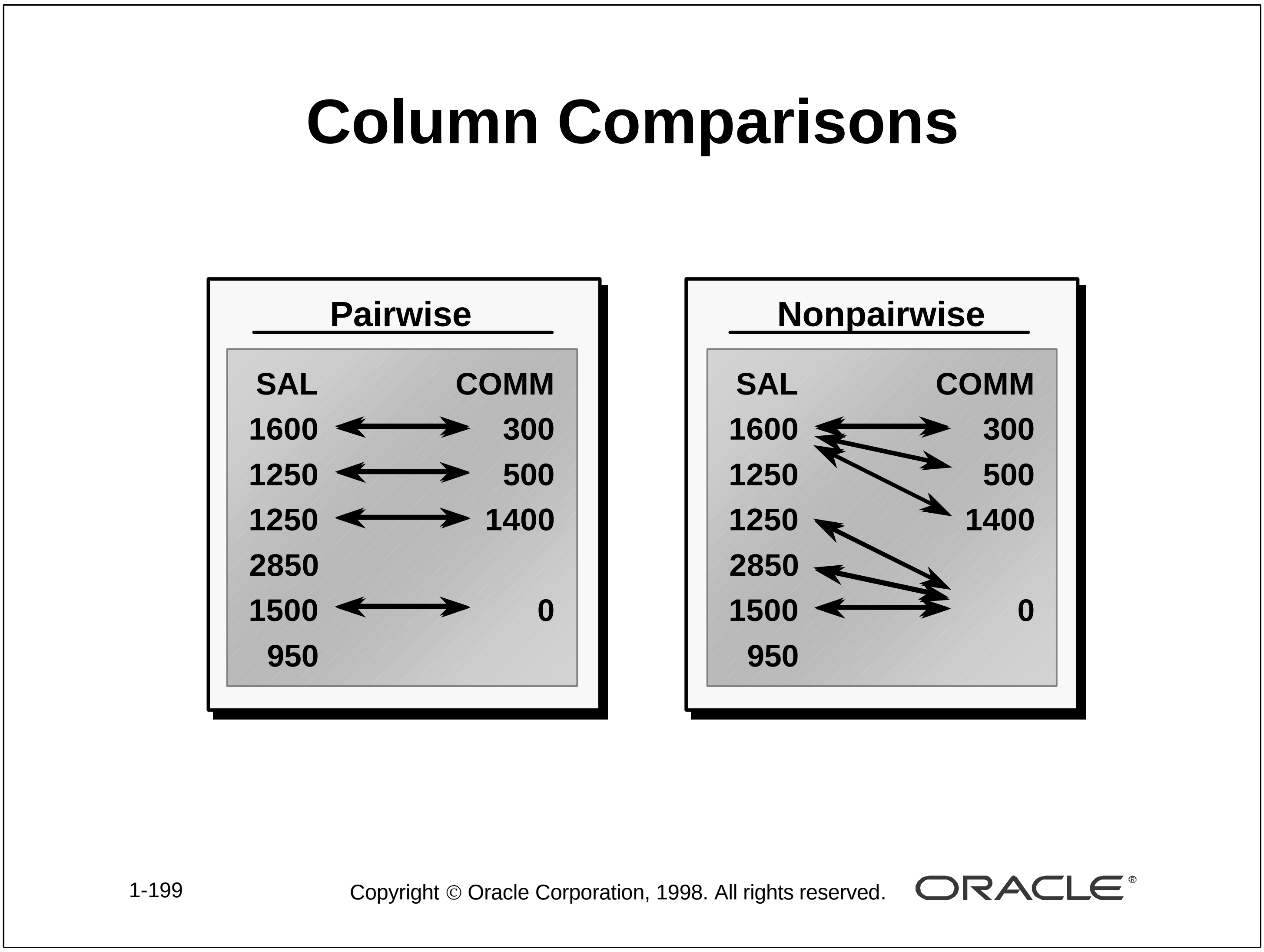
```
SQL> SELECT   ename, deptno, sal, comm
  2  FROM     emp
  3  WHERE   (sal, NVL(comm,-1)) IN
  4                             (SELECT sal, NVL(comm,-1)
  5                              FROM   emp
  6                              WHERE  deptno = 30);
```

### Using Multiple-Column Subqueries

The example above is that of a multiple-column subquery because the subquery returns more than one column. It compares the SAL column and the COMM column. It displays the name, department number, salary, and commission of any employee whose salary and commission matches *both* the commission and salary of any employee in department 30.

The output of the above SQL statement will be as follows:

```
ENAME         DEPTNO       SAL      COMM
---------- --------- --------- ---------
JAMES             30       950
WARD              30      1250       500
MARTIN            30      1250      1400
TURNER            30      1500         0
ALLEN             30      1600       300
BLAKE             30      2850

6 rows selected.
```



# Column Comparisons

Pairwise

SAL COMM
1600 300
1250 500
1250 1400
2850
1500 0
950

Nonpairwise

SAL COMM
1600 300
1250 500
1250 1400
2850
1500 0
950

1-199

## Pairwise Versus Nonpairwise Comparisons

Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons. In the example on the previous slide, a pairwise comparison was executed in the WHERE clause. Each candidate row in the SELECT statement must have *both* the same salary and the same commission of an employee in department 30.

If you want a nonpairwise comparison (a cross product), you must use a WHERE clause with multiple conditions.

# Nonpairwise Comparison Subquery

**Display the name, department number, salary, and commission of any employee whose salary and commission matches the commission and salary of any employee in department 30.**

```
SQL> SELECT   ename, deptno, sal, comm
  2  FROM     emp
  3  WHERE    sal IN               (SELECT sal
  4                                 FROM    emp
  5                                 WHERE   deptno = 30)
  6  AND
  7           NVL(comm,-1) IN      (SELECT  NVL(comm,-1)
  8                                 FROM    emp
  9                                 WHERE   deptno = 30);
```

1-200

### Nonpairwise Comparison Subquery

The above example does a nonpairwise comparison of the columns. It displays the name, department number, salary, and commission of any employee whose salary and commission match the salary and commission of any employee in department 30.

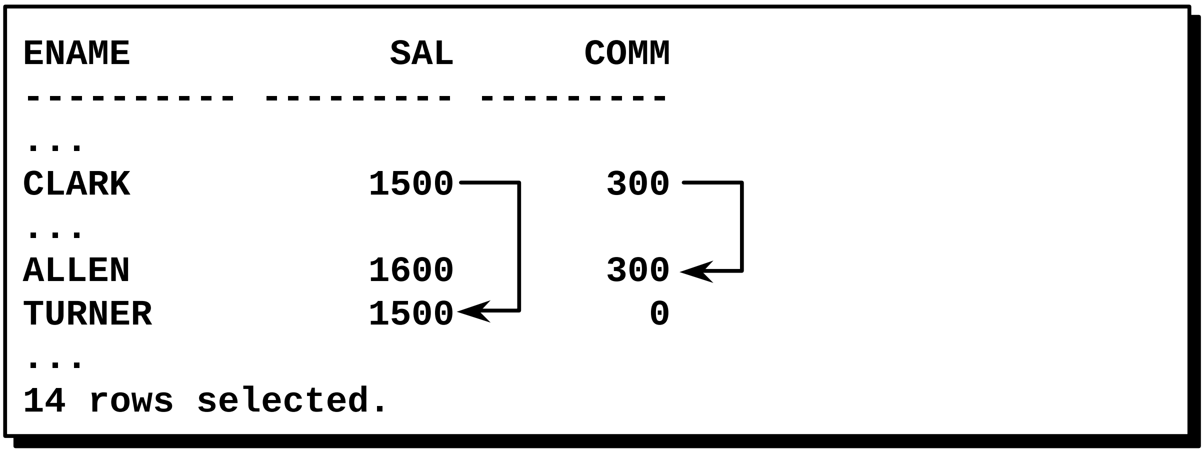
The output of the above SQL statement will be as follows:

```
ENAME          DEPTNO       SAL      COMM
---------- --------- --------- ---------
JAMES              30       950
BLAKE              30      2850
TURNER             30      1500         0
ALLEN              30      1600       300
WARD               30      1250       500
MARTIN             30      1250      1400

6 rows selected.
```

The results of the last two queries were identical even though the comparison conditions were different. The results were obtained because of the specific data in the EMP table.

# Modifying the EMP Table

- **Assume that salary and commission for Clark are modified.**
- **Salary is changed to $1500 and commission to $300.**



```
ENAME                 SAL      COMM
--------- --------- ---------
...
CLARK               1500        300
...
ALLEN               1600        300
TURNER              1500          0
...
14 rows selected.
```

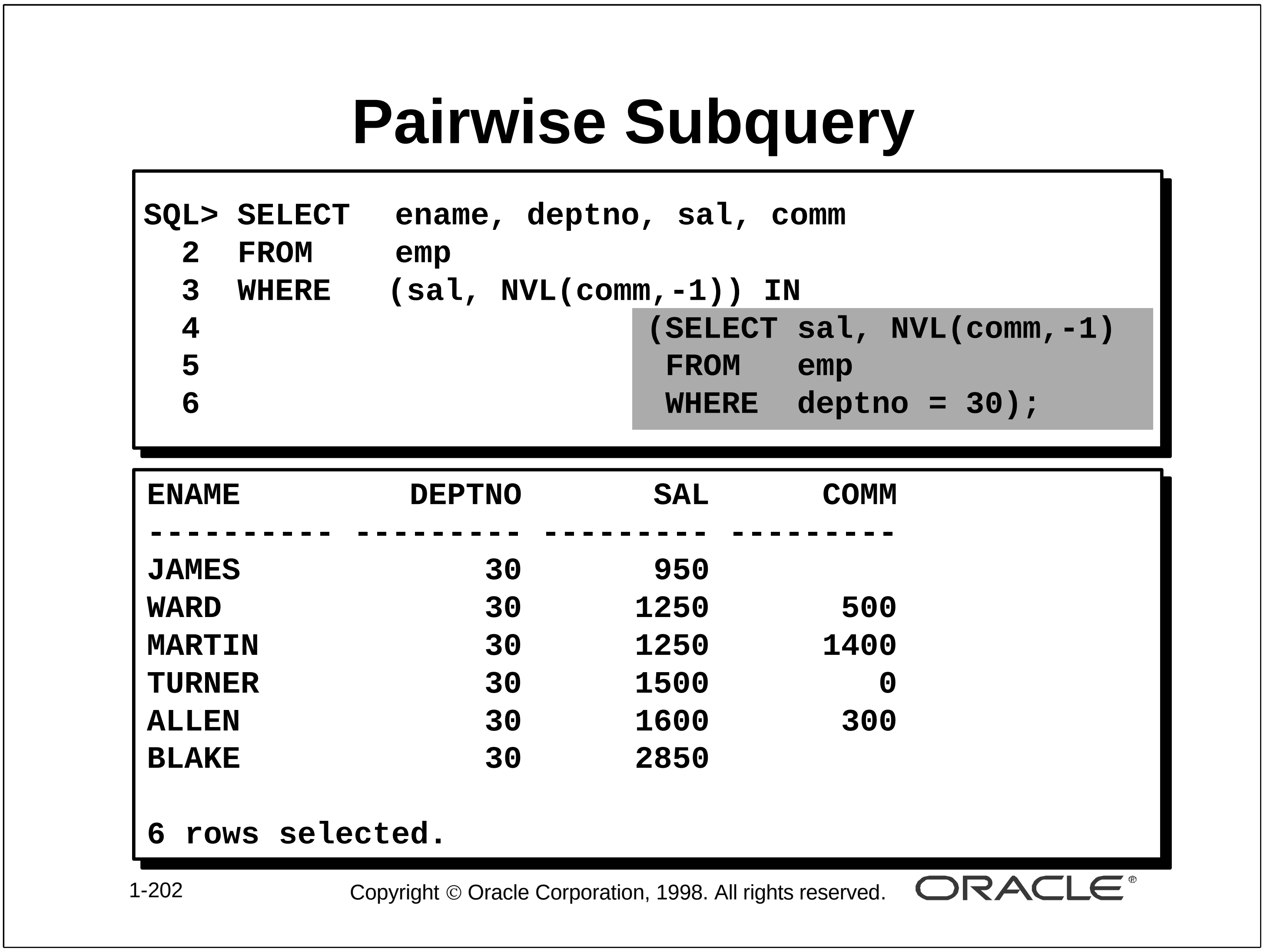
**Example**

Assume that the salary and commission of employee Clark is modified so that he has the same salary as an employee in department 30 and the same commission as a different employee in department 30.

The salary for Clark is now equal to that of Turner ($1500) and the commission for Clark to be equal to that of Allen ($300).

Now run a pairwise and nonpairwise comparison to determine the number of rows returned by each query.

**Note:** The syntax for updating data in a table will be discussed in lesson 9.



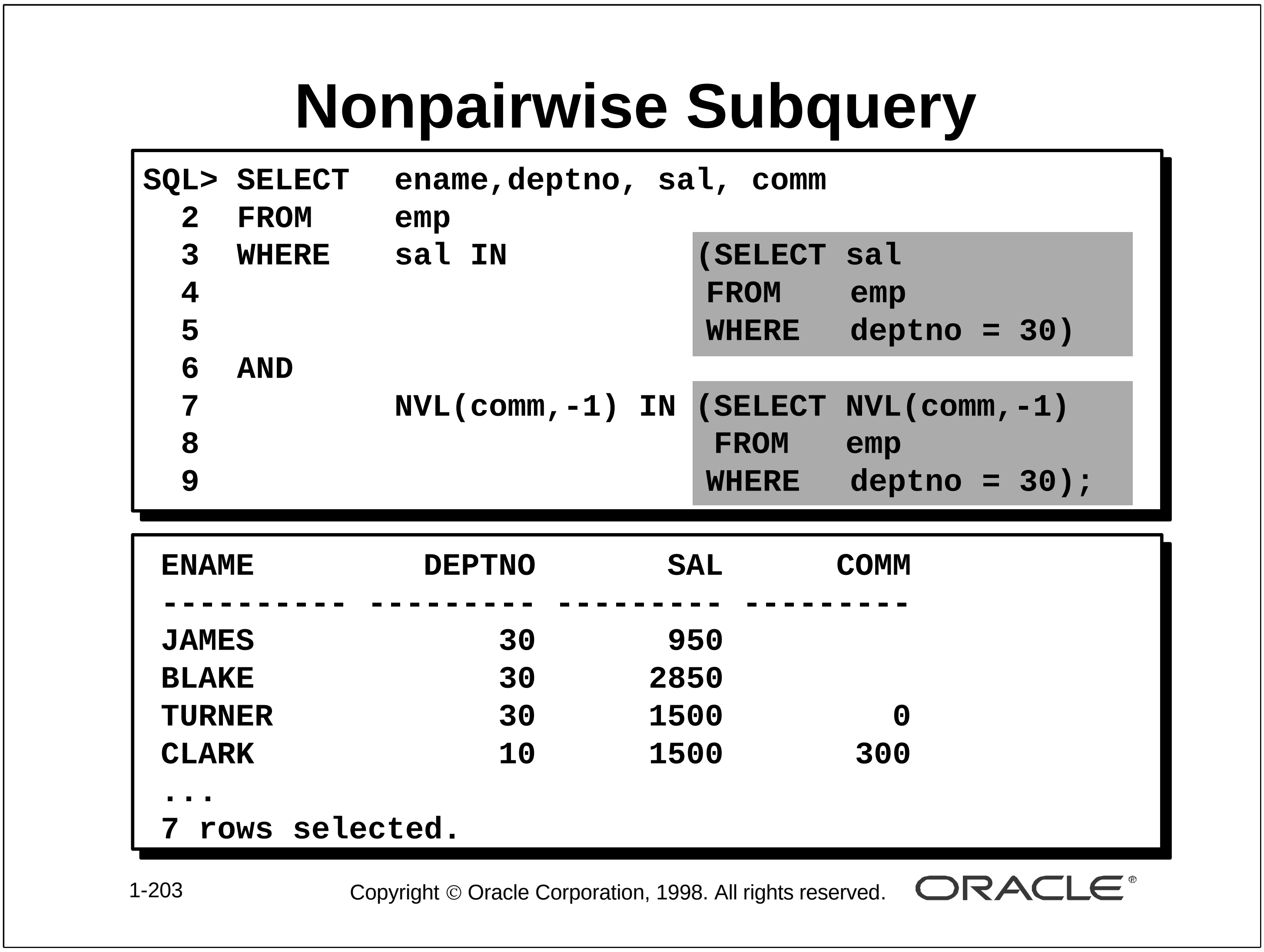
# Pairwise Subquery

```
SQL> SELECT   ename, deptno, sal, comm
  2  FROM     emp
  3  WHERE   (sal, NVL(comm,-1)) IN
  4                            (SELECT sal, NVL(comm,-1)
  5                             FROM   emp
  6                             WHERE  deptno = 30);
```

```
ENAME         DEPTNO       SAL      COMM
---------- --------- --------- ---------
JAMES             30       950
WARD              30      1250       500
MARTIN            30      1250      1400
TURNER            30      1500         0
ALLEN             30      1600       300
BLAKE             30      2850

6 rows selected.
```

### Pairwise Subquery

The output of the pairwise subquery still remains the same and returns six rows.



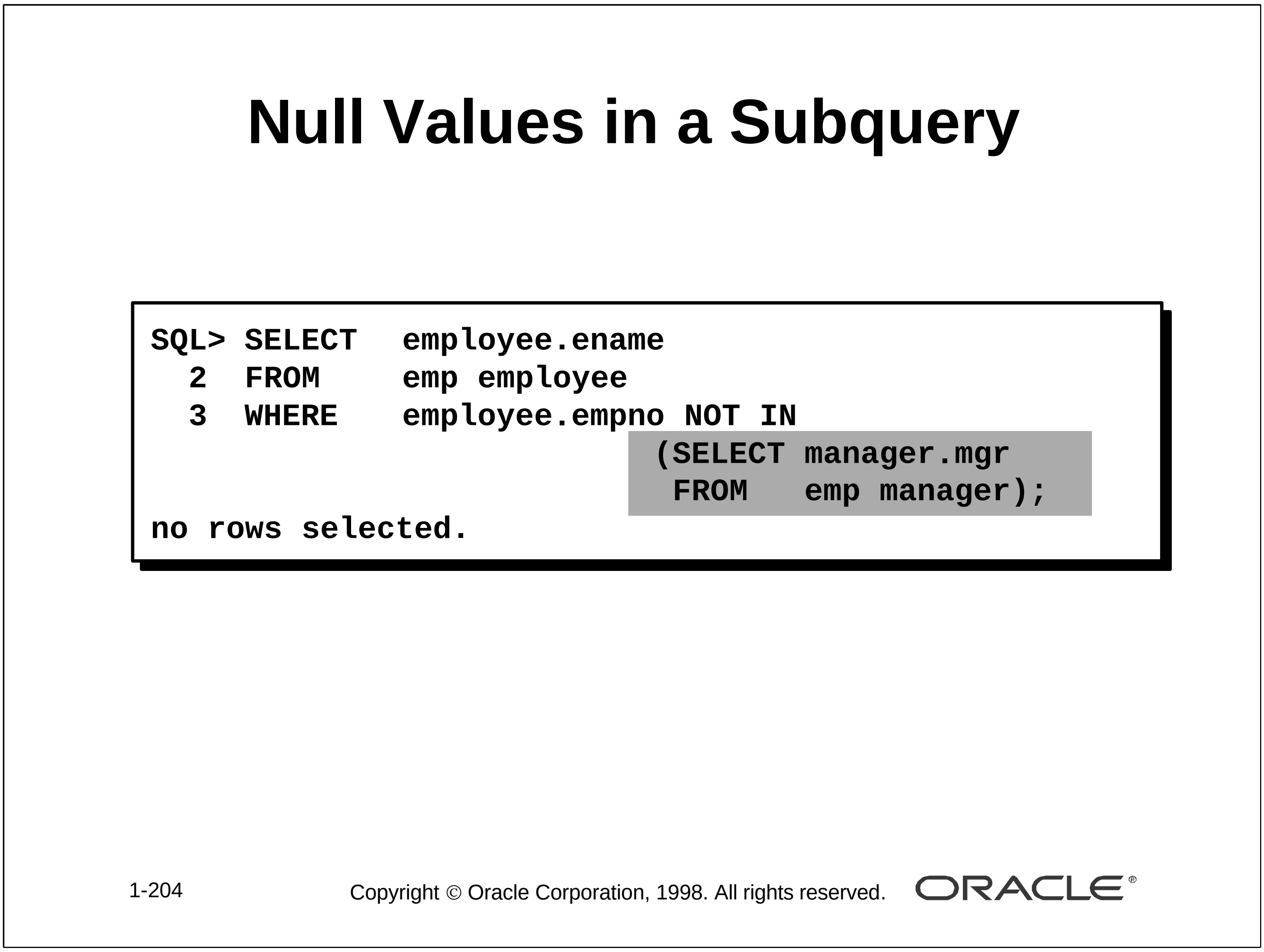
# Nonpairwise Subquery

```
SQL> SELECT   ename,deptno, sal, comm
  2  FROM     emp
  3  WHERE    sal IN             (SELECT sal
  4                               FROM    emp
  5                               WHERE   deptno = 30)
  6  AND
  7           NVL(comm,-1) IN    (SELECT NVL(comm,-1)
  8                                FROM   emp
  9                               WHERE   deptno = 30);
```

```
ENAME         DEPTNO       SAL      COMM
---------- --------- --------- ---------
JAMES             30       950
BLAKE             30      2850
TURNER            30      1500         0
CLARK             10      1500       300
...
7 rows selected.
```

### Nonpairwise Subquery

The results of the nonpairwise subquery include the employee Clark. Clark’s salary is the same as that of Turner and his commission is the same as that of Allen.



# Null Values in a Subquery

```
SQL> SELECT   employee.ename
  2  FROM     emp employee
  3  WHERE    employee.empno NOT IN
                                (SELECT manager.mgr
                                 FROM   emp manager);
no rows selected.
```

1-204

### Returning Nulls in the Resulting Set of a Subquery

The above SQL statement attempts to display all the employees who do not have any subordinates. Logically, this SQL statement should have returned eight rows. However, the SQL statement does not return any rows. One of the values returned by the inner query is a null value and hence the entire query returns no rows. The reason is that all conditions that compare a null value result in a null. So whenever null values are likely to be part of the resultant set of a subquery, do not use the NOT IN operator. The NOT IN operator is equivalent to !=ALL.

Notice that the null value as part of the resultant set of a subquery will not be a problem if you are using the IN operator. The IN operator is equivalent to =ANY. For example, to display the employees who have subordinates, use the following SQL statement:

```
SQL> SELECT    employee.ename
  2  FROM      emp employee
  3  WHERE     employee.empno IN (SELECT manager.mgr
  4                               FROM    emp manager);
```

```
ENAME
----------
KING
…
6 rows selected.
```

# Using a Subquery in the FROM Clause

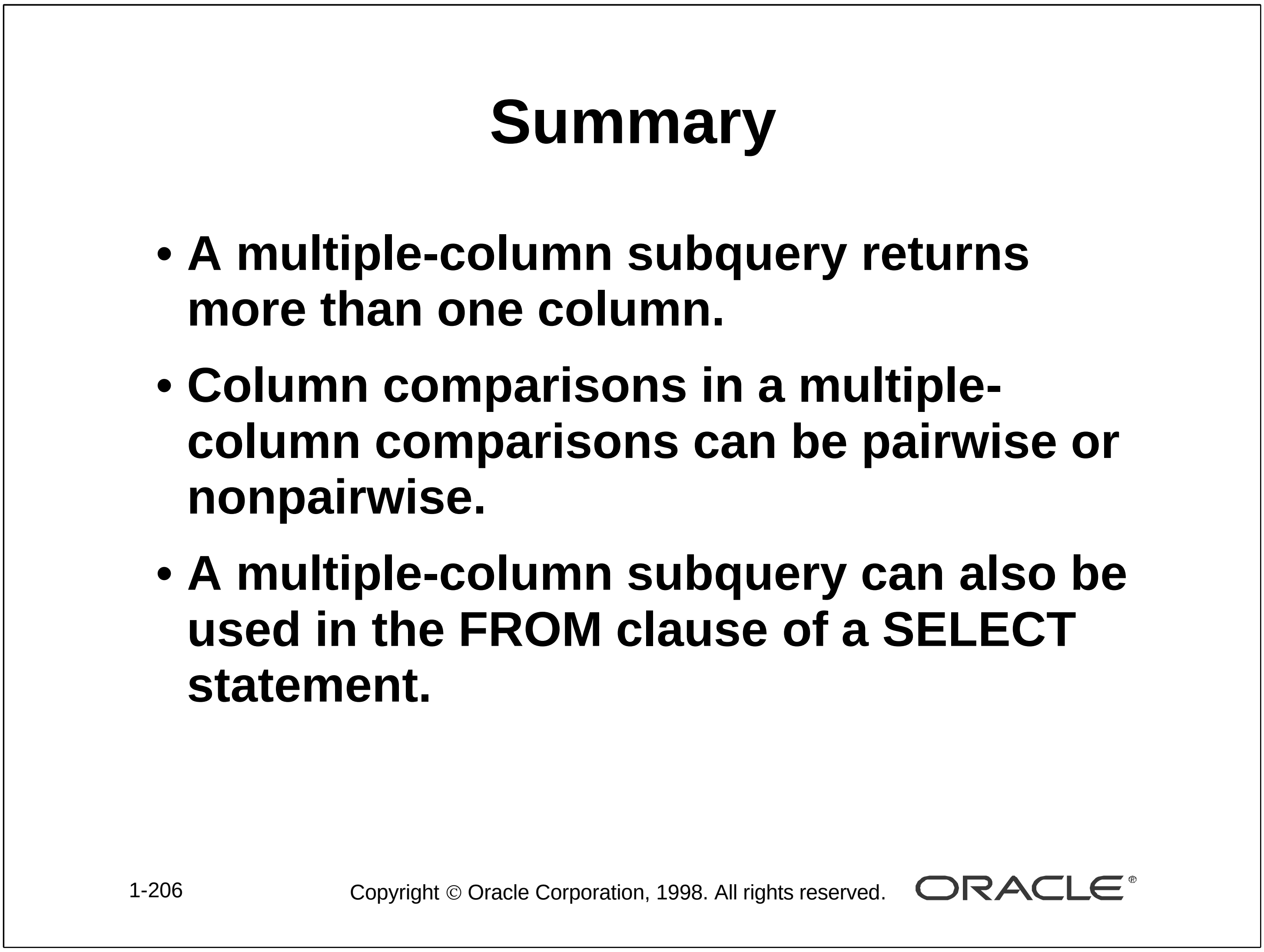
```
SQL> SELECT  a.ename, a.sal, a.deptno, b.salavg
  2  FROM    emp a, (SELECT   deptno, avg(sal) salavg
  3                   FROM     emp
  4                   GROUP BY deptno) b
  5  WHERE   a.deptno = b.deptno
  6  AND     a.sal > b.salavg;
```

```
ENAME             SAL    DEPTNO     SALAVG
---------- --------- --------- ----------
KING             5000        10  2916.6667
JONES            2975        20       2175
SCOTT            3000        20       2175
...
6 rows selected.
```

ORACLE®

## Using a Subquery in the FROM Clause

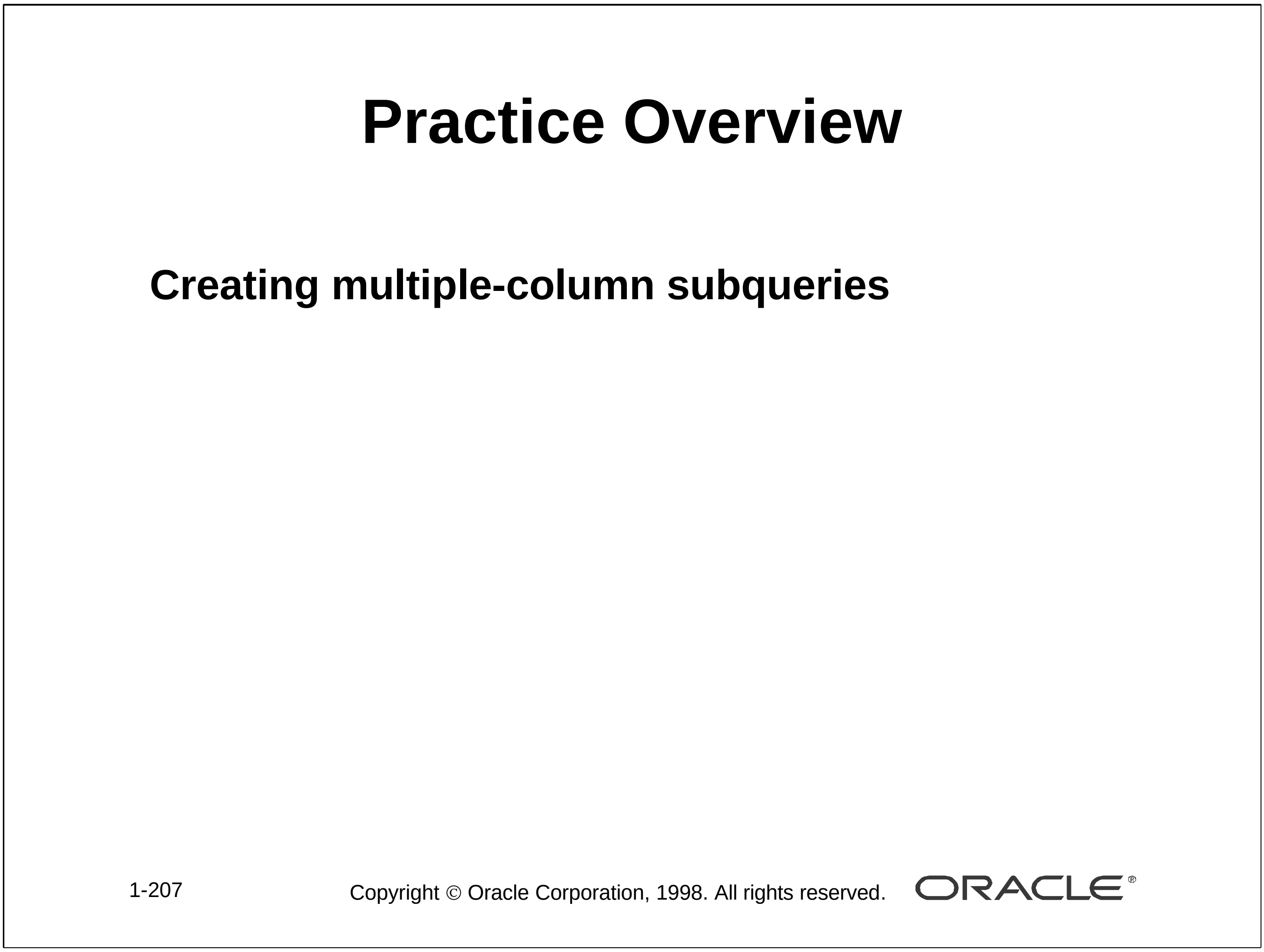
You can use a subquery in the FROM clause of a SELECT statement, which is very similar to how views are used. The example above displays employee names, salaries, department numbers, and average salaries for all the employees who make more than the average salary in their department.



# Summary

- A multiple-column subquery returns more than one column.
- Column comparisons in a multiple-column comparisons can be pairwise or nonpairwise.
- A multiple-column subquery can also be used in the FROM clause of a SELECT statement.

1-206



# Practice Overview

## Creating multiple-column subqueries

1-207

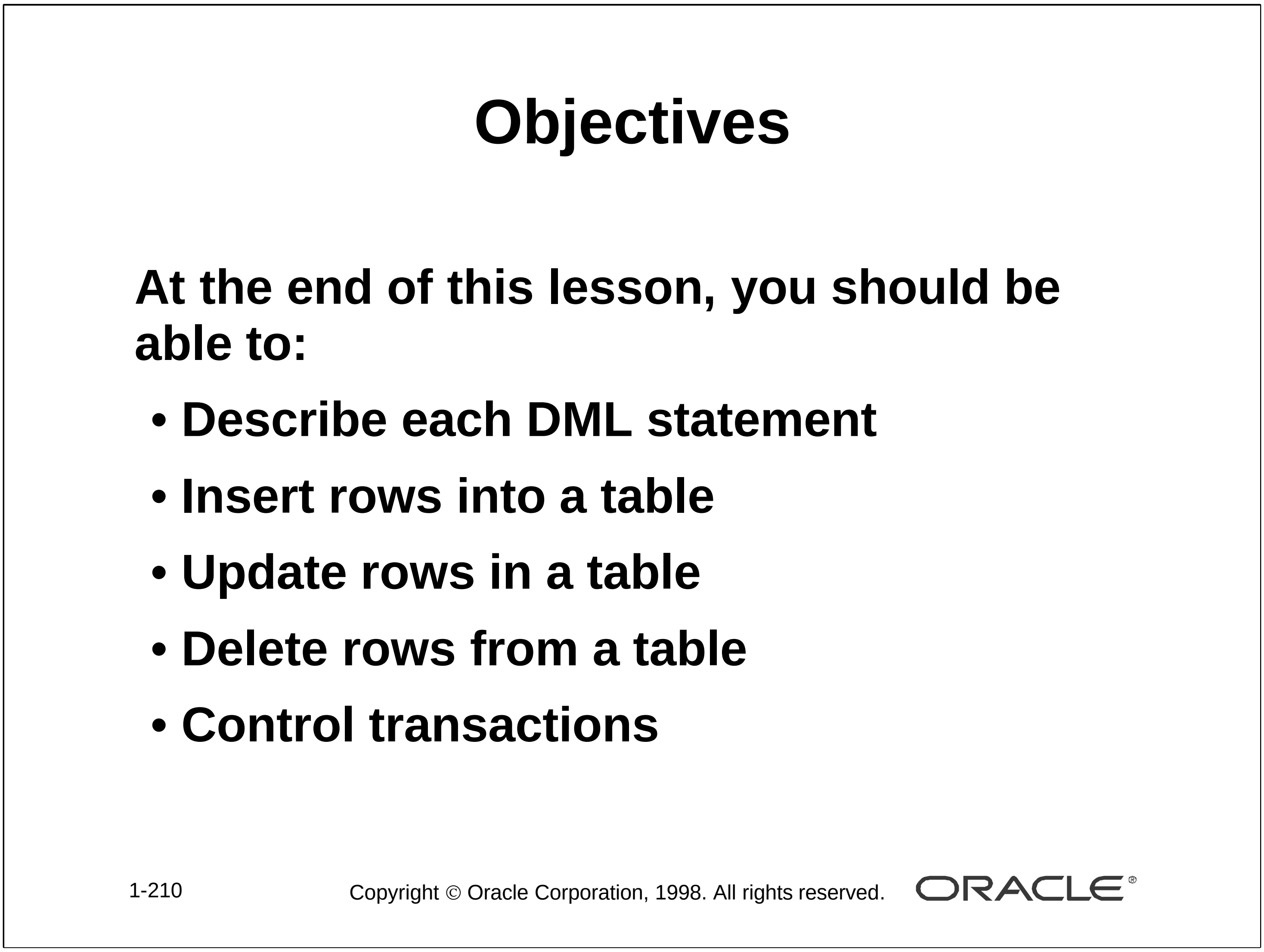
### Practice Overview

In this practice, you will write multiple-value subqueries.

1

# Manipulating Data

| Schedule: | Timing | Topic |
|---|---|---|
| | 40 minutes | Lecture |
| | 30 minutes | Practice |
| | 70 minutes | Total |



# Objectives

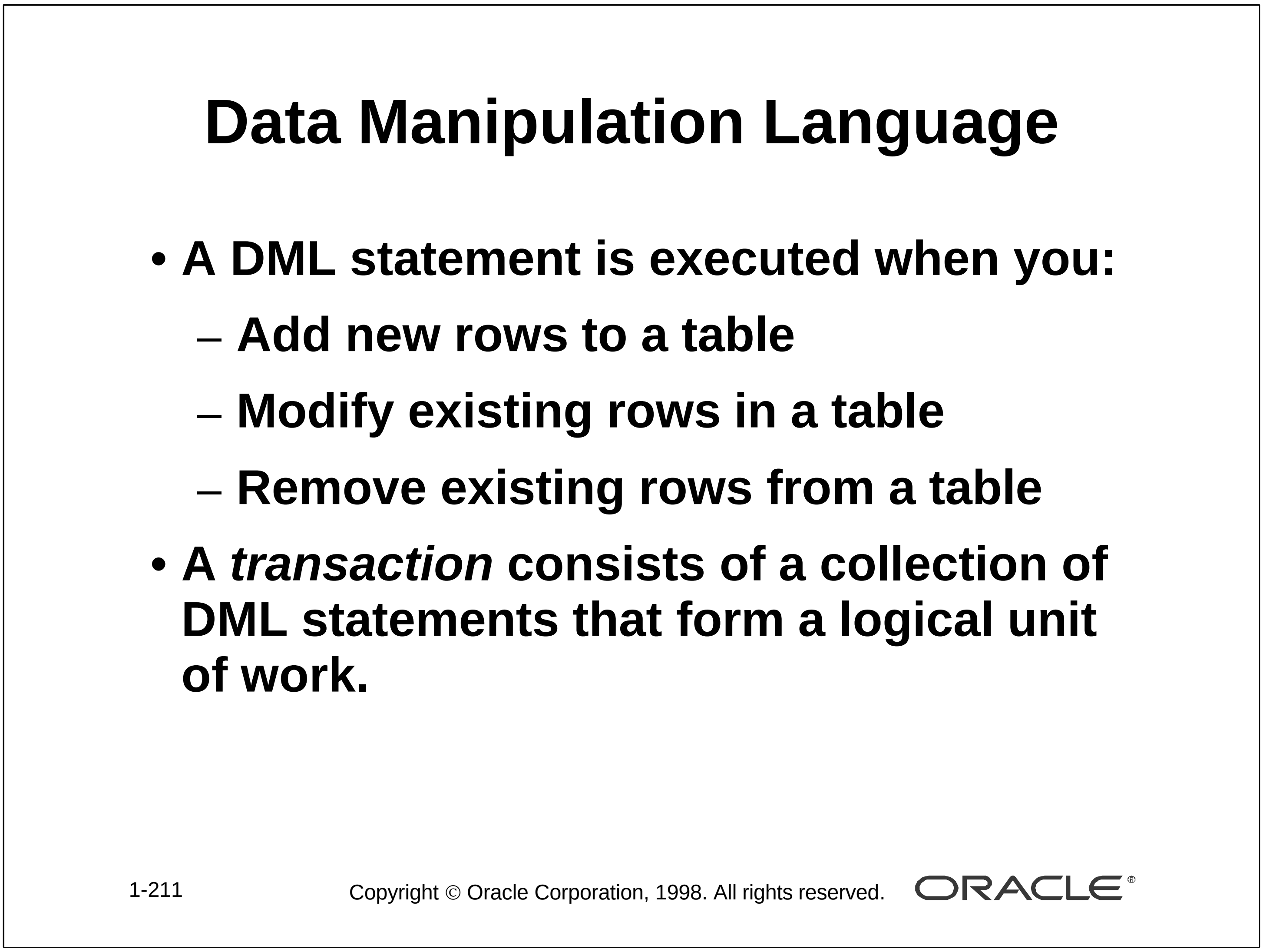
At the end of this lesson, you should be able to:

- Describe each DML statement
- Insert rows into a table
- Update rows in a table
- Delete rows from a table
- Control transactions

1-210

**Lesson Aim**

In this lesson, you will learn how to insert rows into a table, update existing rows in a table, and delete existing rows from a table. You will also learn how to control transactions with the COMMIT, SAVEPOINT, and ROLLBACK statements.



# Data Manipulation Language

- A DML statement is executed when you:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- A *transaction* consists of a collection of DML statements that form a logical unit of work.

1-211

### Data Manipulation Language

Data manipulation language (DML) is a core part of SQL. When you want to add, update, or delete data in the database, you execute a DML statement. A collection of DML statements that form a logical unit of work is called a *transaction***.**

### Class Management Note

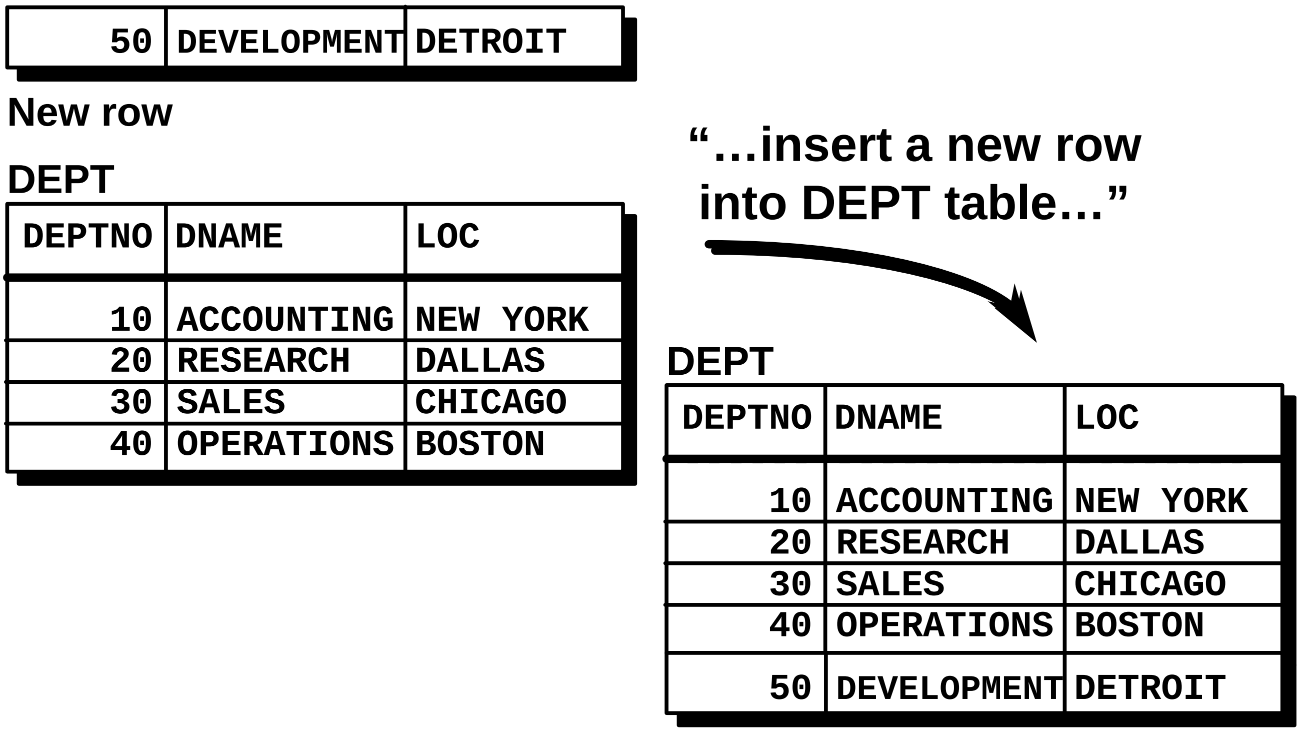
DML statements can be issued directly in SQL*Plus or SQL*DBA, performed automatically by tools such as Developer/2000 or programmed with tools such as the 3GL precompilers.

Every table has INSERT, UPDATE, and DELETE privileges associated with it. These privileges are automatically granted to the creator of the table, but in general they must be explicitly granted to other users.

Starting with Oracle 7.2, you can place a subquery in the place of the table name in an UPDATE statement, essentially the same way a view is used. For example:

```
UPDATE   (SELECT * FROM dept)
 SET      DEPTNO = 90
 WHERE    DEPTNO = 80;
```

# Adding a New Row to a Table



|    |             |         |
|----|-------------|---------|
| 50 | DEVELOPMENT | DETROIT |

New row

DEPT

| DEPTNO | DNAME | LOC |
|---|---|---|
| 10 | ACCOUNTING | NEW YORK |
| 20 | RESEARCH | DALLAS |
| 30 | SALES | CHICAGO |
| 40 | OPERATIONS | BOSTON |

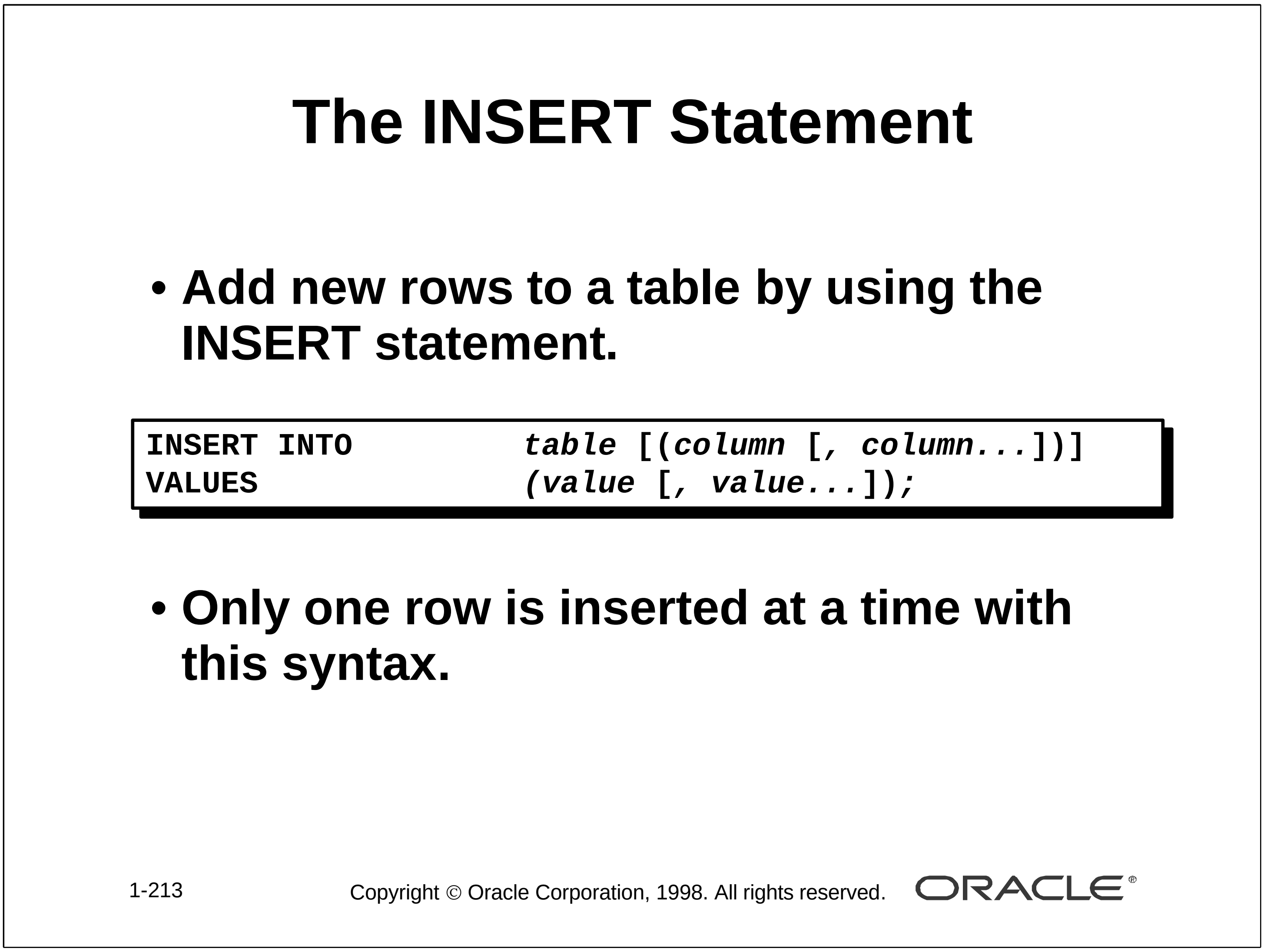
“…insert a new row into DEPT table…”

DEPT

| DEPTNO | DNAME | LOC |
|---|---|---|
| 10 | ACCOUNTING | NEW YORK |
| 20 | RESEARCH | DALLAS |
| 30 | SALES | CHICAGO |
| 40 | OPERATIONS | BOSTON |
| 50 | DEVELOPMENT | DETROIT |

### Adding a New Row to a Table

The graphic above adds a new department to the DEPT table.



# The INSERT Statement

- Add new rows to a table by using the INSERT statement.

```
INSERT INTO         table [(column [, column...])]
VALUES              (value [, value...]);
```

- Only one row is inserted at a time with this syntax.

1-213

ORACLE®

### Adding a New Row to a Table

You can add new rows to a table by issuing the INSERT statement.

In the syntax:

| | |
|---|---|
| *table* | is the name of the table |
| *column* | is the name of the column in the table to populate |
| *value* | is the corresponding value for the column |

**Note:** This statement with the VALUES clause adds only one row at a time to a table.

# Inserting New Rows

- Insert a new row containing values for each column.
- Optionally list the columns in the INSERT clause.

```
SQL> INSERT INTO    dept (deptno, dname, loc)
  2  VALUES         (50, 'DEVELOPMENT', 'DETROIT');
1 row created.
```

- List values in the default order of the columns in the table.
- Enclose character and date values within single quotation marks.

### Adding a New Row to a Table

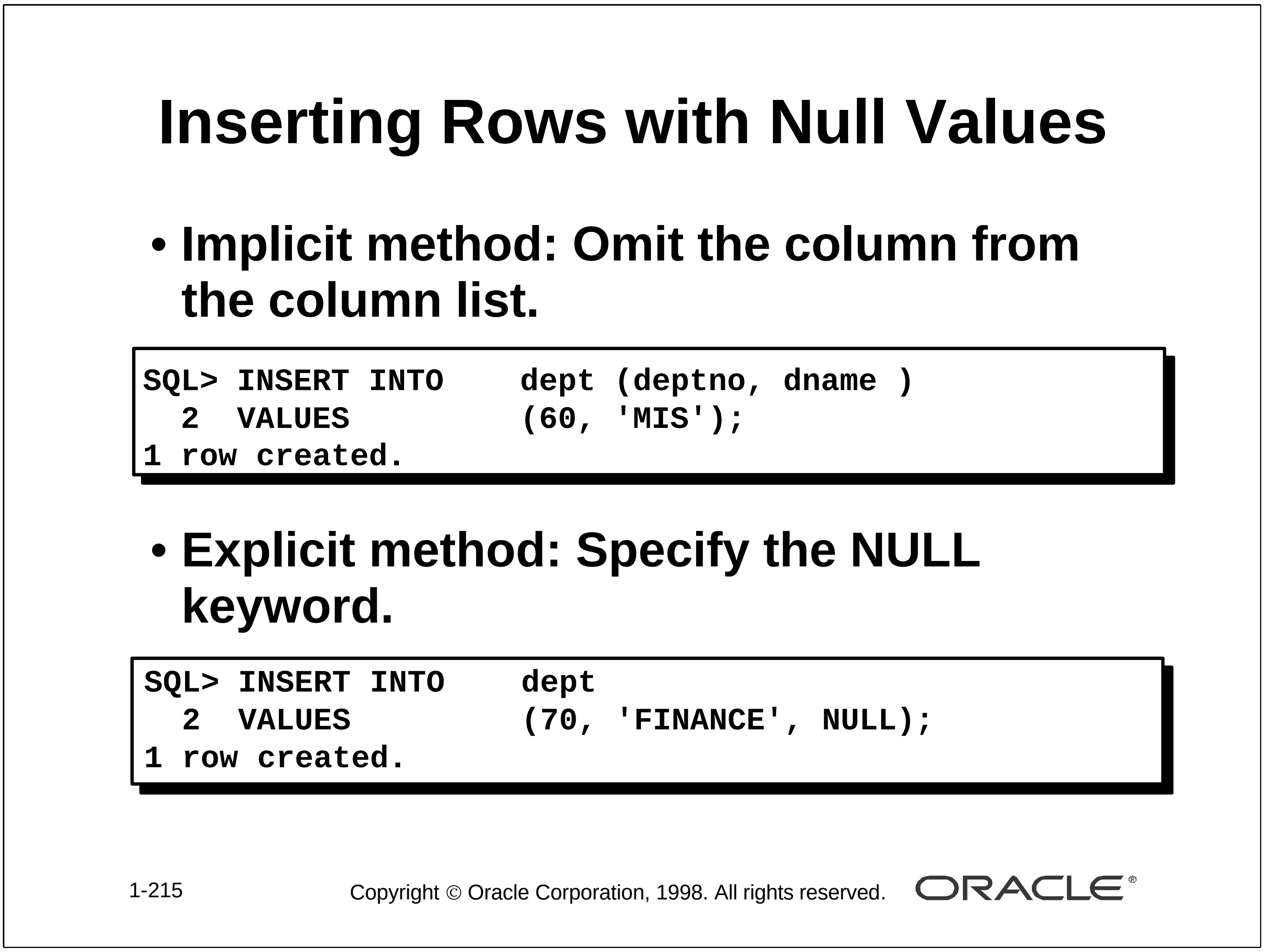
Because you can insert a new row that contains values for each column, the column list is not required in the INSERT clause. However, the values must be listed according to the default order of the columns in the table.

```
SQL> DESCRIBE  dept
```

```
Name                             Null?    Type
-------------------------------- -------- ------------
DEPTNO                           NOT NULL NUMBER(2)
DNAME                                     VARCHAR2(14)
LOC                                       VARCHAR2(13)
```

For clarity, use the column list in the INSERT clause.

Enclose character and date values within single quotation marks; do not enclose numeric values within single quotation marks.



# Inserting Rows with Null Values

- **Implicit method: Omit the column from the column list.**

```
SQL> INSERT INTO    dept (deptno, dname )
  2  VALUES         (60, 'MIS');
1 row created.
```

- **Explicit method: Specify the NULL keyword.**

```
SQL> INSERT INTO    dept
  2  VALUES         (70, 'FINANCE', NULL);
1 row created.
```

1-215

### Methods for Inserting Null Values

| Method | Description |
|---|---|
| Implicit | Omit the column from the column list |
| Explicit | Specify the NULL keyword in the VALUES list<br>Specify the empty string (‘ ’) in the VALUES list; for character strings and dates only |

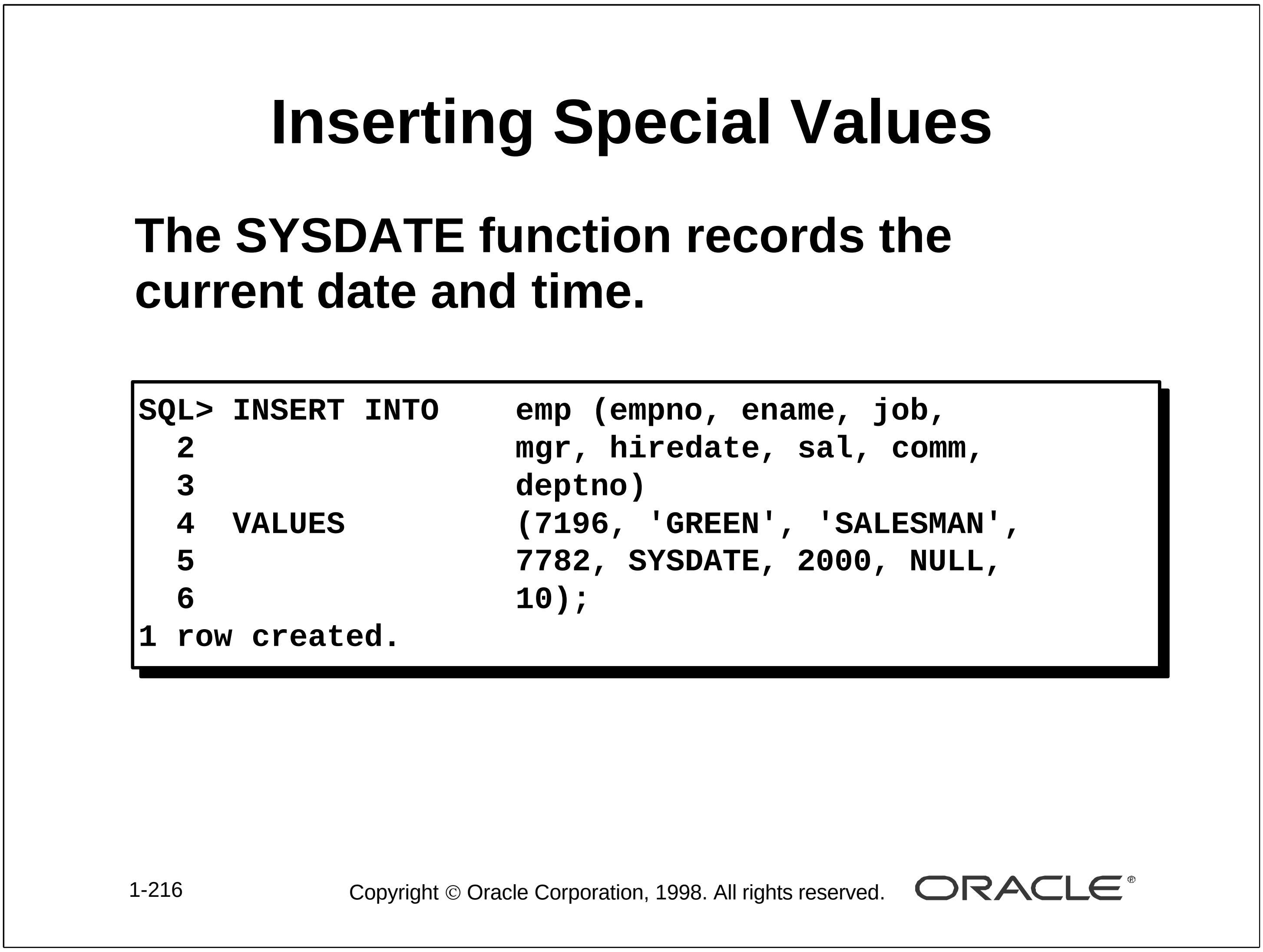
Be sure that the targeted column allows null values by verifying the Null? status from the SQL*Plus DESCRIBE command.

The Oracle8 Server automatically enforces all datatypes, data ranges, and data integrity constraints. Any column that is not listed explicitly obtains a null value in the new row.

### Class Management Note

Common errors that can occur during user input:

- Mandatory value missing for a NOT NULL column
- Duplicate value violates uniqueness constraint
- Foreign key constraint violated
- CHECK constraint violated
- Datatype mismatch
- Value too wide to fit in column



# Inserting Special Values

## The SYSDATE function records the current date and time.

```
SQL> INSERT INTO    emp (empno, ename, job,
  2                 mgr, hiredate, sal, comm,
  3                 deptno)
  4  VALUES         (7196, 'GREEN', 'SALESMAN',
  5                 7782, SYSDATE, 2000, NULL,
  6                 10);
1 row created.
```

1-216

### Inserting Special Values by Using SQL Functions

You can use pseudocolumns to enter special values in your table.

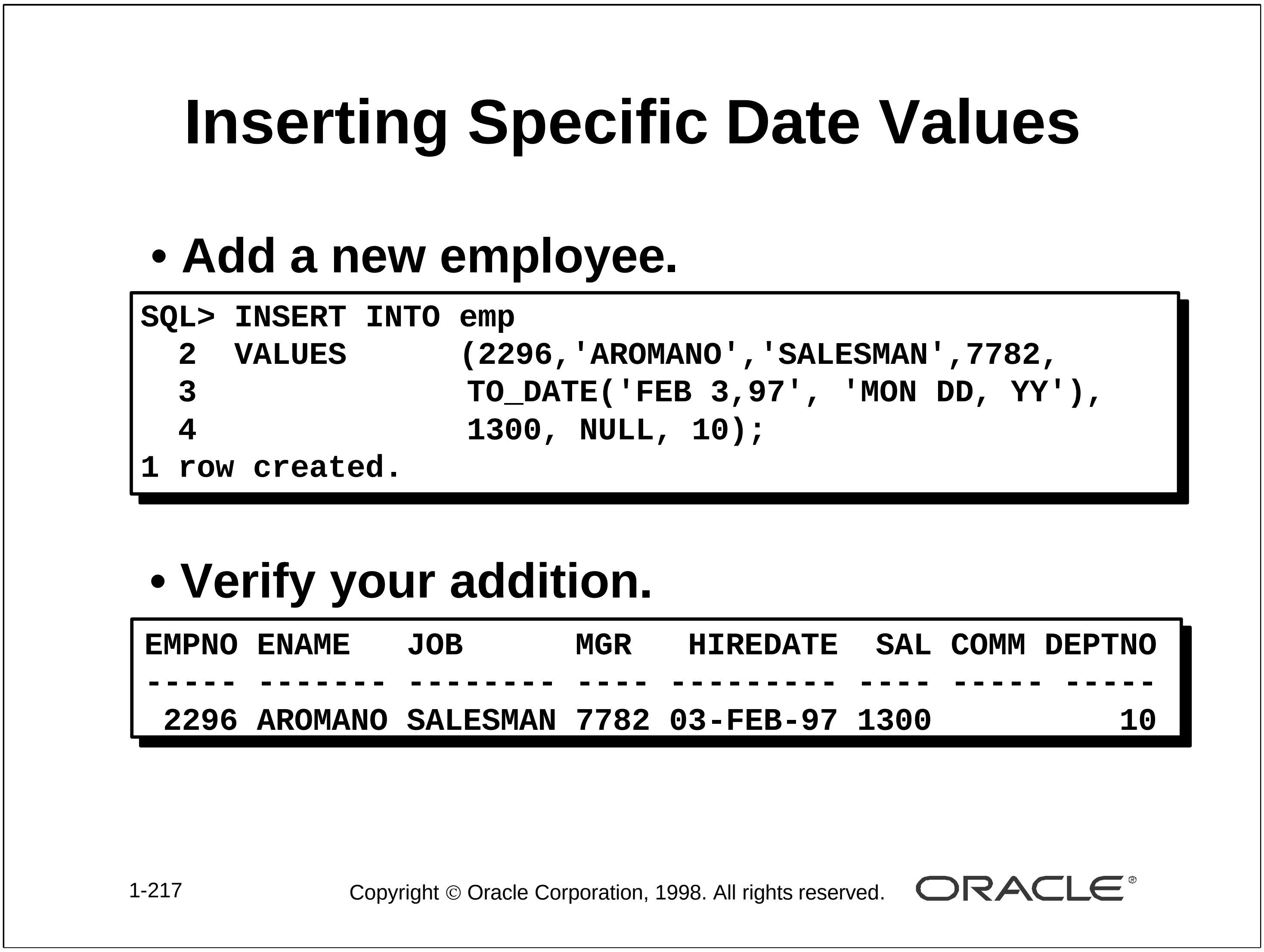
The example above records information for employee Green in the EMP table. It supplies the current date and time in the HIREDATE column. It uses the *SYSDATE* function for current date and time.

You can also use the *USER* function when inserting rows in a table. The USER function records the current username.

### Confirming Additions to the Table

```
SQL> SELECT  empno, ename, job, hiredate, comm
  2  FROM    emp
  3  WHERE   empno = 7196;
```

```
    EMPNO ENAME      JOB       HIREDATE       COMM
--------- ---------- --------- --------- ---------
     7196 GREEN      SALESMAN  01-DEC-97
```



# Inserting Specific Date Values

- Add a new employee.

```
SQL> INSERT INTO emp
  2  VALUES      (2296,'AROMANO','SALESMAN',7782,
  3               TO_DATE('FEB 3,97', 'MON DD, YY'),
  4               1300, NULL, 10);
1 row created.
```

- Verify your addition.

```
EMPNO ENAME   JOB      MGR   HIREDATE  SAL COMM DEPTNO
----- ------- -------- ---- --------- ---- ----- -----
 2296 AROMANO SALESMAN 7782 03-FEB-97 1300          10
```

1-217

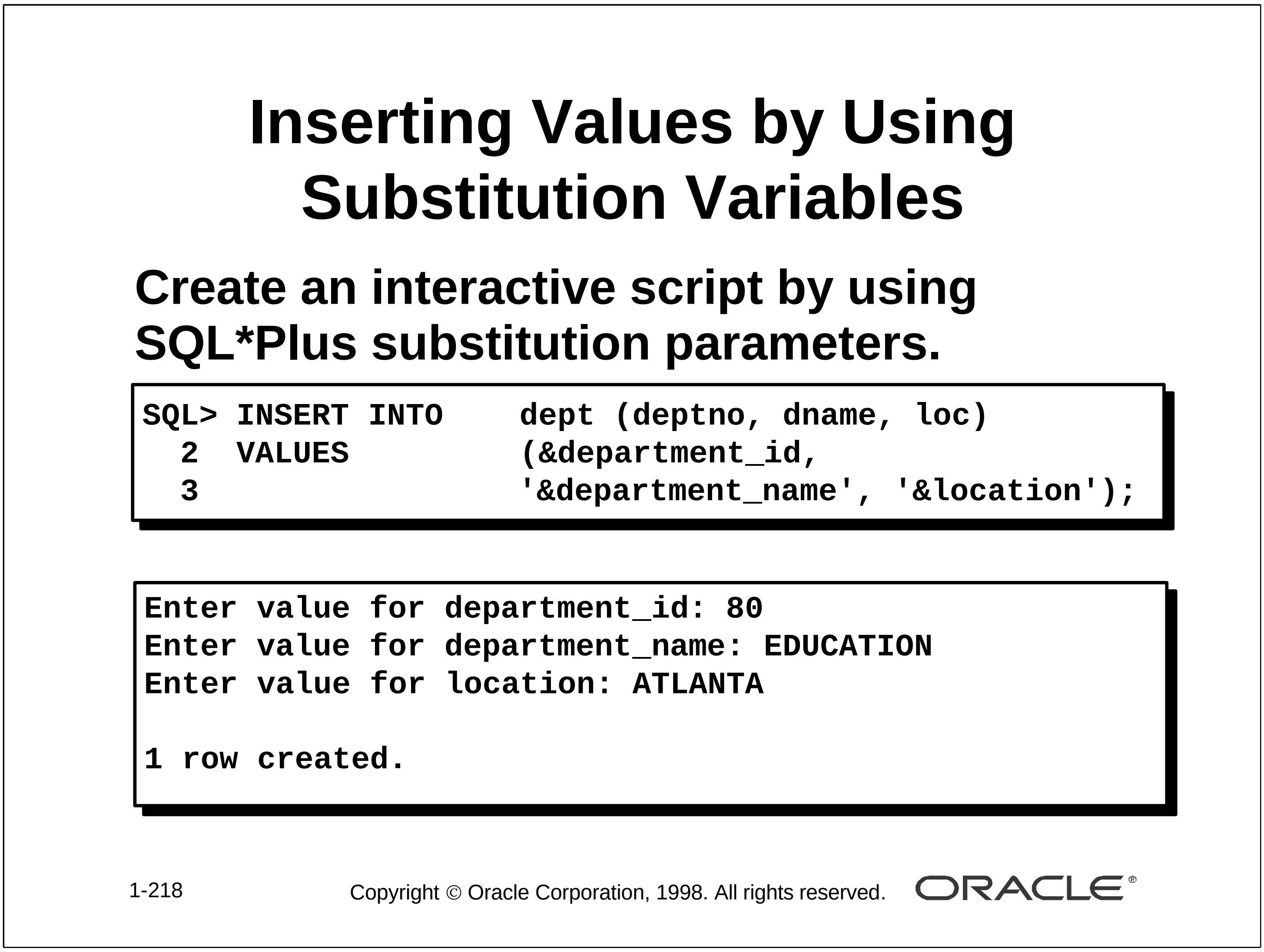
### Inserting Specific Date and Time Values

The format DD-MON-YY is usually used to insert a date value. With this format, recall that the century defaults to the current century. Because the date also contains time information, the default time is midnight (00:00:00).

If a date is required to be entered in another century and a specific time is also required, use the TO_DATE function.

The example on the slide records information for employee Aromano in the EMP table. It sets the HIREDATE column to be February 3, 1997.

If the RR format is set, the century may not be the current one.



# Inserting Values by Using Substitution Variables

## Create an interactive script by using SQL*Plus substitution parameters.

```
SQL> INSERT INTO    dept (deptno, dname, loc)
  2  VALUES         (&department_id,
  3                 '&department_name', '&location');
```

```
Enter value for department_id: 80
Enter value for department_name: EDUCATION
Enter value for location: ATLANTA

1 row created.
```

1-218

### Inserting Values by Using Substitution Variables

You can produce an INSERT statement that allows the user to add values interactively by using SQL*Plus substitution variables.

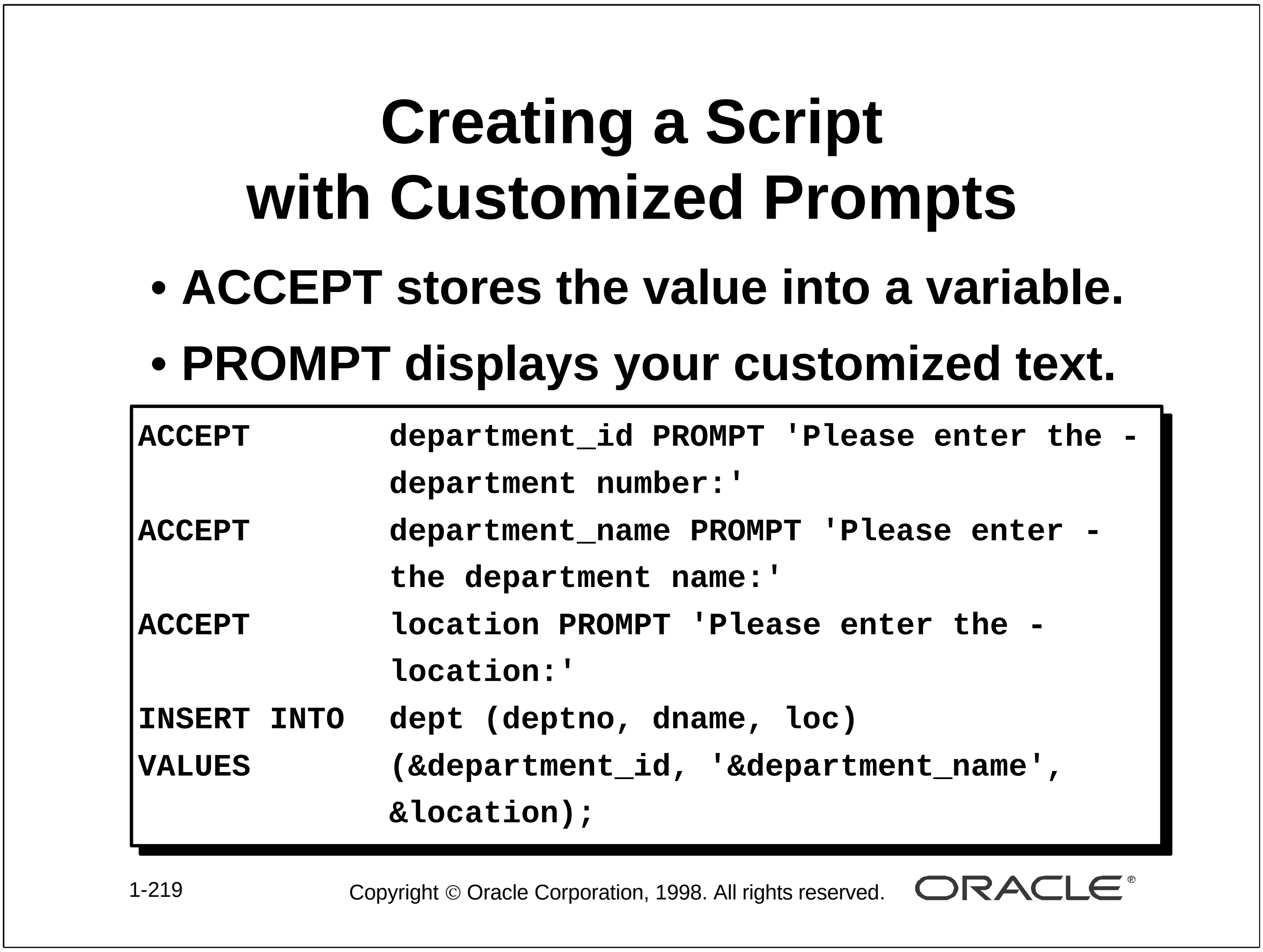
The example above records information for a department in the DEPT table. It prompts the user for the department number, department name, and location.

For date and character values, the ampersand and the variable name are enclosed in single quotation marks.

### Class Management Note

Be sure to mention the following points about the example:

- The names of the SQL*Plus substitution parameters do not have to match the corresponding column names.
- Substitution parameters are lexical variables. Whatever characters the user enters are substituted as text for the variable name.
- The SQL*Plus SET VERIFY command lists the substitution before executing the statement.



# Creating a Script with Customized Prompts

- ACCEPT stores the value into a variable.
- PROMPT displays your customized text.

```
ACCEPT       department_id PROMPT 'Please enter the -
             department number:'
ACCEPT       department_name PROMPT 'Please enter -
             the department name:'
ACCEPT       location PROMPT 'Please enter the -
             location:'
INSERT INTO  dept (deptno, dname, loc)
VALUES       (&department_id, '&department_name',
             &location);
```

1-219

### Creating a Script to Manipulate Data

You can save your command with substitution variables to a file and execute the file. Each time you execute the command, it will prompt you for new values. Customize the prompts by using the SQL*Plus ACCEPT command.

The example on the slide records information for a department in the DEPT table. It prompts the user for the department number, department name, and location by using customized prompt messages.

```
Please enter the department number: 90
Please enter the department name: PAYROLL
Please enter the location: HOUSTON

1 row created.
```

Do not prefix the SQL*Plus substitution parameter with the ampersand (&) when referencing it in the ACCEPT command. Use a dash (-) to continue a SQL*Plus command on the next line.

### Class Management Note

Be sure to mention the following points about the script:

- Do not prefix the SQL*Plus substitution parameter with the ampersand in the ACCEPT command.
- Use a dash to continue a SQL*Plus command on the next line.
- Add a space after the colon in the PROMPT command.

# Copying Rows from Another Table

- Write your INSERT statement with a subquery.

```
SQL> INSERT INTO managers(id, name, salary, hiredate)
  2                   SELECT empno, ename, sal, hiredate
  3                   FROM   emp
  4                   WHERE  job = 'MANAGER';
3 rows created.
```

- Do not use the VALUES clause.
- Match the number of columns in the INSERT clause to those in the subquery.

1-220

### Copying Rows from Another Table

You can use the INSERT statement to add rows to a table where the values are derived from existing tables. In place of the VALUES clause, you use a subquery.

### Syntax

```
INSERT INTO table [ column (, column) ]
                subquery;
```

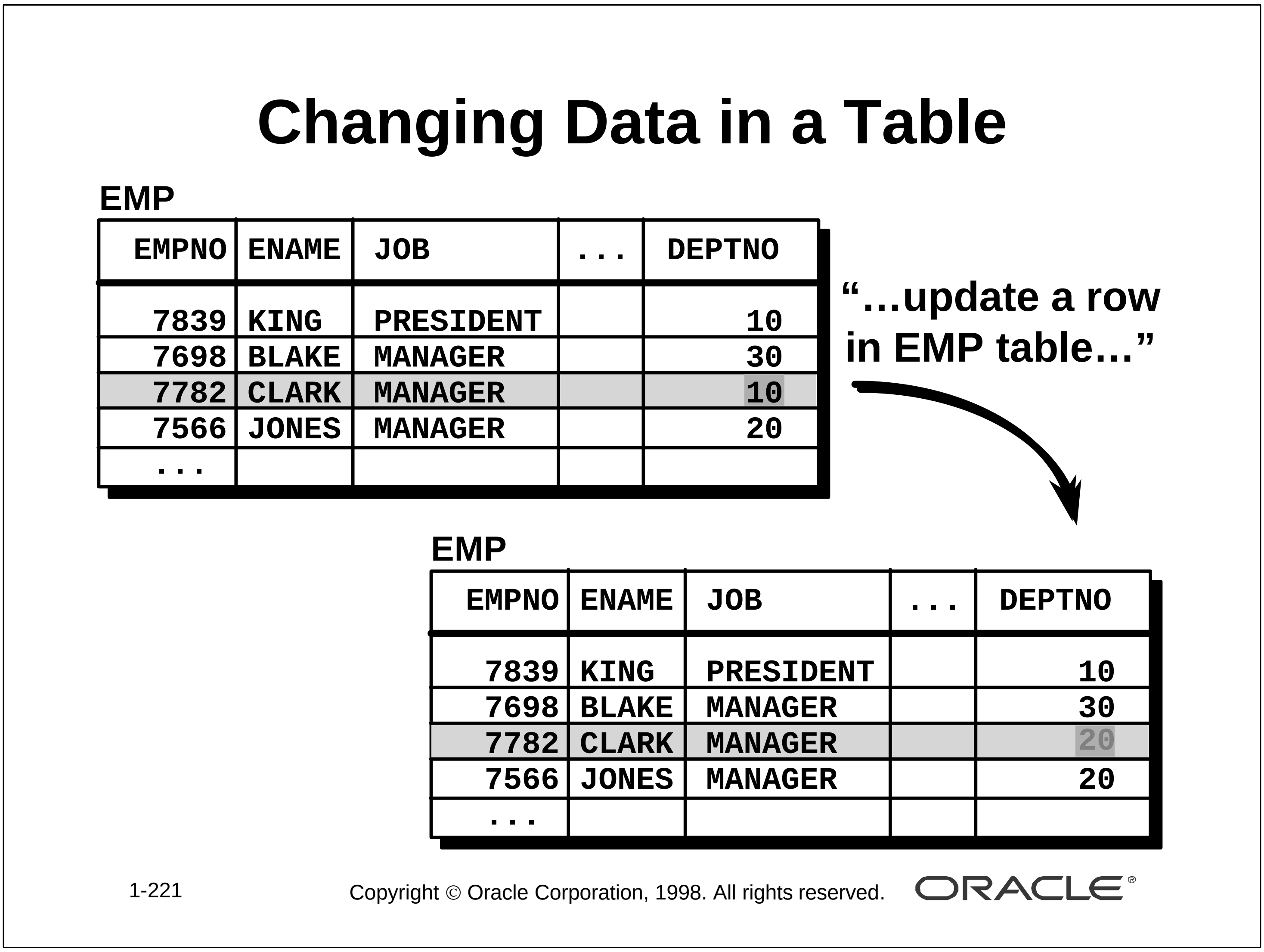
| where: | *table* | is the table name |
|---|---|---|
| | *column* | is the name of the column in the table to populate |
| | *subquery* | is the subquery that returns rows into the table |

For more information, see
*Oracle8 Server SQL Reference, Release 8.0*, “SELECT,” Subqueries section.

The number of columns and their datatypes in the column list of the INSERT clause must match the number of values and their datatypes in the subquery.

### Class Management Note

Do not get into too many details on copying rows from another table.



# Changing Data in a Table

EMP

| EMPNO | ENAME | JOB | ... | DEPTNO |
|---|---|---|---|---|
| 7839 | KING | PRESIDENT | | 10 |
| 7698 | BLAKE | MANAGER | | 30 |
| 7782 | CLARK | MANAGER | | 10 |
| 7566 | JONES | MANAGER | | 20 |
| ... | | | | |

"…update a row in EMP table…"

EMP

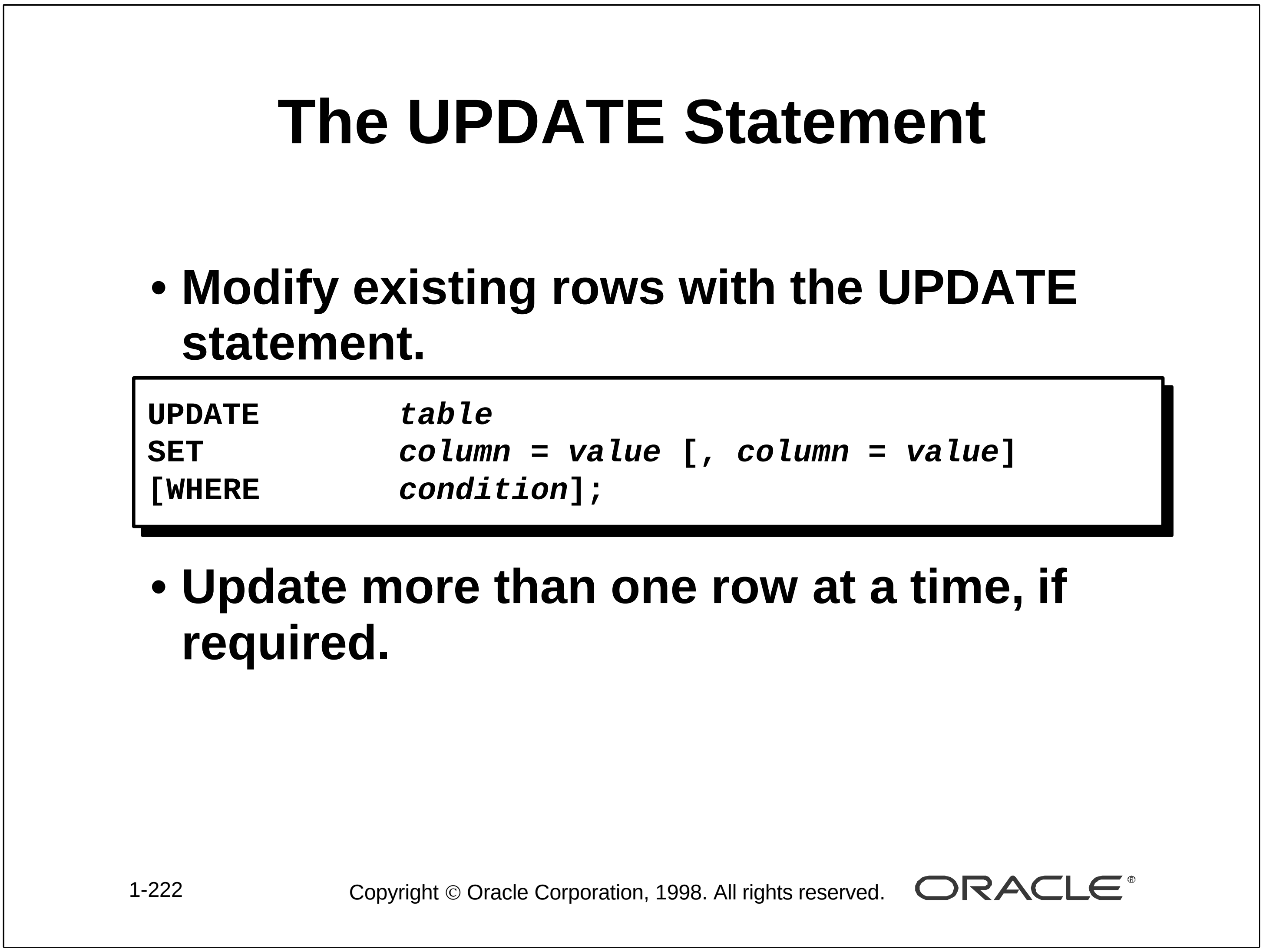
| EMPNO | ENAME | JOB | ... | DEPTNO |
|---|---|---|---|---|
| 7839 | KING | PRESIDENT | | 10 |
| 7698 | BLAKE | MANAGER | | 30 |
| 7782 | CLARK | MANAGER | | 20 |
| 7566 | JONES | MANAGER | | 20 |
| ... | | | | |

1-221

ORACLE®

### Changing Data in a Table

The graphic above changes the department number for Clark from 10 to 20.



# The UPDATE Statement

- Modify existing rows with the UPDATE statement.

```
UPDATE        table
SET           column = value [, column = value]
[WHERE        condition];
```

- Update more than one row at a time, if required.

ORACLE®

### Updating Rows

You can modify existing rows by using the UPDATE statement.

In the above syntax:

*table* is the name of the table

*column* is the name of the column in the table to populate

*value* is the corresponding value or subquery for the column

*condition* identifies the rows to be updated and is composed of column names expressions, constants, subqueries, and comparison operators

Confirm the update operation by querying the table to display the updated rows.

For more information, see
*Oracle8 Server SQL Reference, Release 8.0*, “UPDATE.”

**Note:** In general, use the primary key to identify a single row. Using other columns may unexpectedly cause several rows to be updated. For example, identifying a single row in the EMP table by name is dangerous because more than one employee may have the same name.

### Class Management Note

Demo: *l9sel.sql*, *l9upd.sql*

# Updating Rows in a Table

- Specific row or rows are modified when you specify the WHERE clause.

```
SQL> UPDATE   emp
  2  SET      deptno = 20
  3  WHERE    empno = 7782;
1 row updated.
```

- All rows in the table are modified if you omit the WHERE clause.

```
SQL> UPDATE   employee
  2  SET      deptno = 20;
14 rows updated.
```

### Updating Rows

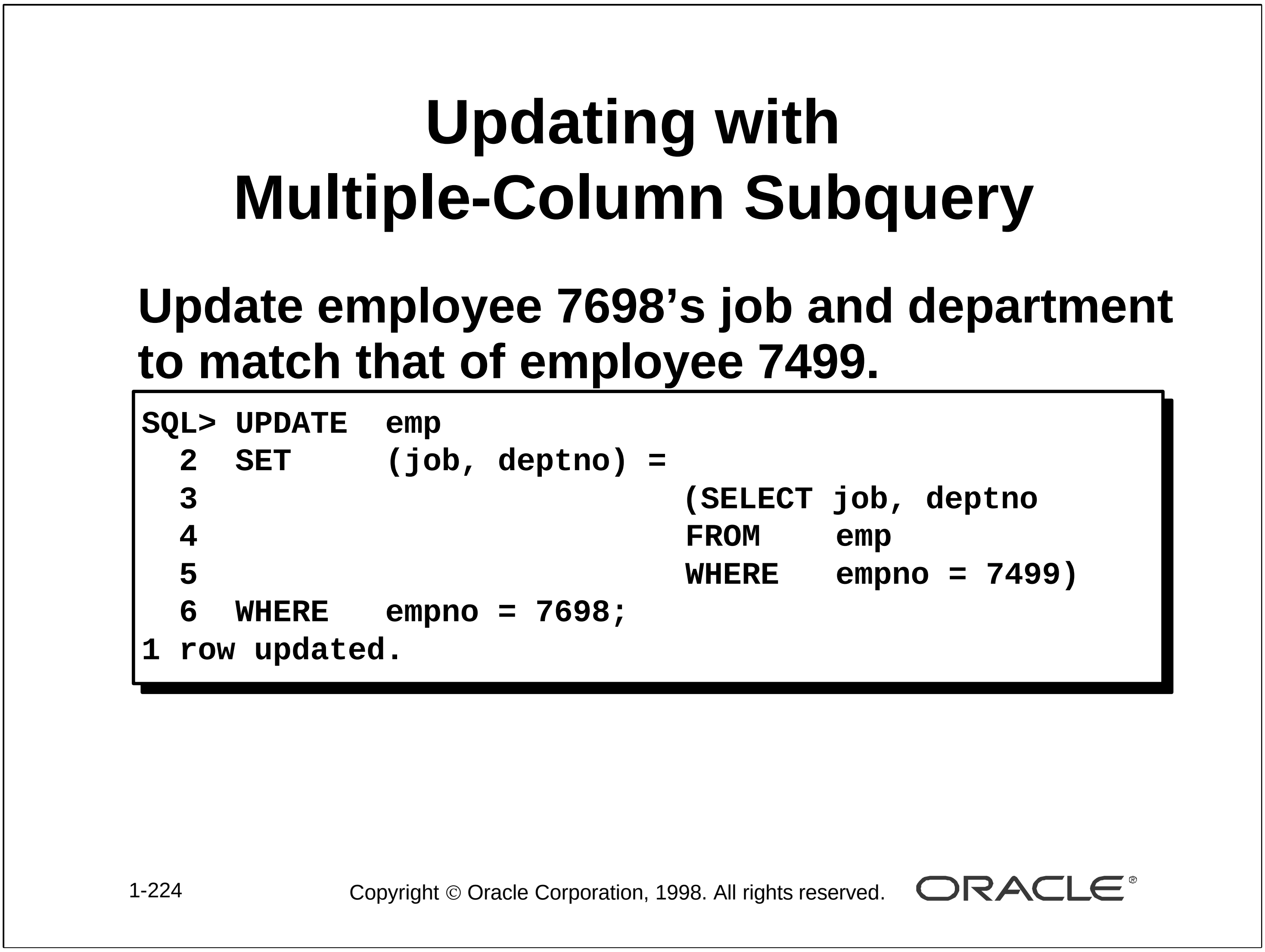
The UPDATE statement modifies specific row(s), if the WHERE clause is specified. The example above transfers employee 7782 (Clark) to department 20.

If you omit the WHERE clause, all the rows in the table are modified.

```
SQL> SELECT  ename, deptno
  2  FROM    employee;
```

```
ENAME         DEPTNO
---------- ---------
KING              20
BLAKE             20
CLARK             20
JONES             20
MARTIN            20
ALLEN             20
TURNER            20
...
14 rows selected.
```

**Note:** The EMPLOYEE table has the same data as the EMP table.



# Updating with Multiple-Column Subquery

## Update employee 7698's job and department to match that of employee 7499.

```
SQL> UPDATE   emp
  2  SET      (job, deptno) =
  3                              (SELECT job, deptno
  4                              FROM    emp
  5                              WHERE   empno = 7499)
  6  WHERE    empno = 7698;
1 row updated.
```

### Updating Rows with a Multiple-Column Subquery

Multiple-column subqueries can be implemented in the SET clause of an UPDATE statement.

### Syntax

```
UPDATE   table
SET      (column, column, ...) =
                   (SELECT   column, column,
                    FROM     table
                    WHERE    condition)
WHERE    condition;
```

### Class Management Note

It may be worth showing participants that the results would be the same for the example above if two different subqueries were used in the SET clause as illustrated below:

```
UPDATE       emp
SET          job =  (SELECT  job FROM emp
                      WHERE   empno = 7499),
             deptno = (SELECT deptno FROM dept
                        WHERE  dname = 'SALES')
WHERE        empno = 7698;
```

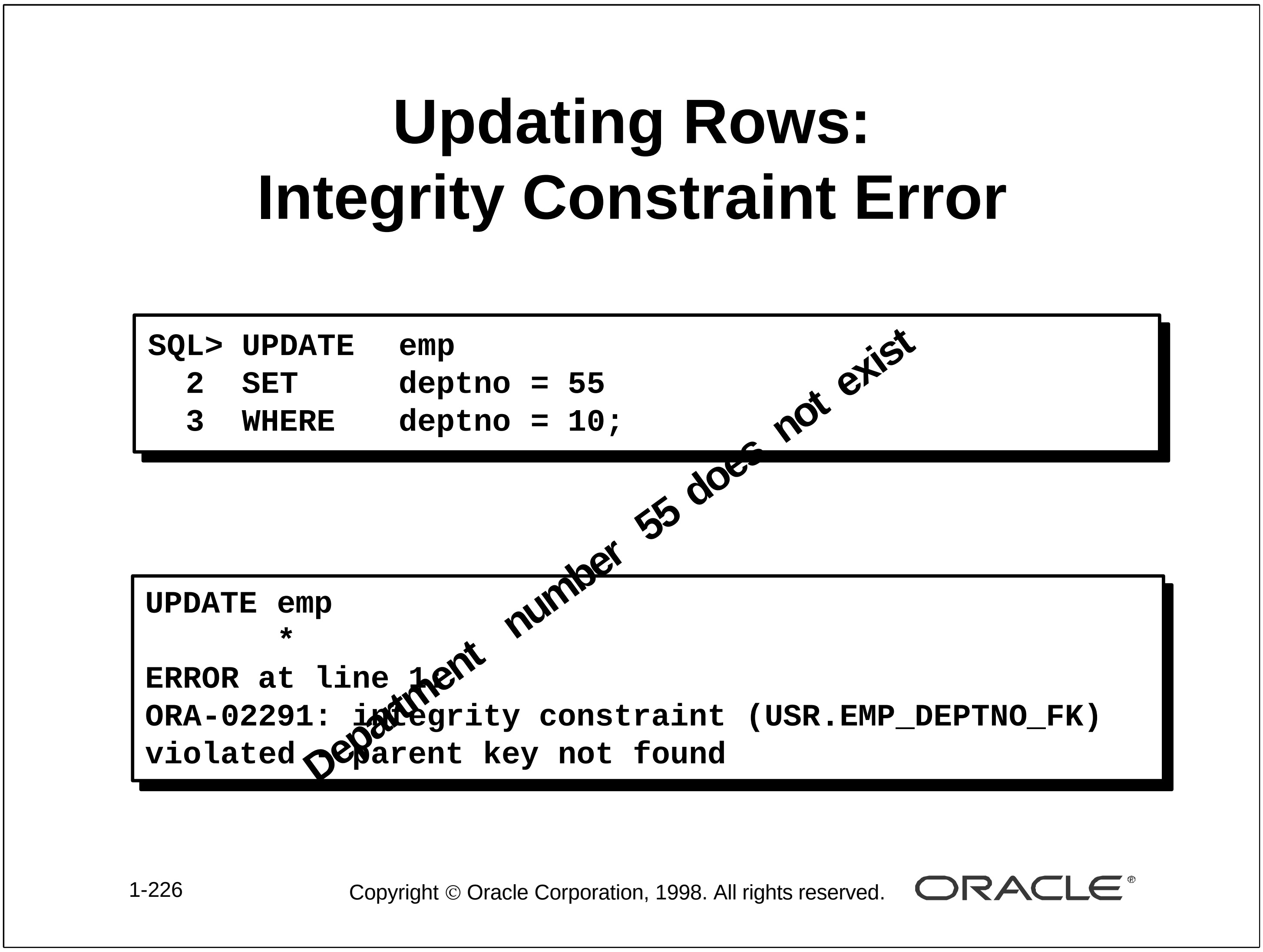
# Updating Rows Based on Another Table

## Use subqueries in UPDATE statements to update rows in a table based on values from another table.

```
SQL> UPDATE  employee
  2  SET     deptno =  (SELECT   deptno
  3                     FROM     emp
  4                     WHERE    empno = 7788)
  5  WHERE   job    =  (SELECT   job
  6                     FROM     emp
  7                     WHERE    empno = 7788);
2 rows updated.
```

### Updating Rows Based on Another Table

You can use subqueries in UPDATE statements to update rows in a table. The example above updates the EMPLOYEE table based on the values from the EMP table. It changes the department number of all employees with employee  7788's job title to employee 7788's current department number.



# Updating Rows: Integrity Constraint Error

```
SQL> UPDATE   emp
  2  SET      deptno = 55
  3  WHERE    deptno = 10;
```

Department number 55 does not exist

```
UPDATE emp
       *
ERROR at line 1:
ORA-02291: integrity constraint (USR.EMP_DEPTNO_FK)
violated - parent key not found
```

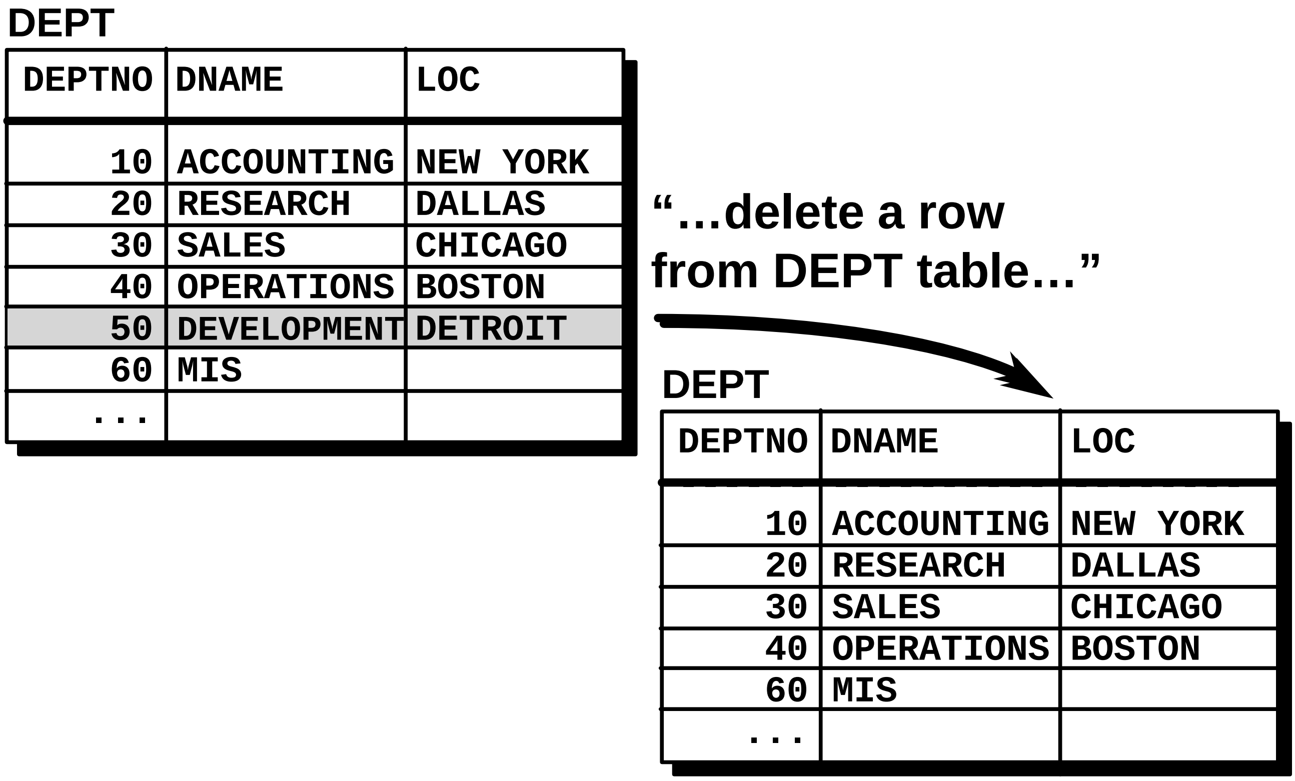
### Integrity Constraint Error

If you attempt to update a record with a value that is tied to an integrity constraint, you will experience an error.

In the example above, department number 55 does not exist in the parent table, DEPT, and so you receive the “parent key” violation ORA-02291.

**Note:** Integrity constraints assure that the data adheres to a predefined set of rules. Lesson 11 will cover integrity constraints in greater depth.

# Removing a Row from a Table



DEPT

| DEPTNO | DNAME | LOC |
|---|---|---|
| 10 | ACCOUNTING | NEW YORK |
| 20 | RESEARCH | DALLAS |
| 30 | SALES | CHICAGO |
| 40 | OPERATIONS | BOSTON |
| 50 | DEVELOPMENT | DETROIT |
| 60 | MIS | |
| ... | | |

"…delete a row from DEPT table…"

DEPT

| DEPTNO | DNAME | LOC |
|---|---|---|
| 10 | ACCOUNTING | NEW YORK |
| 20 | RESEARCH | DALLAS |
| 30 | SALES | CHICAGO |
| 40 | OPERATIONS | BOSTON |
| 60 | MIS | |
| ... | | |

### Removing a Row from a Table

The graphic above removes the DEVELOPMENT department from the DEPT table (assuming there are no constraints defined on the DEPT table).

### Class Management Note

After all the rows have been eliminated with the DELETE statement, only the data structure of the table remains. A more efficient method of emptying a table is with the TRUNCATE statement.
You can use the TRUNCATE statement to quickly remove all rows from a table or cluster. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information. It will be covered in lesson 10.
- Truncating a table does not fire the table's DELETE triggers.
- If the table is the parent of a referential integrity constraint, you cannot truncate the table. Disable the constraint before issuing the TRUNCATE statement.

# The DELETE Statement

## You can remove existing rows from a table by using the DELETE statement.

```
DELETE [FROM]   table
[WHERE          condition];
```

### Deleting Rows

You can remove existing rows by using the DELETE statement.

In the syntax:

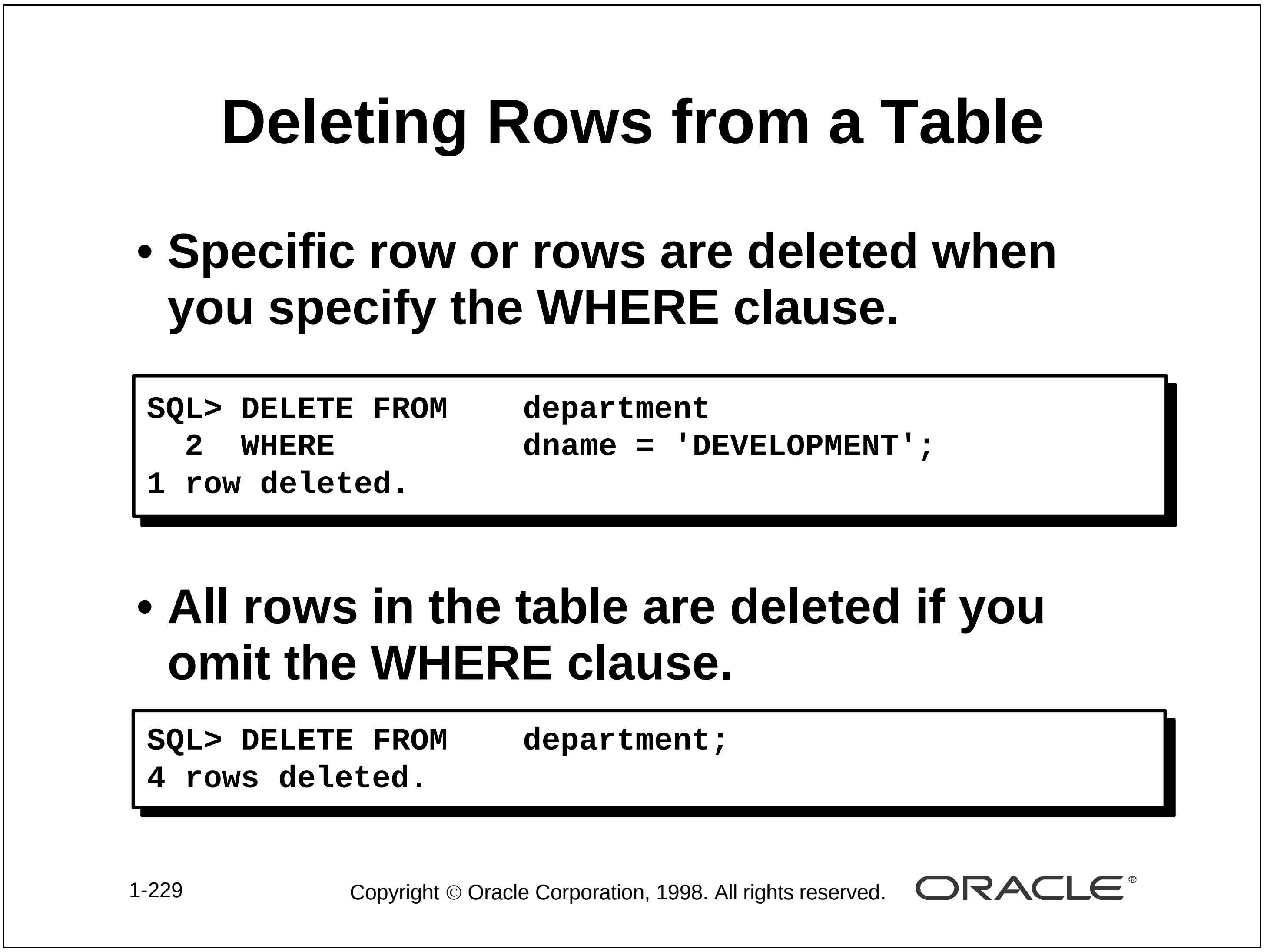
*table* is the table name

*condition* identifies the rows to be deleted and is composed of column names, expressions, constants, subqueries, and comparison operators

For more information, see
*Oracle8 Server SQL Reference, Release 8.0*, “DELETE.”

### Class Management Note

The DELETE statement does not ask for confirmation. However, the delete operation is not made permanent until the data transaction is committed. Therefore, you can undo the operation with the ROLLBACK statement if you make a mistake.



# Deleting Rows from a Table

- Specific row or rows are deleted when you specify the WHERE clause.

```
SQL> DELETE FROM    department
  2  WHERE          dname = 'DEVELOPMENT';
1 row deleted.
```

- All rows in the table are deleted if you omit the WHERE clause.

```
SQL> DELETE FROM    department;
4 rows deleted.
```

1-229

### Deleting Rows

You can delete specific row(s) by specifying the WHERE clause in the DELETE statement. The example above deletes the DEVELOPMENT department from the DEPARTMENT table. You can confirm the delete operation by displaying the deleted rows by using the SELECT statement.

```
SQL> SELECT  *
  2  FROM    department
  3  WHERE   dname = 'DEVELOPMENT';
no rows selected.
```

### Example

Remove all employees who started after January 1, 1997.

```
SQL> DELETE FROM  emp
  2  WHERE        hiredate > TO_DATE('01.01.97', 'DD.MM.YY');
1 row deleted.
```

If you omit the WHERE clause, all rows in the table will be deleted. The second example on the slide deletes all the rows from the DEPARTMENT table because no WHERE clause had been specified.

**Note:** The DEPARTMENT table has the same data as the DEPT table.

# Deleting Rows Based on Another Table

## Use subqueries in DELETE statements to remove rows from a table based on values from another table.

```
SQL> DELETE FROM    employee
  2  WHERE          deptno =
  3                        (SELECT   deptno
  4                         FROM     dept
  5                         WHERE    dname ='SALES');
6 rows deleted.
```

### Deleting Rows Based on Another Table

You can use subqueries to delete rows from a table based on values from another table. The example above deletes all the employees who are in department 30. The subquery searches the DEPT table to find the department number for the SALES department. The subquery then feeds the department number to the main query, which deletes rows of data from the EMPLOYEE table based on this department number.

# Deleting Rows: Integrity Constraint Error

```
SQL> DELETE FROM    dept
  2  WHERE          deptno = 10;
```

```
DELETE FROM dept
            *
ERROR at line 1:
ORA-02292: integrity constraint (USR.EMP_DEPTNO_FK)
violated - child record found
```

You cannot delete a row that contains a primary key that is used as a foreign key in another table.

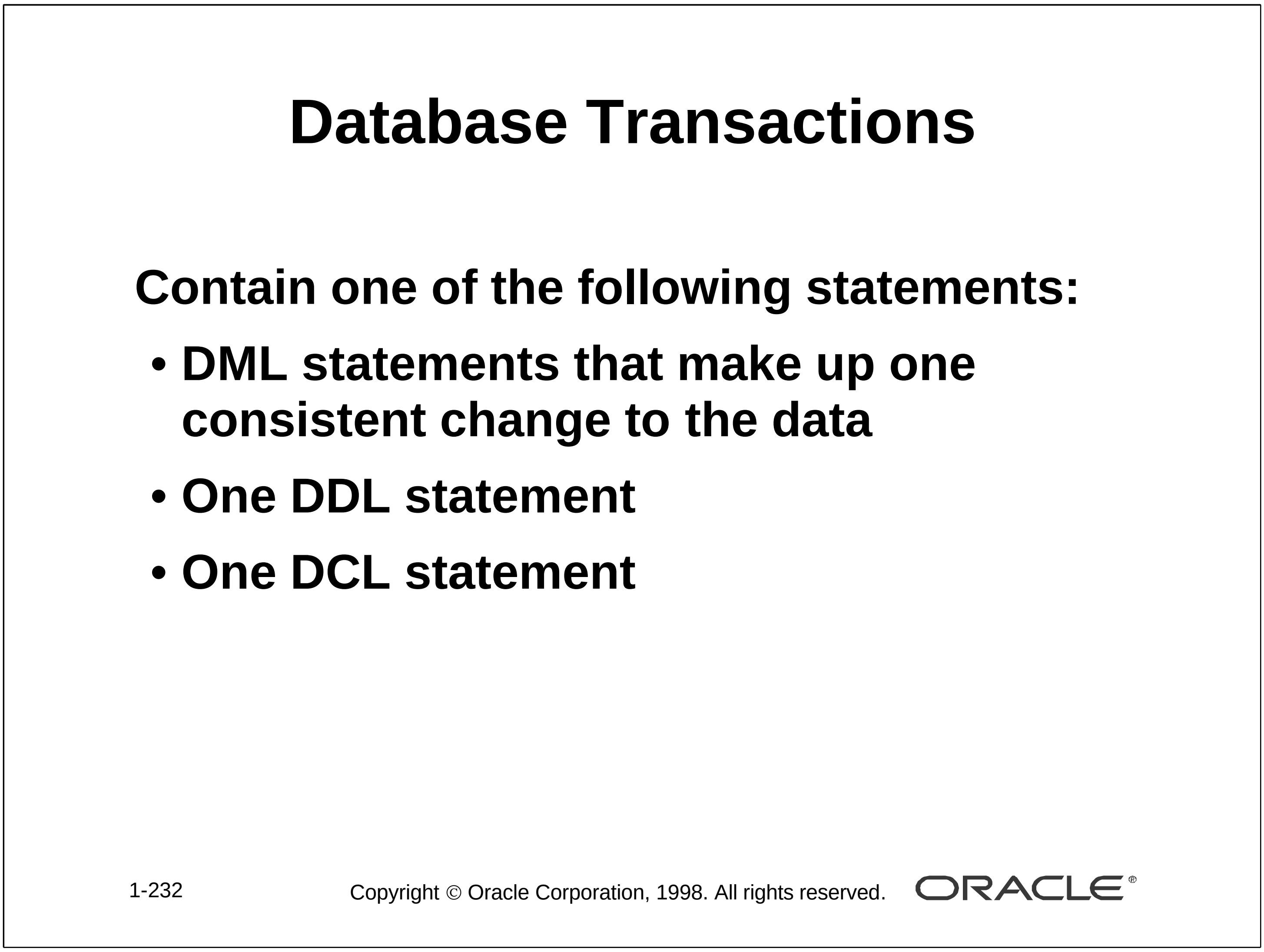
### Integrity Constraint Error

If you attempt to delete a record with a value that is tied to an integrity constraint, you will experience an error.

The example above tries to delete department number 10 from the DEPT table, but it results in an error because department number is used as a foreign key in the EMP table. If the parent record you attempt to delete has child records, then you receive the “child record found” violation ORA-02292.

### Class Management Note

If there are referential integrity constraints in use, you might receive an Oracle Server error message when you attempt to delete a row. However, if the referential integrity constraint contains the ON DELETE CASCADE option, then the selected row and its children are deleted from their respective tables.



# Database Transactions

## Contain one of the following statements:

- DML statements that make up one consistent change to the data
- One DDL statement
- One DCL statement

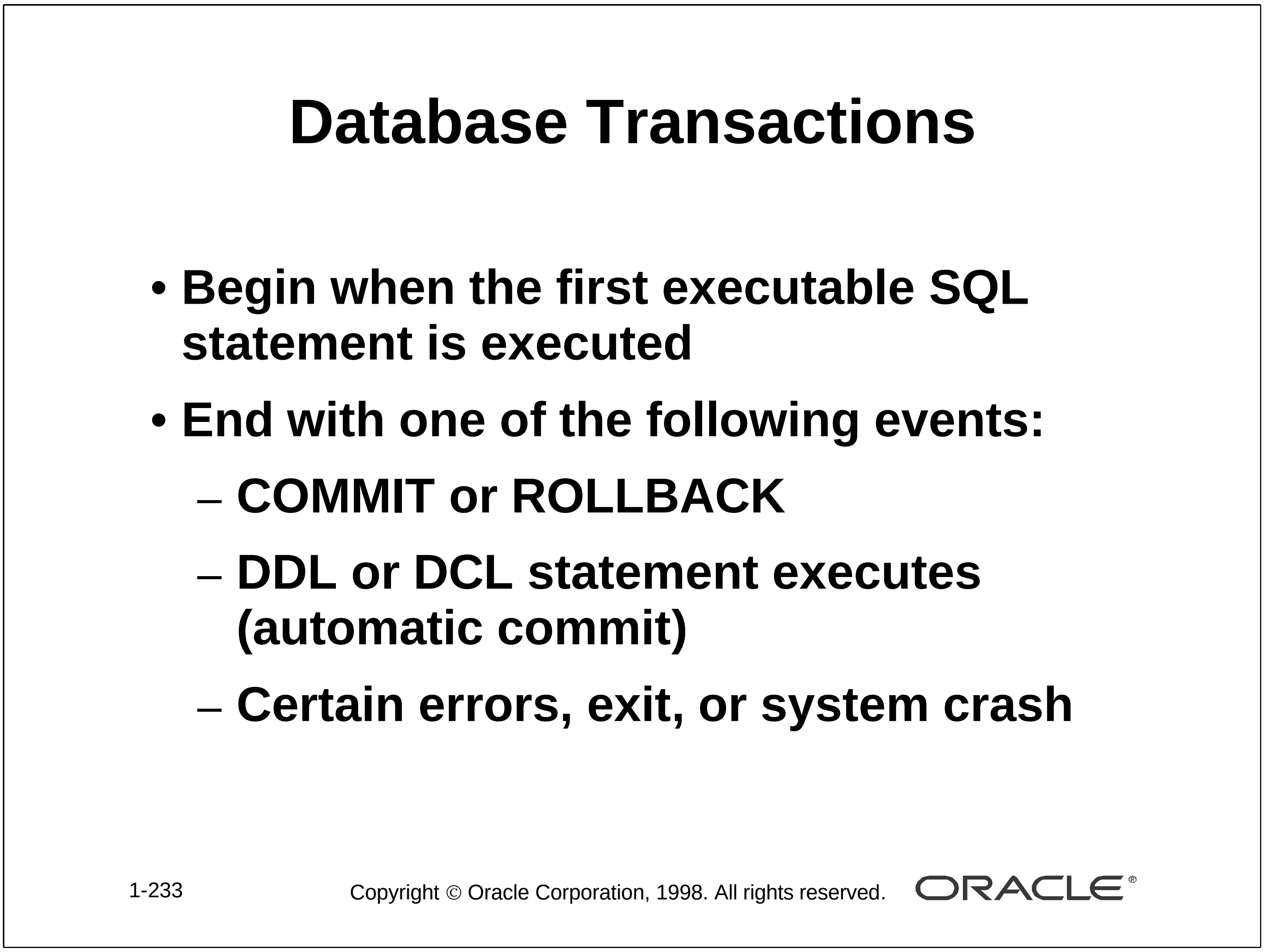
## Database Transactions

The Oracle8 Server ensures data consistency based on transactions. Transactions give you more flexibility and control when changing data, and they assure data consistency in the event of user process failure or system failure.

Transactions consist of DML statements that make up one consistent change to the data. For example, a transfer of funds between two accounts should include the debit to one account and the credit to another account in the same amount. Both actions should either fail or succeed together. The credit should not be committed without the debit.

## Transaction Types

| Type | Description |
|---|---|
| Data manipulation language (DML) | Consists of any number of DML statements that the Oracle8 Server treats as a single entity or a logical unit of work |
| Data definition language (DDL) | Consists of only one DDL statement |
| Data control language (DCL) | Consists of only one DCL statement |



# Database Transactions

- Begin when the first executable SQL statement is executed
- End with one of the following events:
  - COMMIT or ROLLBACK
  - DDL or DCL statement executes (automatic commit)
  - Certain errors, exit, or system crash

1-233

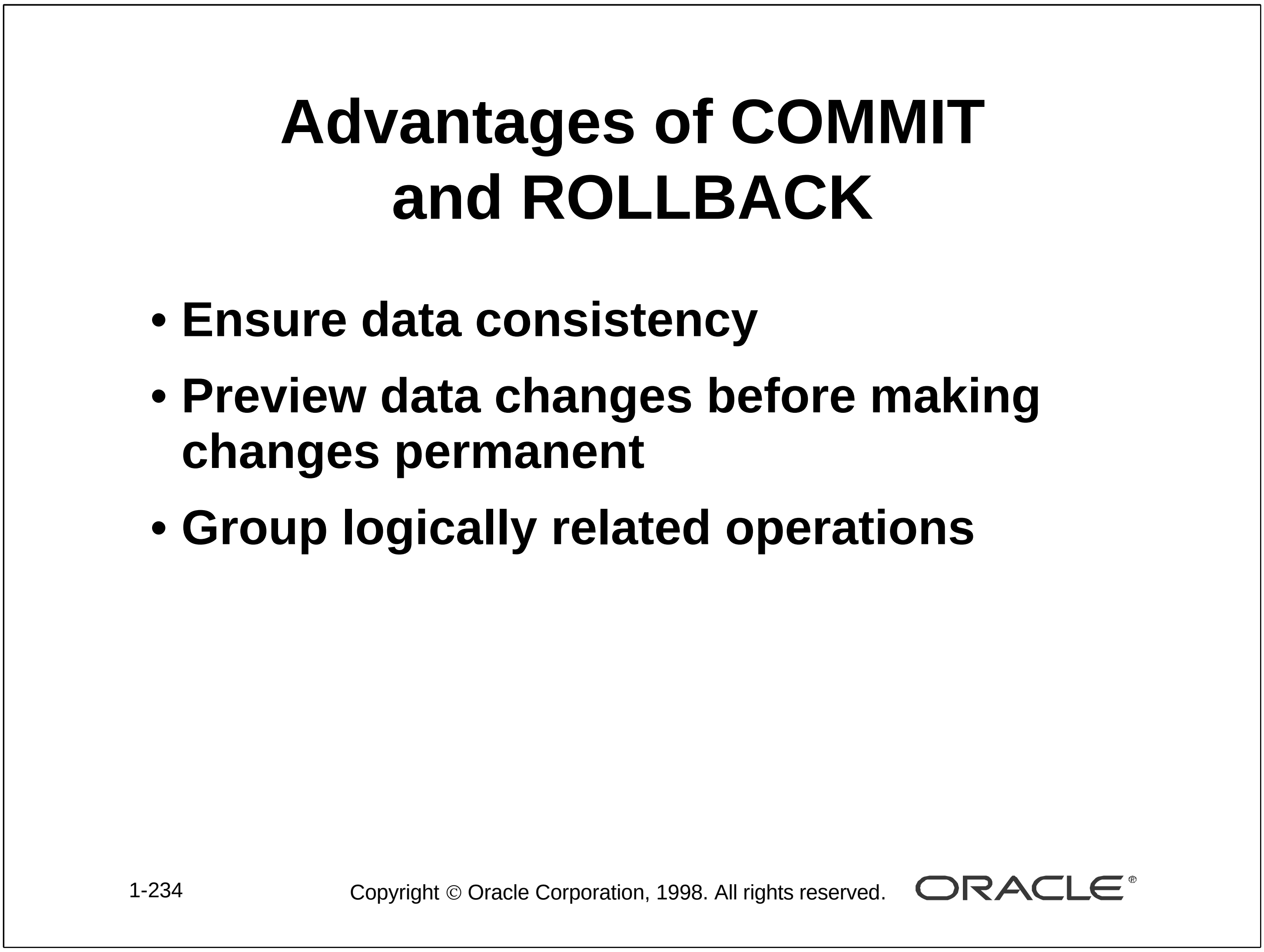
### When Does a Transaction Start and End?

A transaction begins when the first executable SQL statement is encountered and terminates when one of the following occurs:

- A COMMIT or ROLLBACK statement is issued
- A DDL statement, such as CREATE, is issued
- A DCL statement is issued
- Certain errors are detected, such as deadlocks
- The user exits SQL*Plus
- A machine fails or the system crashes

After one transaction ends, the next executable SQL statement will automatically start the next transaction.

A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction.



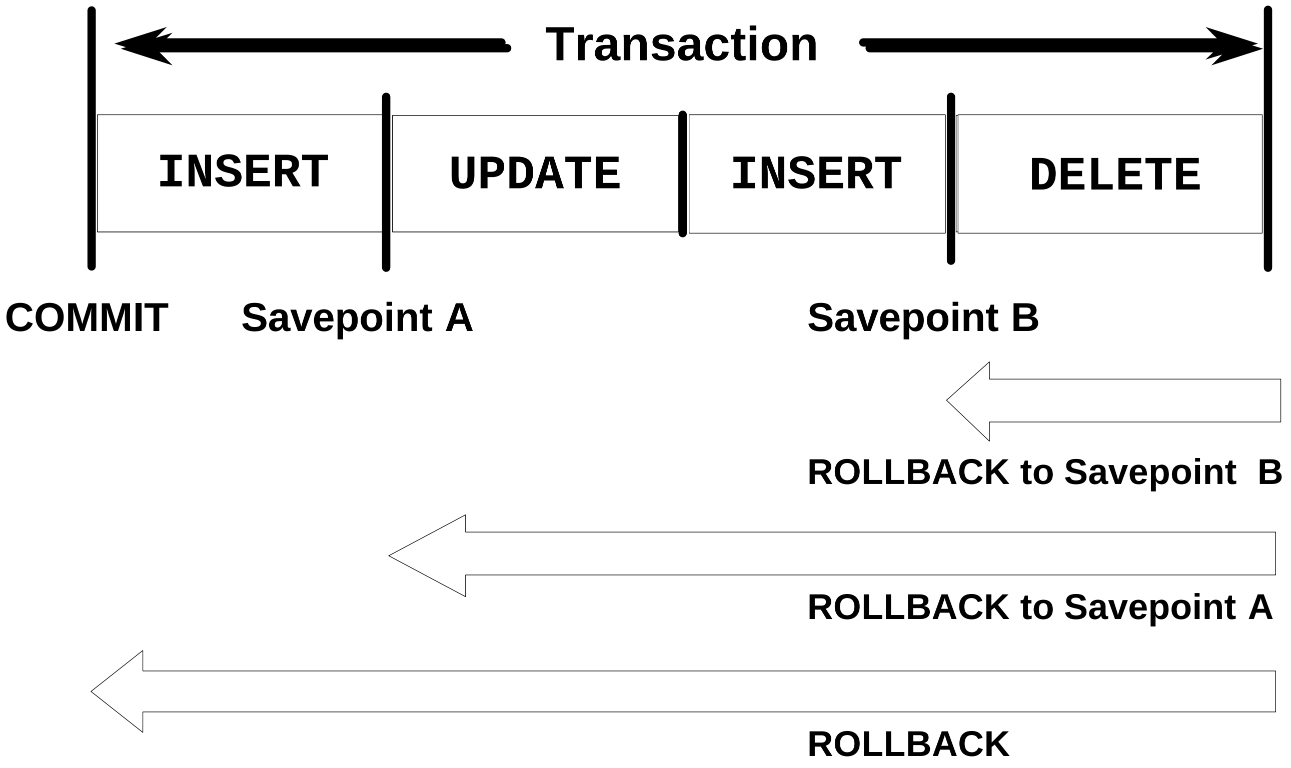
# Advantages of COMMIT and ROLLBACK

- Ensure data consistency
- Preview data changes before making changes permanent
- Group logically related operations

1-234

ORACLE®

# Controlling Transactions



Transaction

INSERT

UPDATE

INSERT

DELETE

COMMIT

Savepoint A

Savepoint B

ROLLBACK to Savepoint B

ROLLBACK to Savepoint A

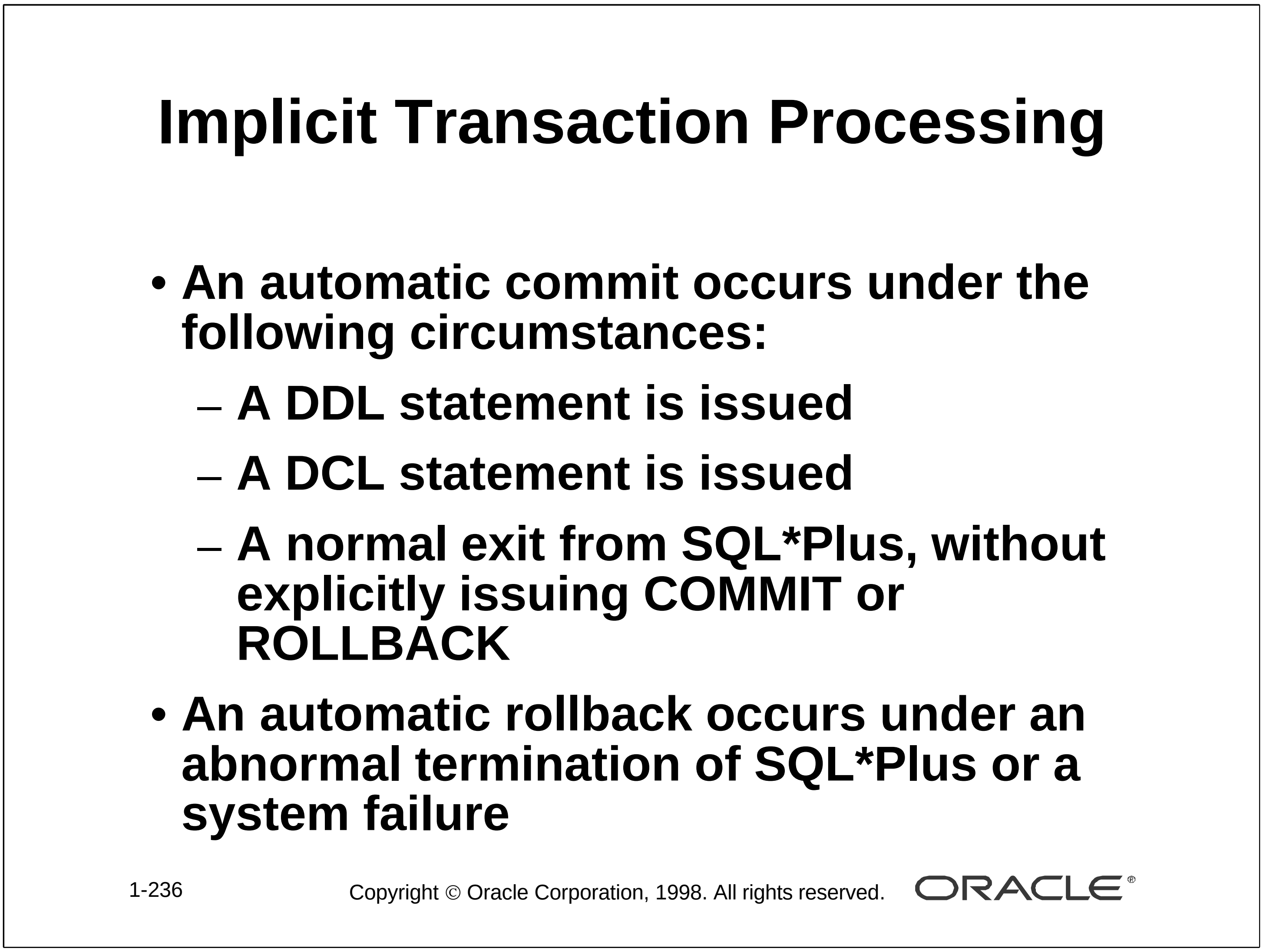
ROLLBACK

### Explicit Transaction Control Statements

You can control the logic of transactions by using the COMMIT, SAVEPOINT, and ROLLBACK statements.

| Statement | Description |
|---|---|
| COMMIT | Ends the current transaction by making all pending data changes permanent |
| SAVEPOINT *name* | Marks a savepoint within the current transaction |
| ROLLBACK [TO *SAVEPOINT name*] | A ROLLBACK ends the current transaction by discarding all pending data changes. ROLLBACK TO *SAVEPOINT name* discards the savepoint and all subsequent changes |

**Note:** SAVEPOINT is not ANSI standard SQL.



# Implicit Transaction Processing

- An automatic commit occurs under the following circumstances:
  - A DDL statement is issued
  - A DCL statement is issued
  - A normal exit from SQL*Plus, without explicitly issuing COMMIT or ROLLBACK
- An automatic rollback occurs under an abnormal termination of SQL*Plus or a system failure

1-236

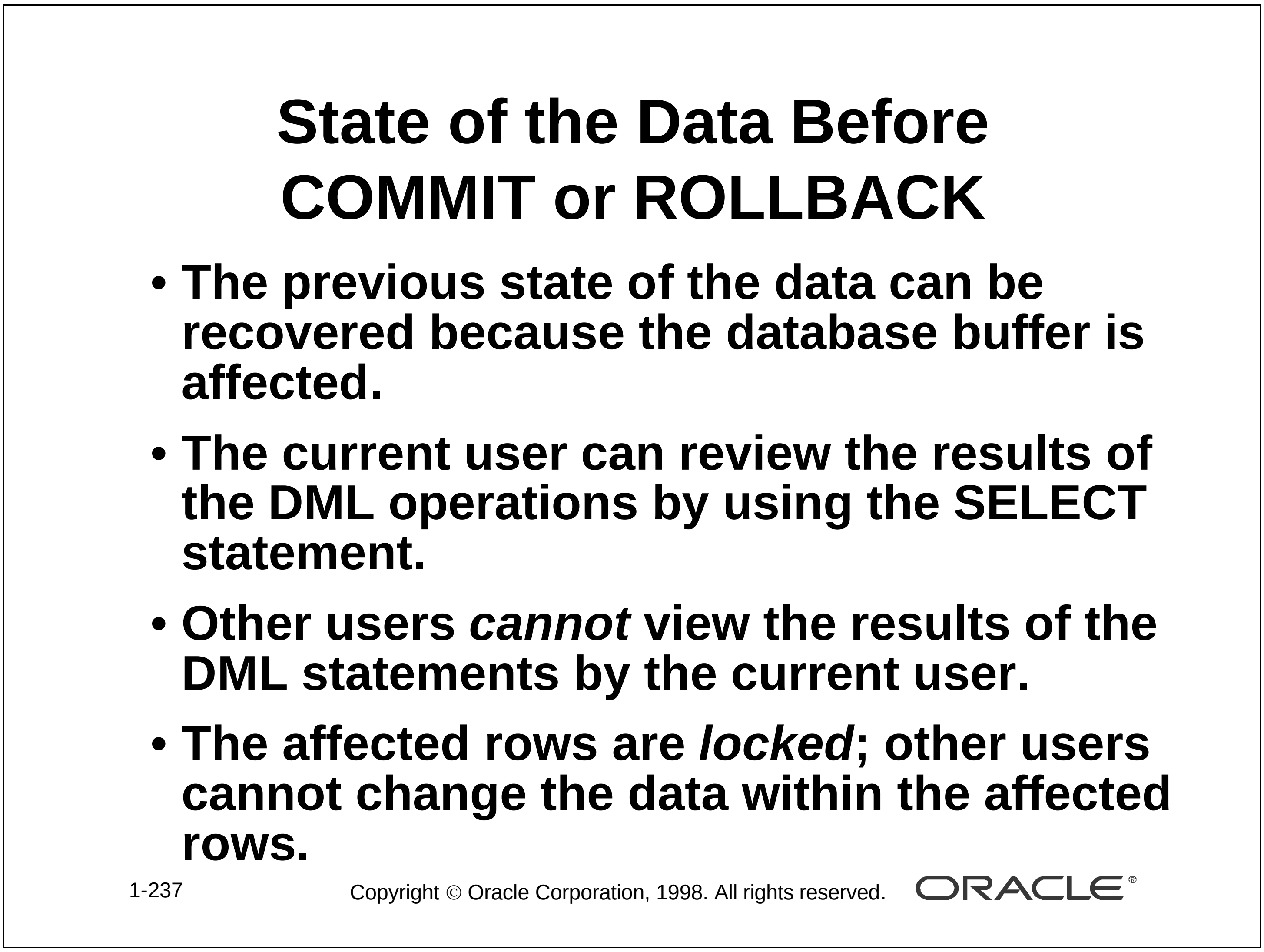
## Implicit Transaction Processing

| Status | Circumstances |
|---|---|
| Automatic commit | DDL statement or DCL statement is issued<br>Normal exit from SQL*Plus, without explicitly issuing COMMIT or ROLLBACK |
| Automatic rollback | Abnormal termination of SQL*Plus or system failure |

**Note:** A third command is available in SQL*Plus. The SQL*Plus AUTOCOMMIT command can be toggled to be ON or OFF. If set to ON, each individual DML statement is committed as soon as it is executed. You cannot roll back the changes. If set to OFF, COMMIT can be issued explicitly. Also, COMMIT is issued when a DDL statement is issued or when you exit from SQL*Plus.

## System Failures

When a transaction is interrupted by a system failure, the entire transaction is automatically rolled back. This prevents the error from causing unwanted changes to the data and returns the tables to their state at the time of the last commit. In this way, SQL*Plus protects the integrity of the tables.



# State of the Data Before COMMIT or ROLLBACK

- The previous state of the data can be recovered because the database buffer is affected.
- The current user can review the results of the DML operations by using the SELECT statement.
- Other users *cannot* view the results of the DML statements by the current user.
- The affected rows are *locked*; other users cannot change the data within the affected rows.

1-237

### Committing Changes

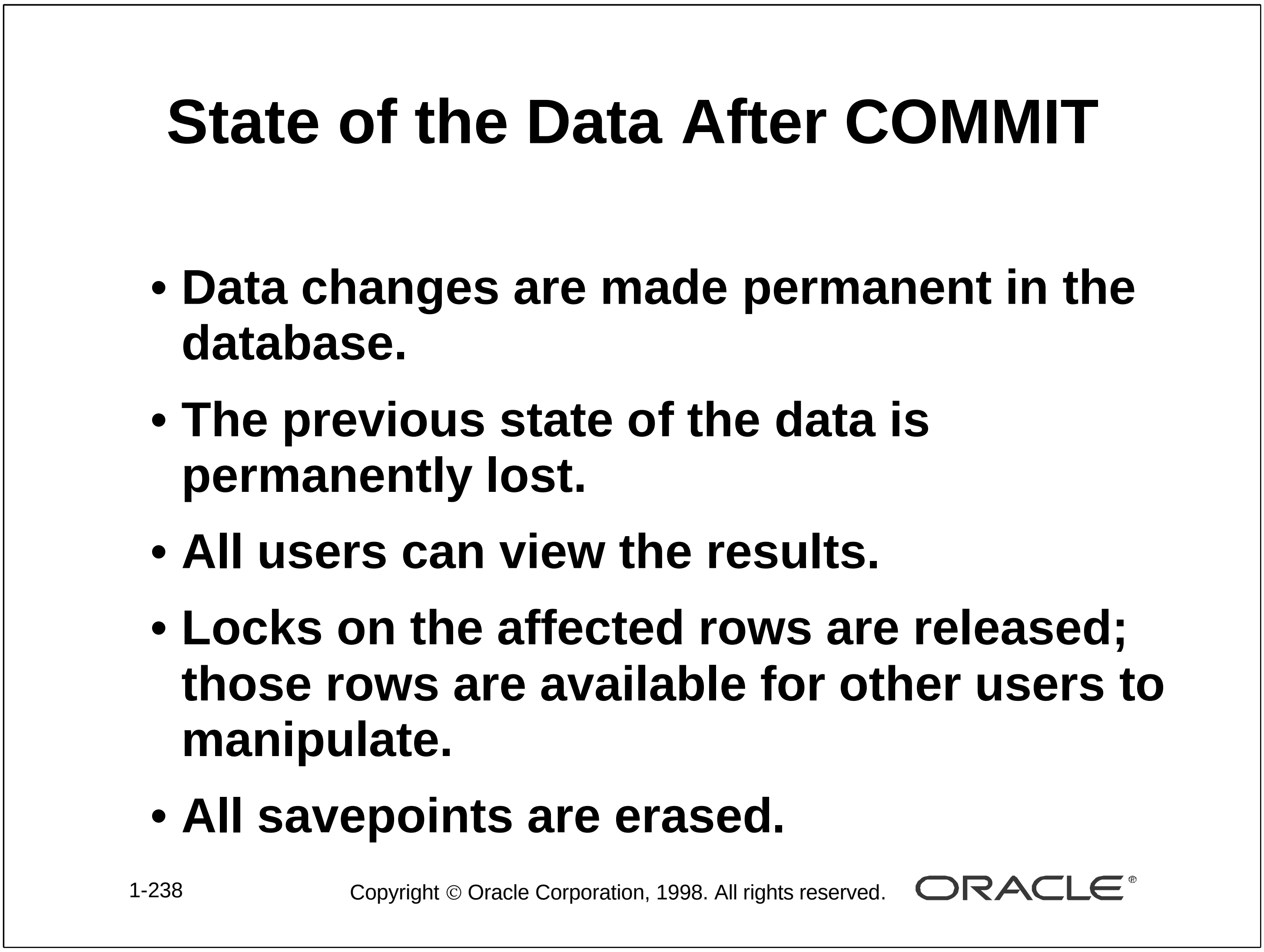
Every data change made during the transaction is temporary until the transaction is committed.

### State of the Data Before COMMIT or ROLLBACK

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current user can review the results of the data manipulation operations by querying the tables.
- Other users cannot view the results of the data manipulation operations made by the current user. Oracle8 institutes read consistency to ensure that each user sees data as it existed at the last commit.
- The affected rows are locked; other users cannot change the data within the affected rows

### Class Management Note

With the Oracle8 Server, data changes may actually be written to the database files before COMMIT, but they are still only temporary.
If a number of users are making changes simultaneously to the same table, then each user sees only his or her changes until other users commit their changes.
Other users see data as it is committed in the database (in other words, before changes).
By default, the Oracle8 Server has *row-level locking*. It is possible to alter the default locking mechanism.



# State of the Data After COMMIT

- Data changes are made permanent in the database.
- The previous state of the data is permanently lost.
- All users can view the results.
- Locks on the affected rows are released; those rows are available for other users to manipulate.
- All savepoints are erased.

1-238

### Committing Changes

Make all pending changes permanent by using the COMMIT statement. Following a COMMIT:

- Data changes are written to the database.
- The previous state of the data is permanently lost.
- All users can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other users to perform new data changes.
- All savepoints are erased.

# Committing Data

- **Make the changes.**

```
SQL> UPDATE   emp
  2  SET      deptno = 10
  3  WHERE    empno = 7782;
1 row updated.
```

- **Commit the changes.**

```
SQL> COMMIT;
Commit complete.
```

### Committing Changes

The above example updates the EMP table and sets the department number for employee 7782 (Clark) to 10. It then makes the change permanent by issuing the COMMIT statement.

### Example

Create a new ADVERTISING department with at least one employee. Make the data change permanent.

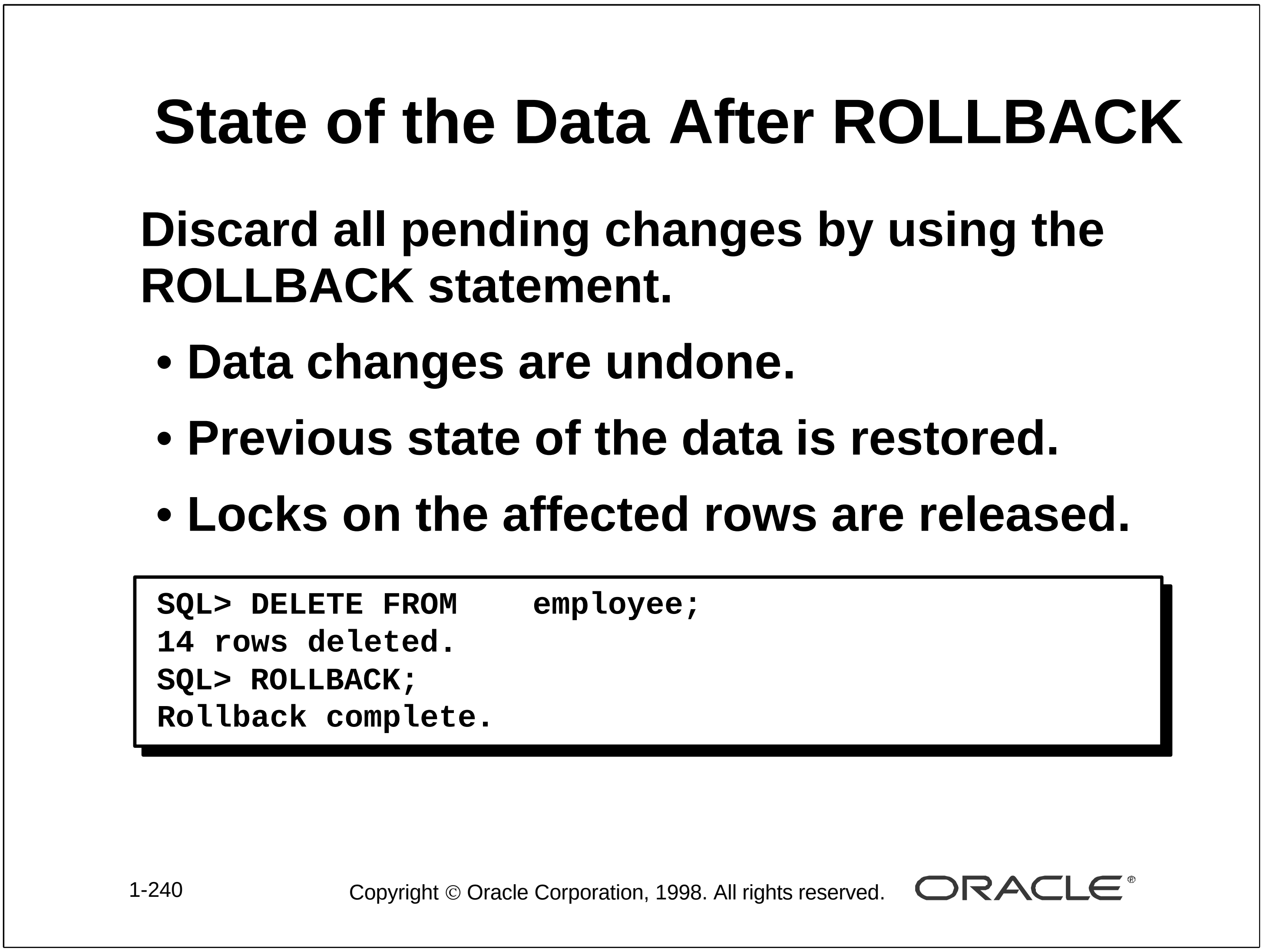
```
SQL> INSERT INTO department(deptno, dname, loc)
  2  VALUES      (50, 'ADVERTISING', 'MIAMI');
1 row created.
```

```
SQL> UPDATE  employee
  2  SET     deptno = 50
  3  WHERE   empno = 7876;
1 row updated.
```

```
SQL> COMMIT;
Commit complete.
```

### Class Management Note

Use this example to explain how COMMIT ensures that two related operations must occur together, or not at all. In this case, COMMIT prevents empty departments from being created.



# State of the Data After ROLLBACK

**Discard all pending changes by using the ROLLBACK statement.**

- **Data changes are undone.**
- **Previous state of the data is restored.**
- **Locks on the affected rows are released.**

```
SQL> DELETE FROM    employee;
14 rows deleted.
SQL> ROLLBACK;
Rollback complete.
```

1-240

### Rolling Back Changes

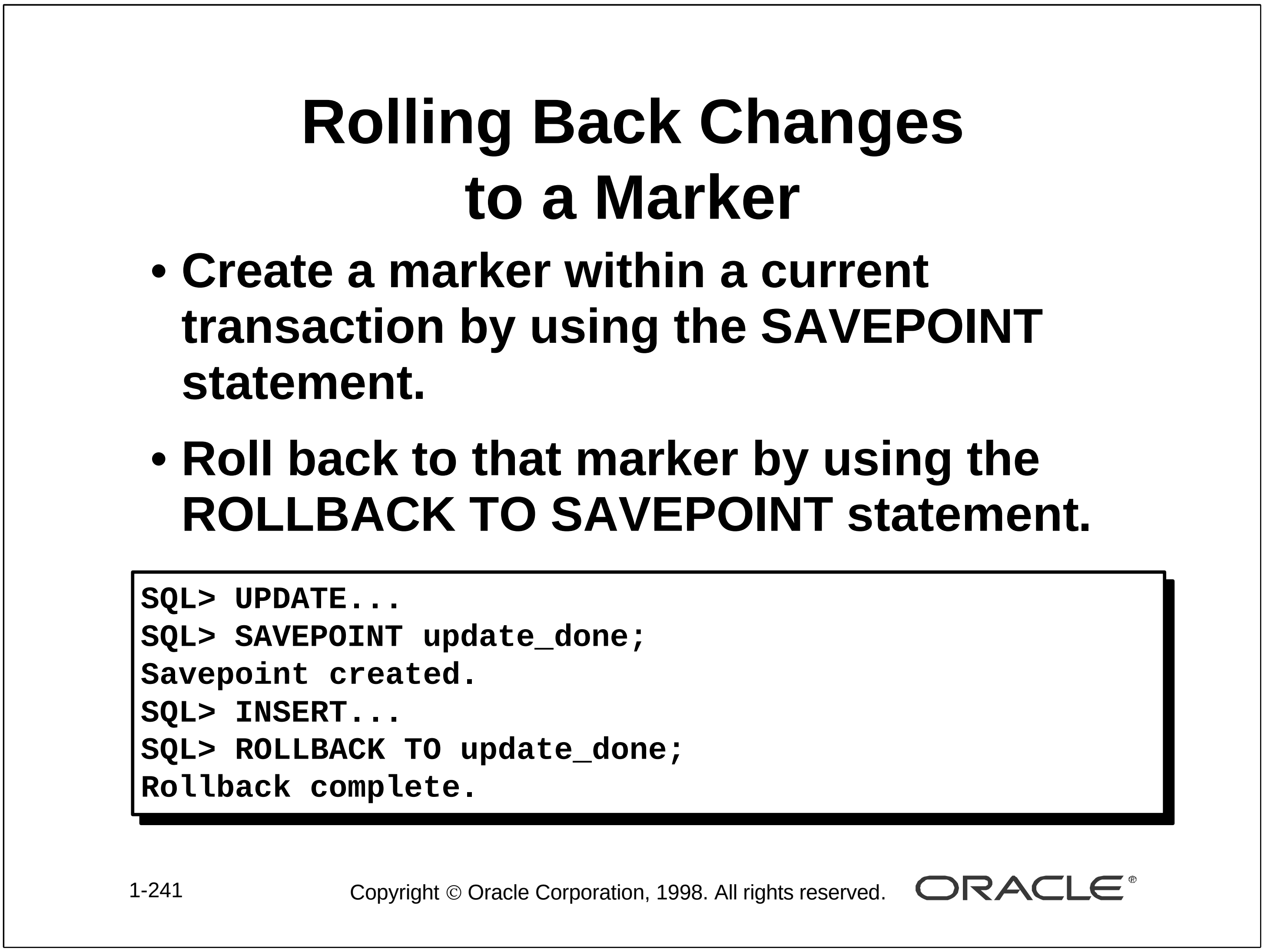
Discard all pending changes by using the ROLLBACK statement. Following a ROLLBACK

- Data changes are undone.
- The previous state of the data is restored.
- The locks on the affected rows are released.

### Example

While attempting to remove a record from the TEST table, you can accidentally empty the table. You can correct the mistake, then reissue the proper statement, and make the data change permanent.

```
SQL> DELETE FROM  test;
25,000 rows deleted.
SQL> ROLLBACK;
Rollback complete.
SQL> DELETE FROM  test
  2  WHERE        id = 100;
1 row deleted.
SQL> SELECT   *
  2  FROM     test
  3  WHERE    id = 100;
No rows selected.
SQL> COMMIT;
Commit complete.
```



# Rolling Back Changes to a Marker

- Create a marker within a current transaction by using the SAVEPOINT statement.
- Roll back to that marker by using the ROLLBACK TO SAVEPOINT statement.

```
SQL> UPDATE...
SQL> SAVEPOINT update_done;
Savepoint created.
SQL> INSERT...
SQL> ROLLBACK TO update_done;
Rollback complete.
```

1-241

### Rolling Back Changes to a Savepoint

You can create a marker within the current transaction by using the SAVEPOINT statement. The transaction therefore can be divided into smaller sections. You can then discard pending changes up to that marker by using the ROLLBACK TO SAVEPOINT statement.

If you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.

### Class Management Note

Savepoints are especially useful in PL/SQL or a 3GL program in which recent changes can be undone conditionally based on runtime conditions.

# Statement-Level Rollback

- If a single DML statement fails during execution, only that statement is rolled back.
- Oracle8 implements an implicit savepoint.
- All other changes are retained.
- The user should terminate transactions explicitly by executing a COMMIT or ROLLBACK statement.

### Statement-Level Rollback

Part of a transaction can be discarded by an implicit rollback if a statement execution error is detected. If a single DML statement fails during execution of a transaction, its effect is undone by a statement-level rollback, but the changes made by the previous DML statements in the transaction will not be discarded. They can be committed or rolled back explicitly by the user.

Oracle issues an implicit COMMIT before and after any data definition language (DDL) statement. So, even if your DDL statement does not execute successfully, you cannot roll back the previous statement because the server issued a commit.

Terminate your transactions explicitly by executing a COMMIT or ROLLBACK statement.

### Class Management Note

The Oracle8 Server implements locks on data to provide data concurrency in the database. Those locks are released when certain events occur (such as a system failure) or the transaction is completed.
Implicit locks on the database are obtained when a DML statement is successfully executed. The Oracle8 Server by default locks data at the lowest level possible.
Manually acquire locks on the database tables by executing a LOCK TABLE statement or the SELECT statement with the FOR UPDATE clause.

# Read Consistency

- Read consistency guarantees a consistent view of the data at all times.
- Changes made by one user do not conflict with changes made by another user.
- Ensures that on the same data:
  - Readers do not wait for writers
  - Writers do not wait for readers

**Read Consistency**

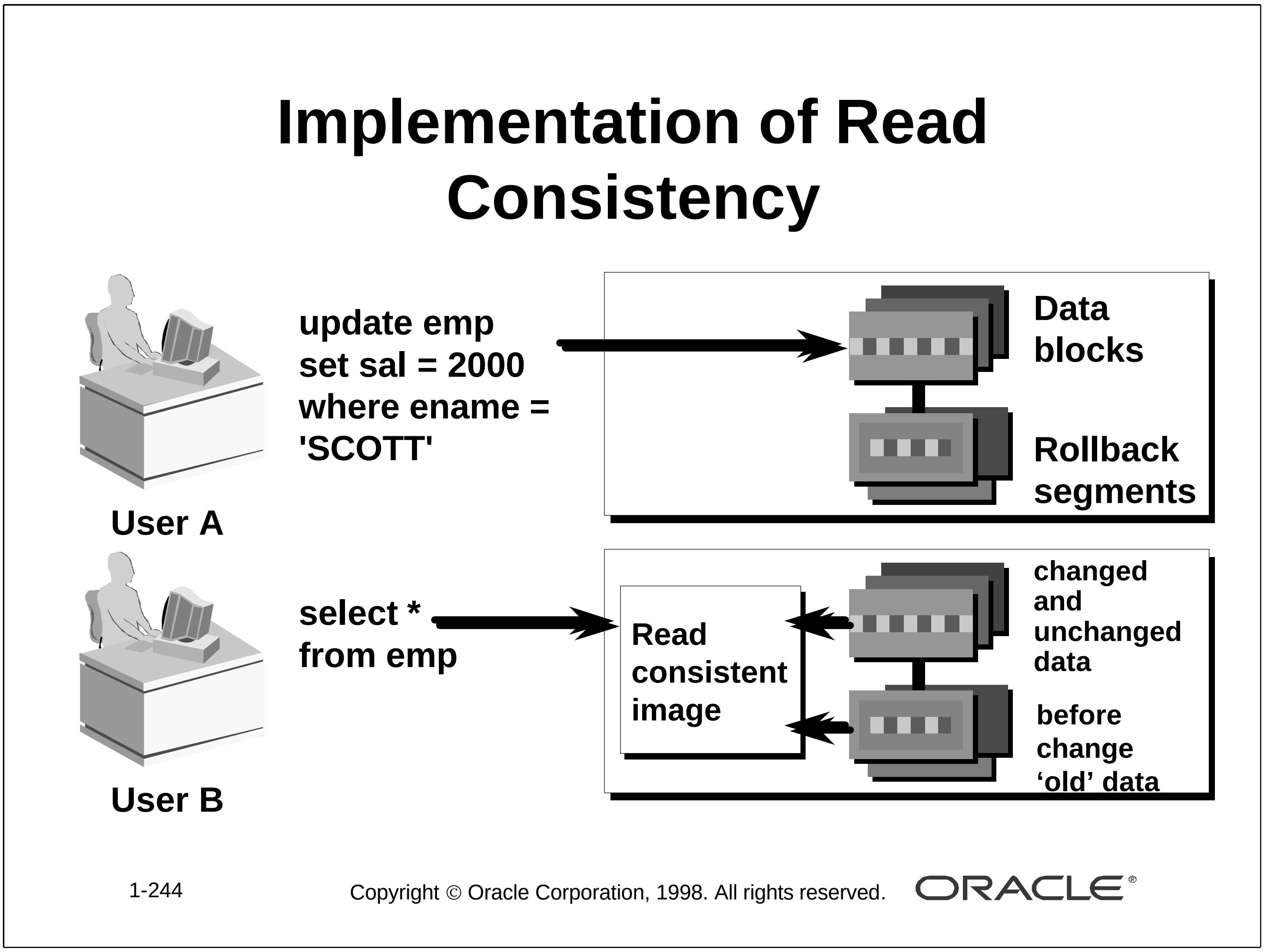
Database users make two types of access to the database

- Read operations (SELECT statement)
- Write operations (INSERT, UPDATE, DELETE statements)

You need read consistency so that the following occur:

- The database reader and writer are ensured a consistent view of the data
- Readers do not view data that is in the process of being changed
- Writers are ensured that the changes to the database are done in a consistent way
- Changes made by one writer do not disrupt or conflict with changes another writer is making

The purpose of read consistency is to ensure that each user sees data as it existed at the last commit, before a DML operation started.



# Implementation of Read Consistency

User A

update emp
set sal = 2000
where ename =
'SCOTT'

Data blocks

Rollback segments

User B

select *
from emp

Read consistent image

changed and unchanged data

before change ‘old’ data

1-244

### Implementation of Read Consistency

Read consistency is an automatic implementation. It keeps a partial copy of the database in rollback segments.

When an insert, update, or delete operation is made to the database,  Oracle8 Server takes a copy of the data before it is changed and writes it to a *rollback segment*.

All readers, except the one who issued the change, still see the database as it existed before the changes started; they view the rollback segments’ “snapshot” of the data.

Before changes are committed to the database, only the user who is modifying the data sees the database with the alterations, everyone else sees the snapshot in the rollback segment. This guarantees that readers of the data read consistent data that is not currently undergoing change.

When a DML statement is committed, the change made to the database becomes visible to anyone executing a SELECT statement.

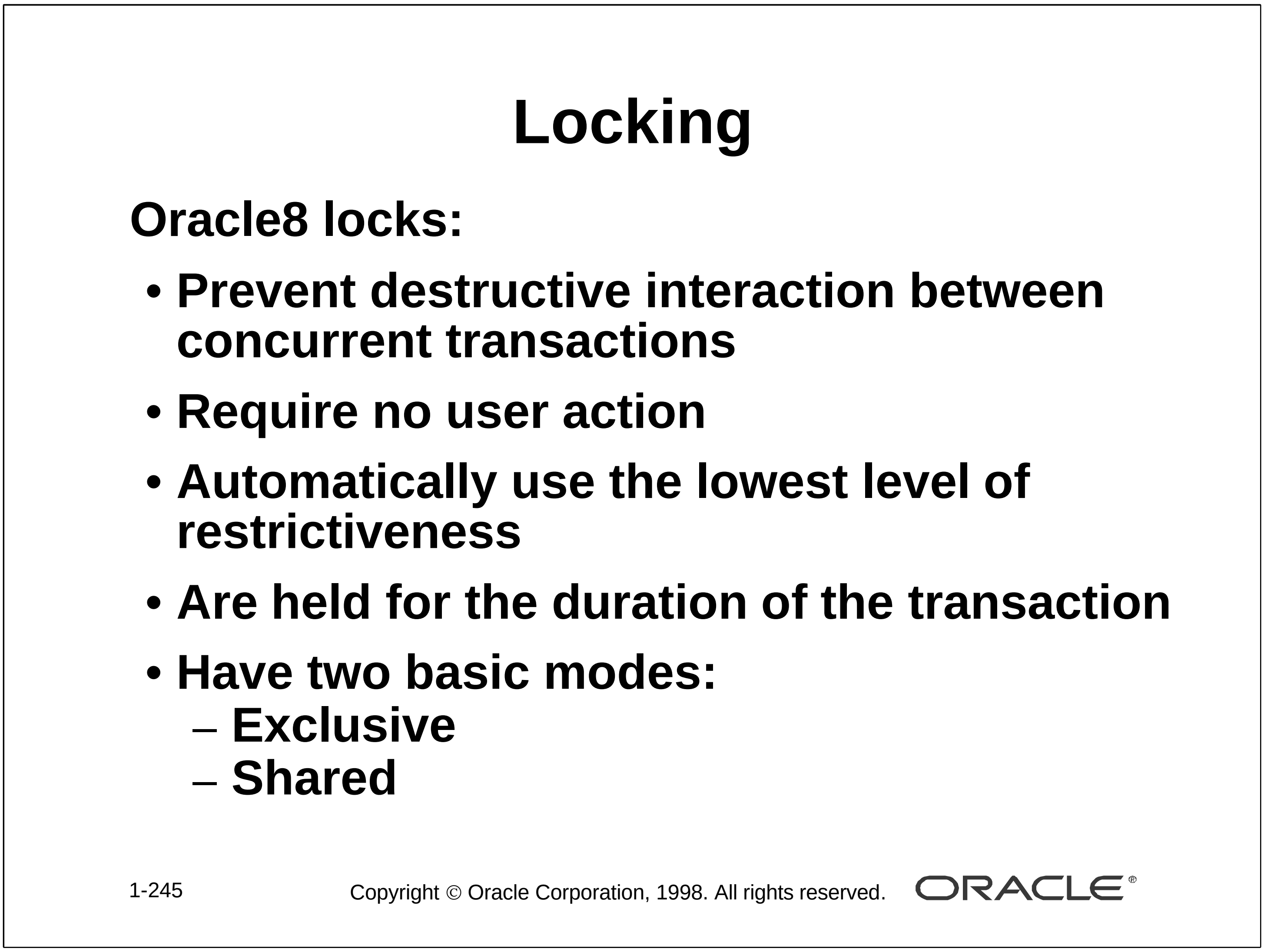
The space occupied by the “old” data in the rollback segment file is freed for reuse.

If the transaction is rolled back, the changes are “undone.”

- The original, older version, of the data in the rollback segment is written back to the table.
- All users see the database as it existed before the transaction began.

### Class Management Note

Do not get into the details of data blocks and rollback segments. Explain them in basic terms to describe read consistency.



# Locking

**Oracle8 locks:**

- **Prevent destructive interaction between concurrent transactions**
- **Require no user action**
- **Automatically use the lowest level of restrictiveness**
- **Are held for the duration of the transaction**
- **Have two basic modes:**
  - **Exclusive**
  - **Shared**

1-245

### What are Locks?

Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource, either a user object (such as tables or rows) or system objects not visible to users (such as shared data structures and data dictionary rows).

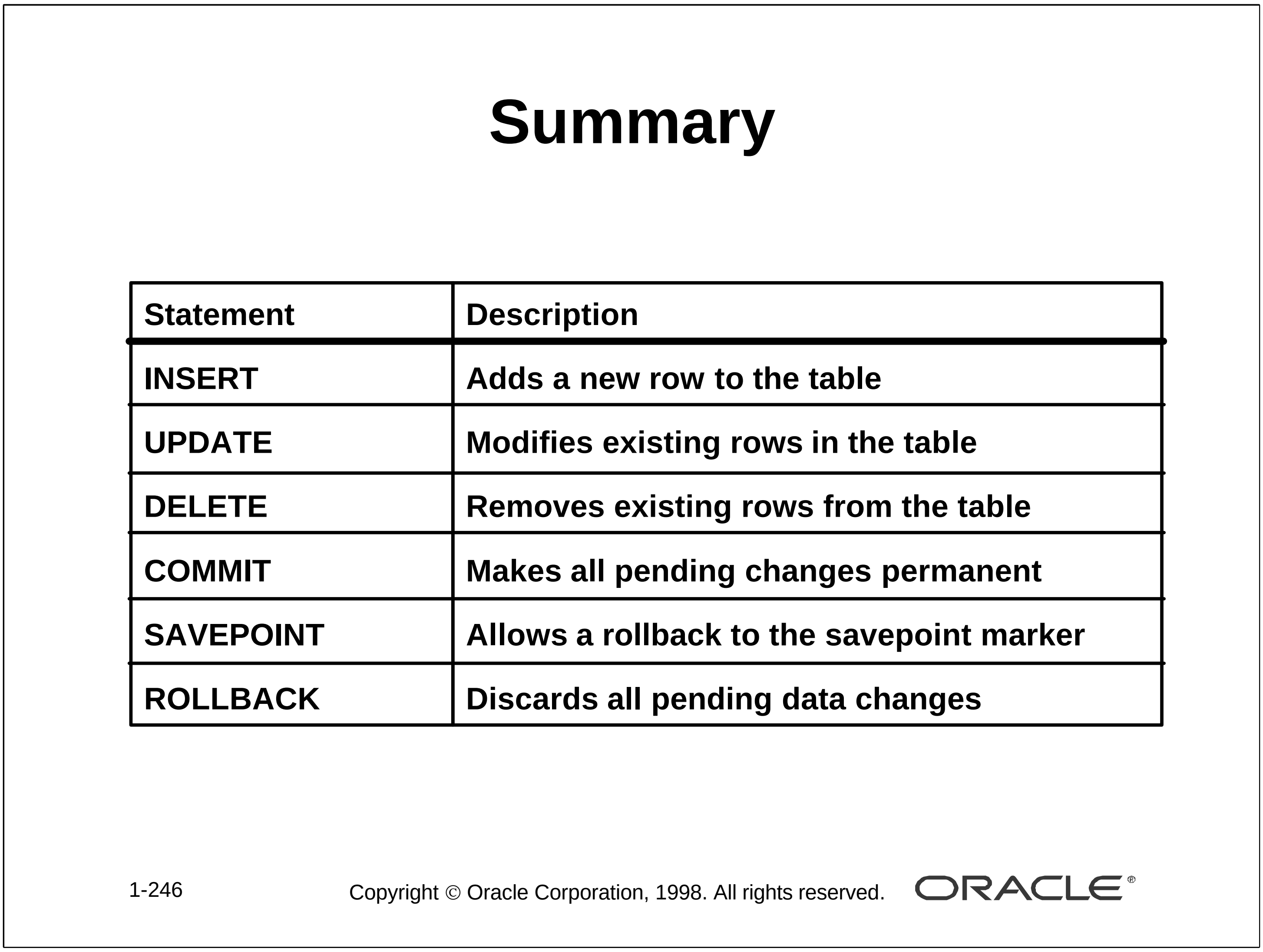
### How Oracle Locks Data

Locking in an Oracle database is fully automatic and requires no user action. Implicit locking occurs for all SQL statements. The Oracle default locking mechanism automatically uses the lowest applicable level of restrictiveness, thus providing the highest degree of concurrency yet also providing maximum data integrity. Oracle also allows the user to lock data manually.

### Locking Modes

Oracle uses two modes of locking in a multiuser database.

| Lock Mode | Description |
|---|---|
| *exclusive* | Prevents a resource from being shared<br>The first transaction to lock a resource exclusively, is the only transaction that can alter the resource until the exclusive lock is released |
| *share lock* | Allows the resource to be shared<br>Multiple users reading data can share the data, holding share locks to prevent concurrent access by a writer (who needs an exclusive lock)<br>Several transactions can acquire share locks on the same resource |



# Summary

| Statement | Description |
|---|---|
| INSERT | Adds a new row to the table |
| UPDATE | Modifies existing rows in the table |
| DELETE | Removes existing rows from the table |
| COMMIT | Makes all pending changes permanent |
| SAVEPOINT | Allows a rollback to the savepoint marker |
| ROLLBACK | Discards all pending data changes |

### Summary

Manipulate data in the Oracle database by using the INSERT, UPDATE, and DELETE statements. Control data changes by using the COMMIT, SAVEPOINT, and ROLLBACK statements.

Oracle8 guarantees a consistent view of data at all times.

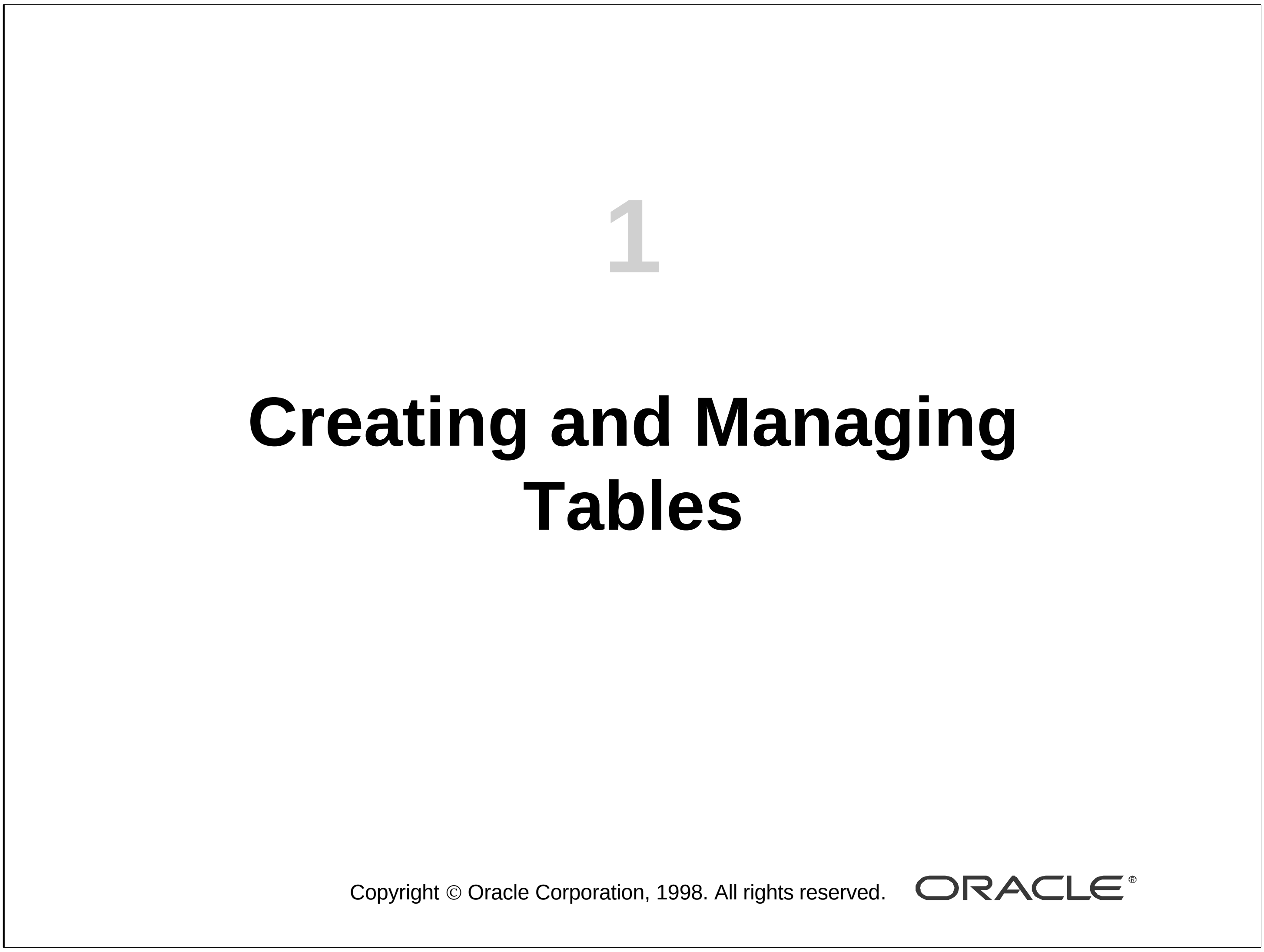
Locking can be implicit or explicit.

# Practice Overview

- Inserting rows into the tables.
- Updating and deleting rows in the table.
- Controlling transactions.

### Practice Overview

In this practice, you will add rows to the MY_EMPLOYEE table, update, and delete data from the table, and control your transactions.



1

# Creating and Managing Tables

| Schedule: | Timing | Topic |
|---|---|---|
| | 30 minutes | Lecture |
| | 20 minutes | Practice |
| | 50 minutes | Total |

# Objectives

**At the end of this lesson, you will be able to:**

- **Describe the main database objects**
- **Create tables**
- **Describe the datatypes that can be used when specifying column definition**
- **Alter table definitions**
- **Drop, rename, and truncate tables**

1-252

**Lesson Aim**

In this lesson, you will learn about main database objects and their relationships to each other. You will also learn how to create, alter, and drop tables.

# Database Objects

| Object | Description |
|---|---|
| Table | Basic unit of storage; composed of rows and columns |
| View | Logically represents subsets of data from one or more tables |
| Sequence | Generates primary key values |
| Index | Improves the performance of some queries |
| Synonym | Gives alternative names to objects |

### Database Objects

An Oracle8 database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

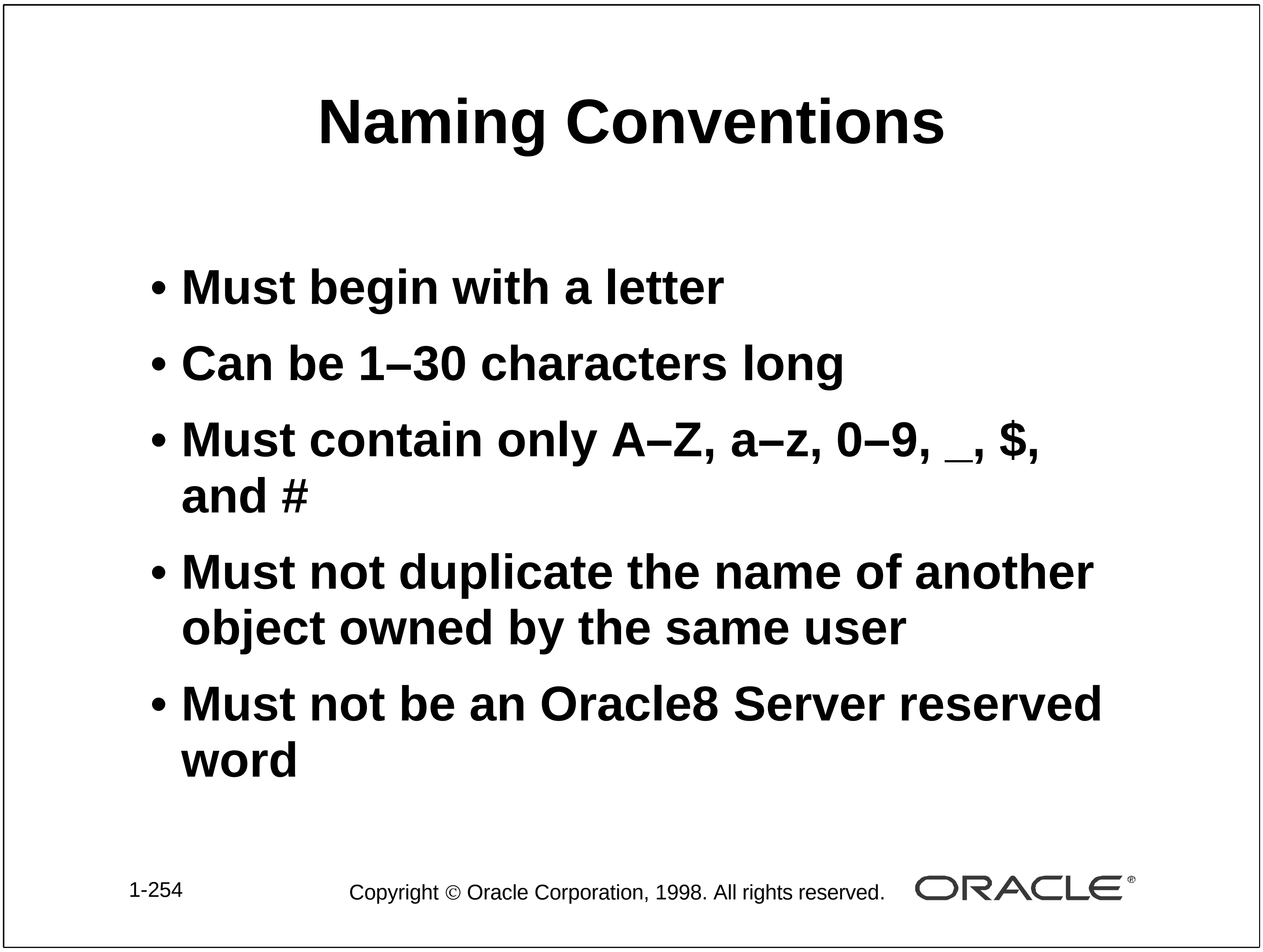
- Table: Stores data
- View: Subset of data from one or more tables
- Sequence: Generates primary key values
- Index: Improves the performance of some queries
- Synonym: Gives alternative names to objects

### Oracle8 Table Structures

- Tables can be created at any time, even while users are using the database.
- You do not need to specify the size of any table. The size is ultimately defined by the amount of space allocated to the database as a whole. It is important, however, to estimate how much space a table will use over time.
- Table structure can be modified online.

### Class Management Note

Tables can have up to 1000 columns and must conform to standard database object naming conventions.
Column definitions can be omitted when using the AS subquery clause. Tables are created without data unless a query is specified. Rows are usually added by using INSERT statements, SQL*Loader, or  a form.



# Naming Conventions

- Must begin with a letter
- Can be 1–30 characters long
- Must contain only A–Z, a–z, 0–9, _, $, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an Oracle8 Server reserved word

1-254

### Naming Rules

Name database tables and columns according to the standard rules for naming any Oracle8 database object.

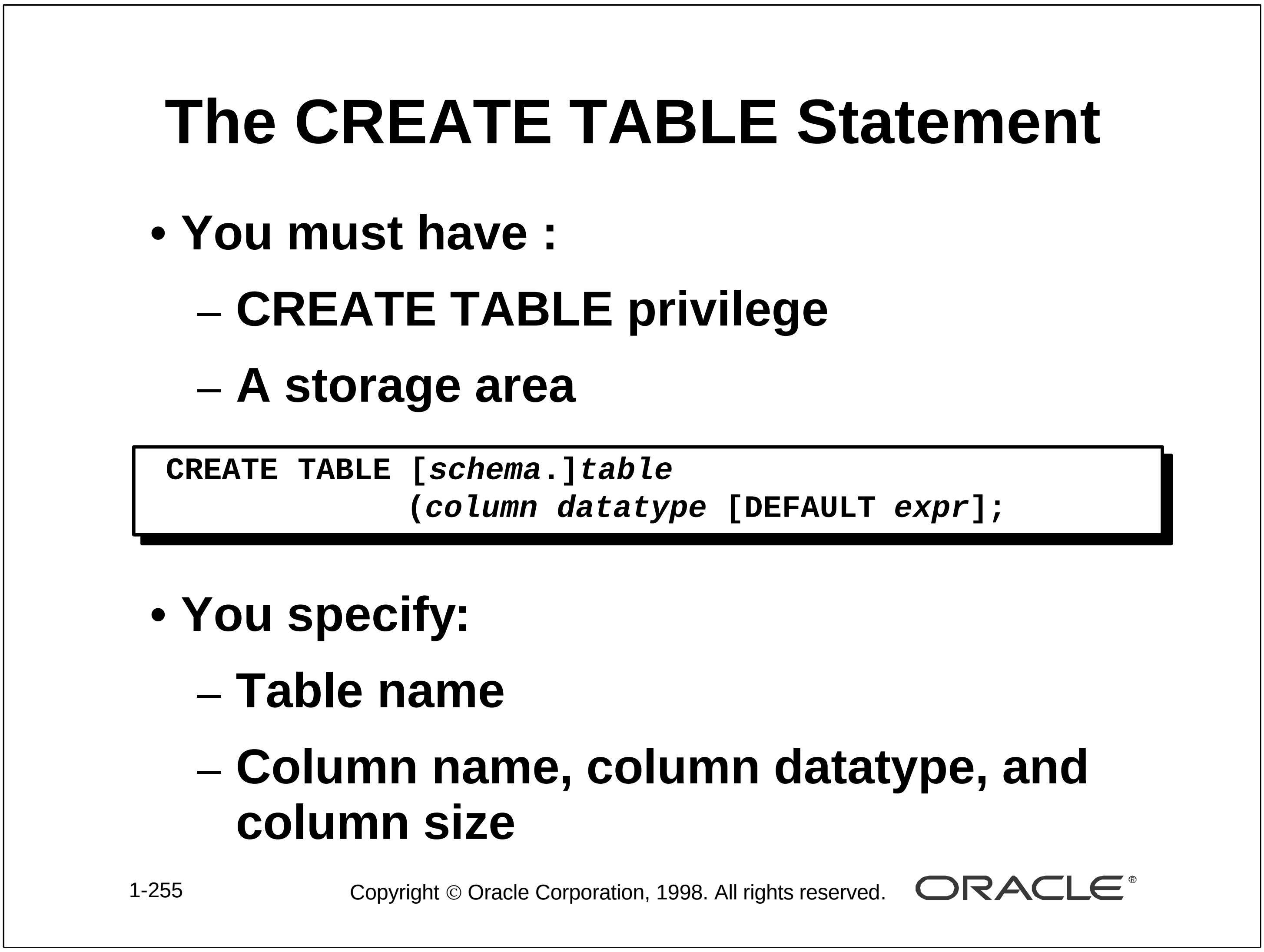
- Table names and column names must begin with a letter and can be 1-30 characters long.
- Names must contain only the characters A-Z, a-z, 0-9, _ (underscore), $, and # (legal characters, but their use is discouraged).
- Names must not duplicate the name of another object owned by the same Oracle8 Server user.
- Names must not be an Oracle8 reserved word.

### Naming Guidelines

- Use descriptive names for tables and other database objects.
- Name the same entity consistently in different tables. For example, the department number column is called DEPTNO in both the EMP table and the DEPT table.

**Note:** Names are case insensitive. For example, EMP is treated as the same name as eMP or eMp.

For more information, see
*Oracle8 Server SQL Reference, Release 8.0,* "Object Names and Qualifiers."



# The CREATE TABLE Statement

- You must have :
  - CREATE TABLE privilege
  - A storage area

```
CREATE TABLE [schema.]table
             (column datatype [DEFAULT expr];
```

- You specify:
  - Table name
  - Column name, column datatype, and column size

1-255

### The CREATE TABLE Statement

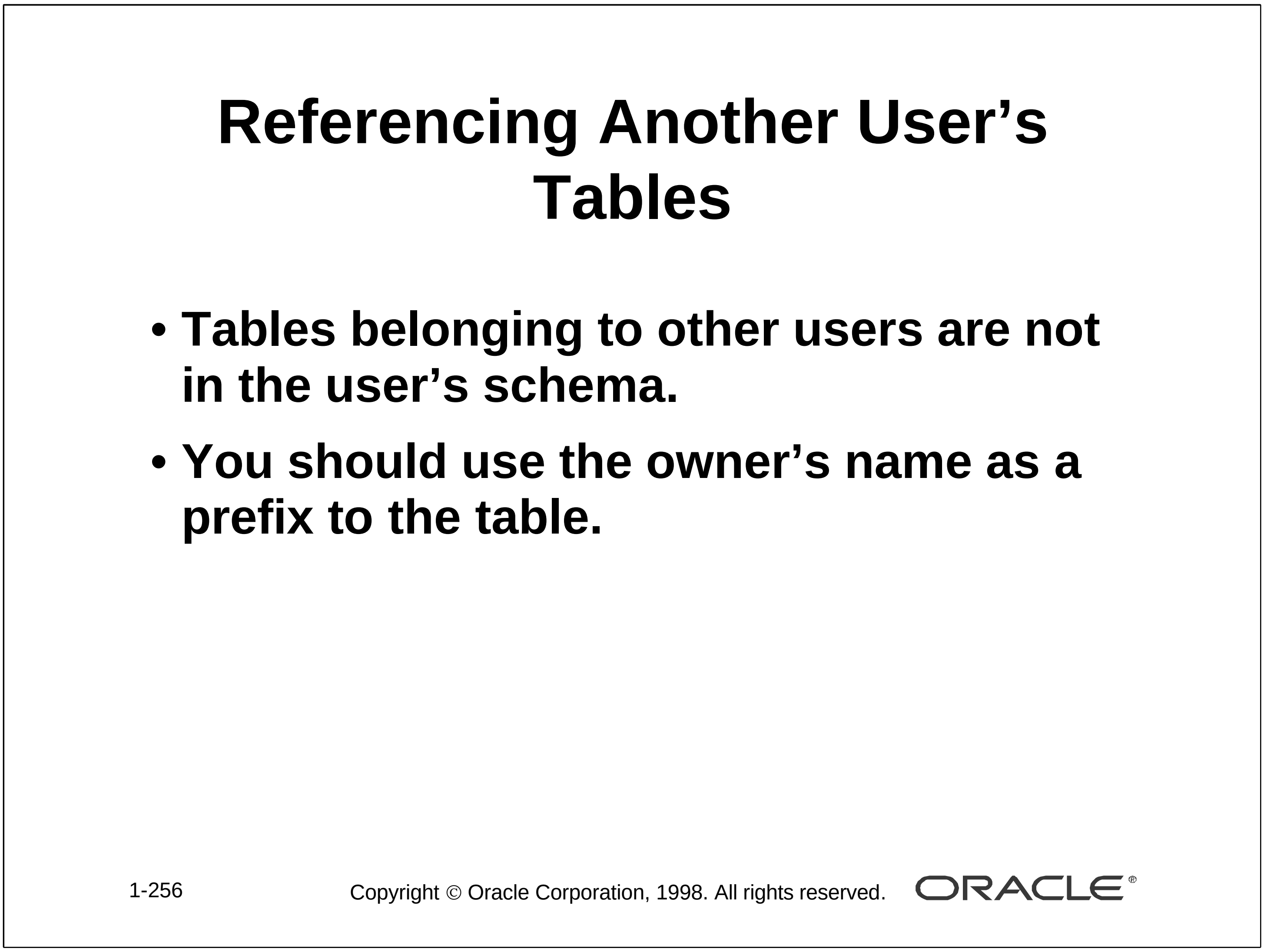
Create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the data definition language (DDL) statements, which you are covered in the next several lessons. DDL statements are a subset of SQL statements used to create, modify, or remove Oracle8 database structures. These statements have an immediate effect on the database, and they also record information in the data dictionary.

To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects. The database administrator uses data control language (DCL) statements, which are covered in a later lesson, to grant privileges to users.

In the syntax:

| | |
|---|---|
| *schema* | is the same as the owner’s name |
| *table* | is the name of the table |
| DEFAULT *expr* | specifies a default value if a value is omitted in the INSERT statement |
| *column* | is the name of the column |
| *datatype* | is the column’s datatype and length |

For more information, see
*Oracle8 Server SQL Reference, Release 8.0*, “CREATE TABLE.”



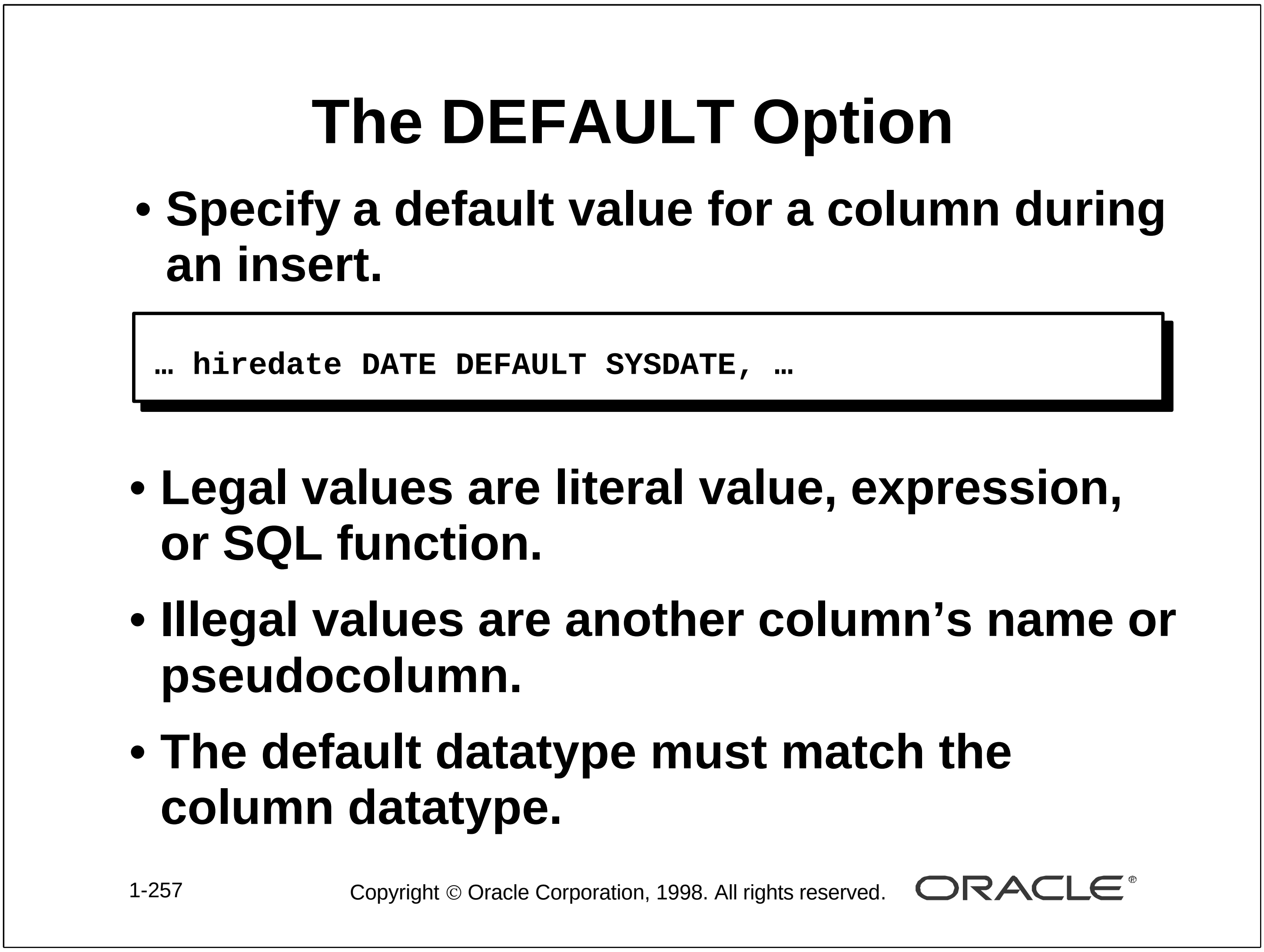
# Referencing Another User’s Tables

- Tables belonging to other users are not in the user’s schema.
- You should use the owner’s name as a prefix to the table.

## Referencing Another User’s Tables

A *schema* is a collection of objects. Schema objects are the logical structures that directly refer to the data in a database. Schema objects include tables, views, synonyms, sequences, stored procedures, indexes, clusters, and database links.

If a table does not belong to the user, the owner’s name must be prefixed to the table.



# The DEFAULT Option

- Specify a default value for a column during an insert.

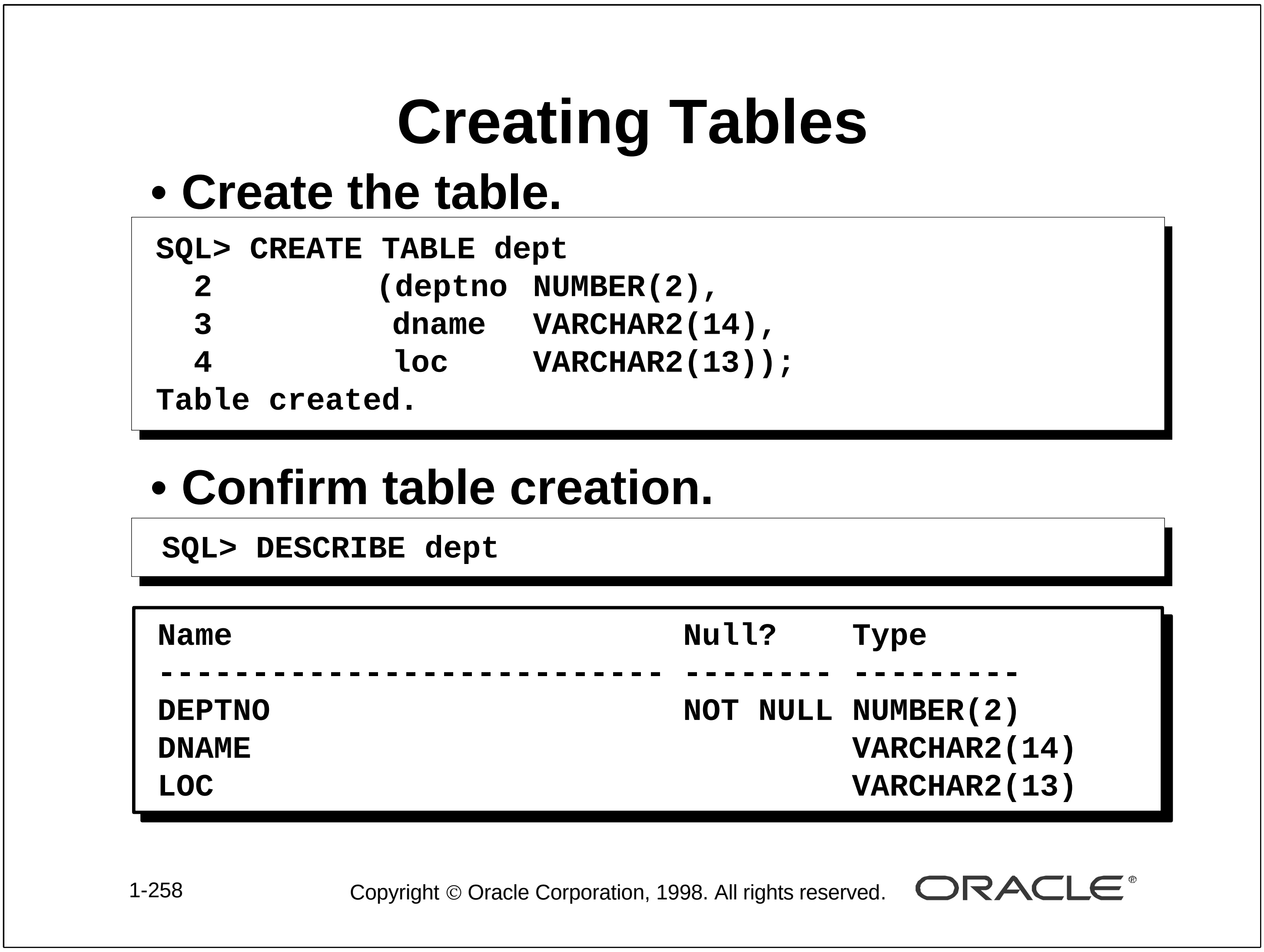
```
… hiredate DATE DEFAULT SYSDATE, …
```

- Legal values are literal value, expression, or SQL function.
- Illegal values are another column’s name or pseudocolumn.
- The default datatype must match the column datatype.

1-257

### The DEFAULT Option

A column can be given a default value by using the DEFAULT option. This option prevents null values from entering the columns if a row is inserted without a value for the column. The default value can be a literal, an expression, or a SQL function, such as SYSDATE and USER, but the value cannot be the name of another column or a pseudocolumn, such as NEXTVAL or CURRVAL. The default expression must match the datatype of the column.



# Creating Tables

- Create the table.

```
SQL> CREATE TABLE dept
  2         (deptno  NUMBER(2),
  3          dname   VARCHAR2(14),
  4          loc     VARCHAR2(13));
Table created.
```

- Confirm table creation.

```
 SQL> DESCRIBE dept
```

```
Name                           Null?    Type
------------------------------ -------- ---------
DEPTNO                         NOT NULL NUMBER(2)
DNAME                                   VARCHAR2(14)
LOC                                     VARCHAR2(13)
```

### Creating Tables

The example above creates the DEPT table, with three columns—namely, DEPTNO, DNAME, and LOC. It further confirms the creation of the table by issuing the DESCRIBE command.

Since creating a table is a DDL statement, an automatic commit takes place when this statement is executed.

# Querying the Data Dictionary

- **Describe tables owned by the user.**

```
SQL> SELECT   *
  2  FROM     user_tables;
```

- **View distinct object types owned by the user.**

```
SQL> SELECT   DISTINCT object_type
  2  FROM     user_objects;
```

- **View tables, views, synonyms, and sequences owned by the user.**

```
SQL> SELECT   *
  2  FROM     user_catalog;
```

### Querying the Data Dictionary

You can query the data dictionary tables to view various database objects owned by you. The data dictionary tables frequently used are these:

- USER_TABLES
- USER_OBJECTS
- USER_CATALOG

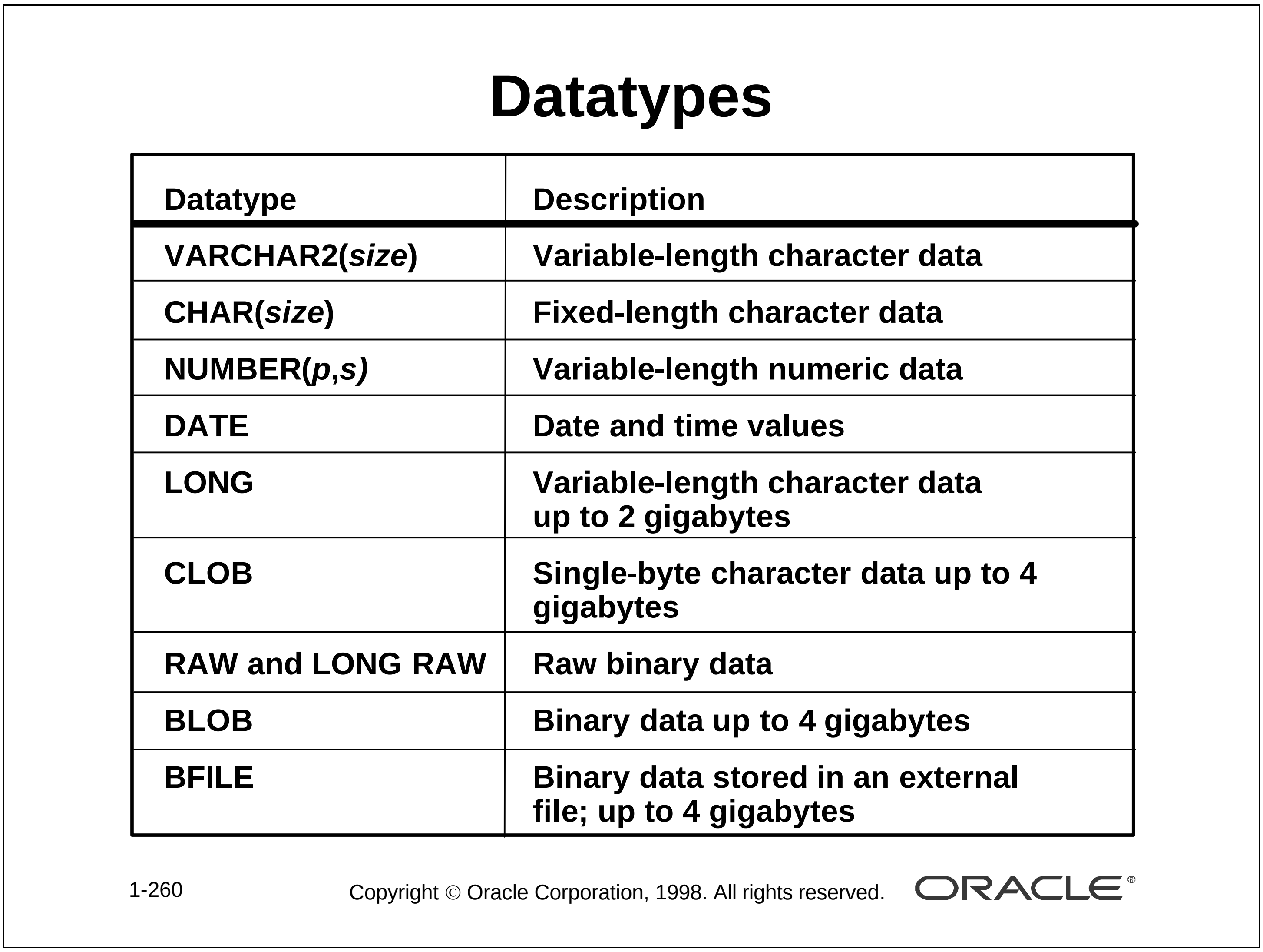
**Note:** USER_CATALOG has a synonym called CAT. You can use this synonym instead of USER_CATALOG in SQL statements.

```
SQL> SELECT *
   2  FROM   CAT;
```

### Class Management Note

(For slide 10-10)

Oracle8 introduces large object (LOB) datatypes that can store large and unstructured data such as text, image, video, and spatial data, up to 4 gigabytes in size.

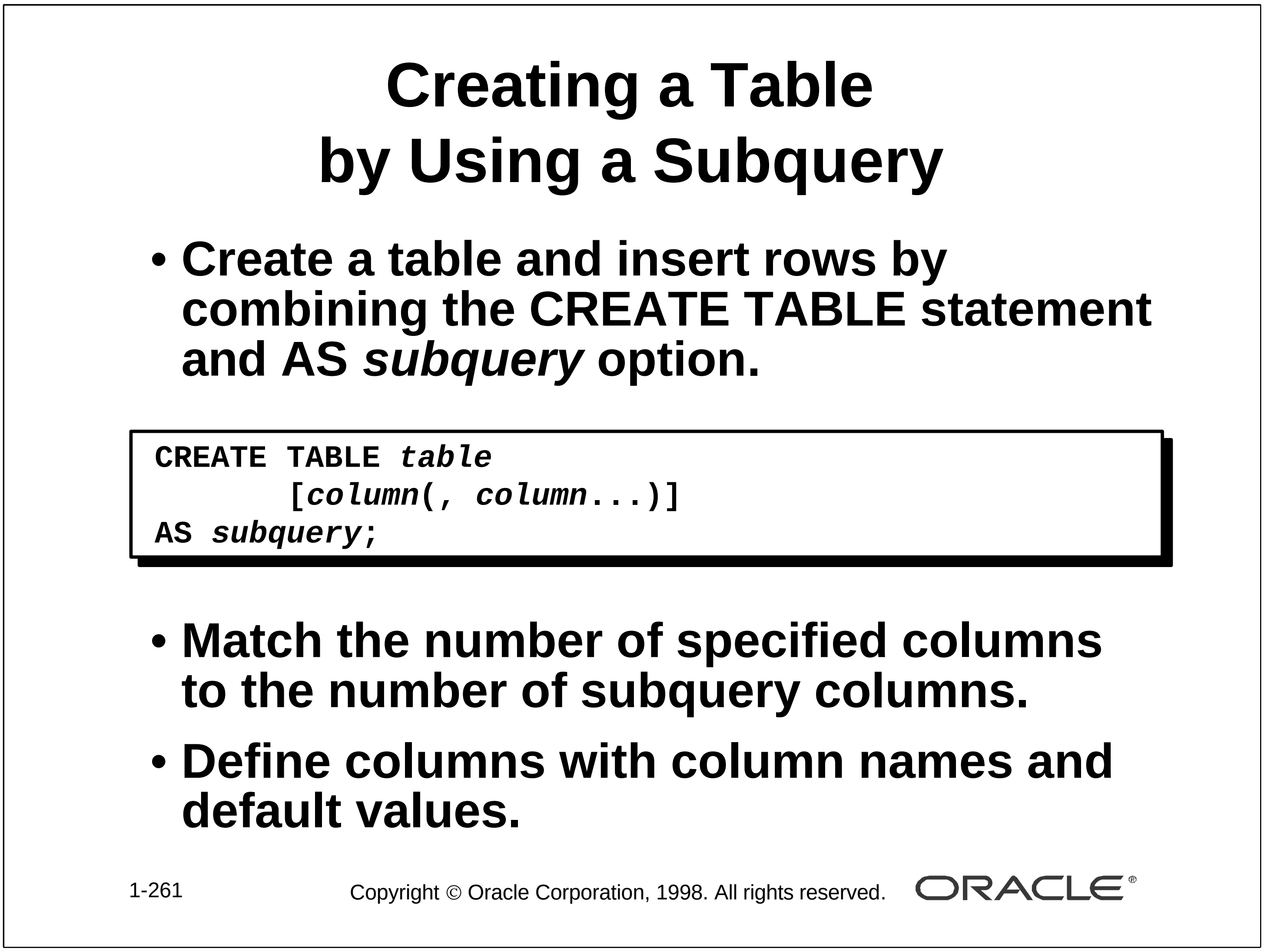


# Datatypes

| Datatype | Description |
|---|---|
| VARCHAR2(*size*) | Variable-length character data |
| CHAR(*size*) | Fixed-length character data |
| NUMBER(*p,s)* | Variable-length numeric data |
| DATE | Date and time values |
| LONG | Variable-length character data up to 2 gigabytes |
| CLOB | Single-byte character data up to 4 gigabytes |
| RAW and LONG RAW | Raw binary data |
| BLOB | Binary data up to 4 gigabytes |
| BFILE | Binary data stored in an external file; up to 4 gigabytes |

**Datatypes**

| **Datatype** | **Description** |
|---|---|
| VARCHAR2(*size*) | Variable-length character data. A maximum *size* must be specified. Default and minimum *size* is 1, maximum *size* is 4000. |
| CHAR(*size*) | Fixed-length character data of length *size* bytes. Default and minimum *size* is 1, maximum *size* is 2000. |
| NUMBER(*p*,*s*) | Number having precision *p* and scale *s*; the precision is the total number of decimal digits, and the scale is the number of digits to the right of the decimal point. The precision can range from 1 to 38 and the scale can range from -84 to 127. |
| DATE | Date and time values between January 1, 4712 B.C., and December 31, 9999 A.D. |
| LONG | Variable-length character data up to 2 gigabytes. |
| CLOB | Single-byte character data up to 4 gigabytes. |
| RAW(*size*) | Raw binary data of length *size*. Maximum *size* is 2000. A maximum size must be specified. |
| LONG RAW | Raw binary data of variable length up to 2 gigabytes. |
| BLOB | Binary data up to 4 gigabytes. |
| BFILE | Binary data stored in an external file; up to 4 gigabytes. |



# Creating a Table by Using a Subquery

- Create a table and insert rows by combining the CREATE TABLE statement and AS *subquery* option.

```
CREATE TABLE table
       [column(, column...)]
AS subquery;
```

- Match the number of specified columns to the number of subquery columns.
- Define columns with column names and default values.

1-261

ORACLE®

### Creating a Table from Rows in Another Table

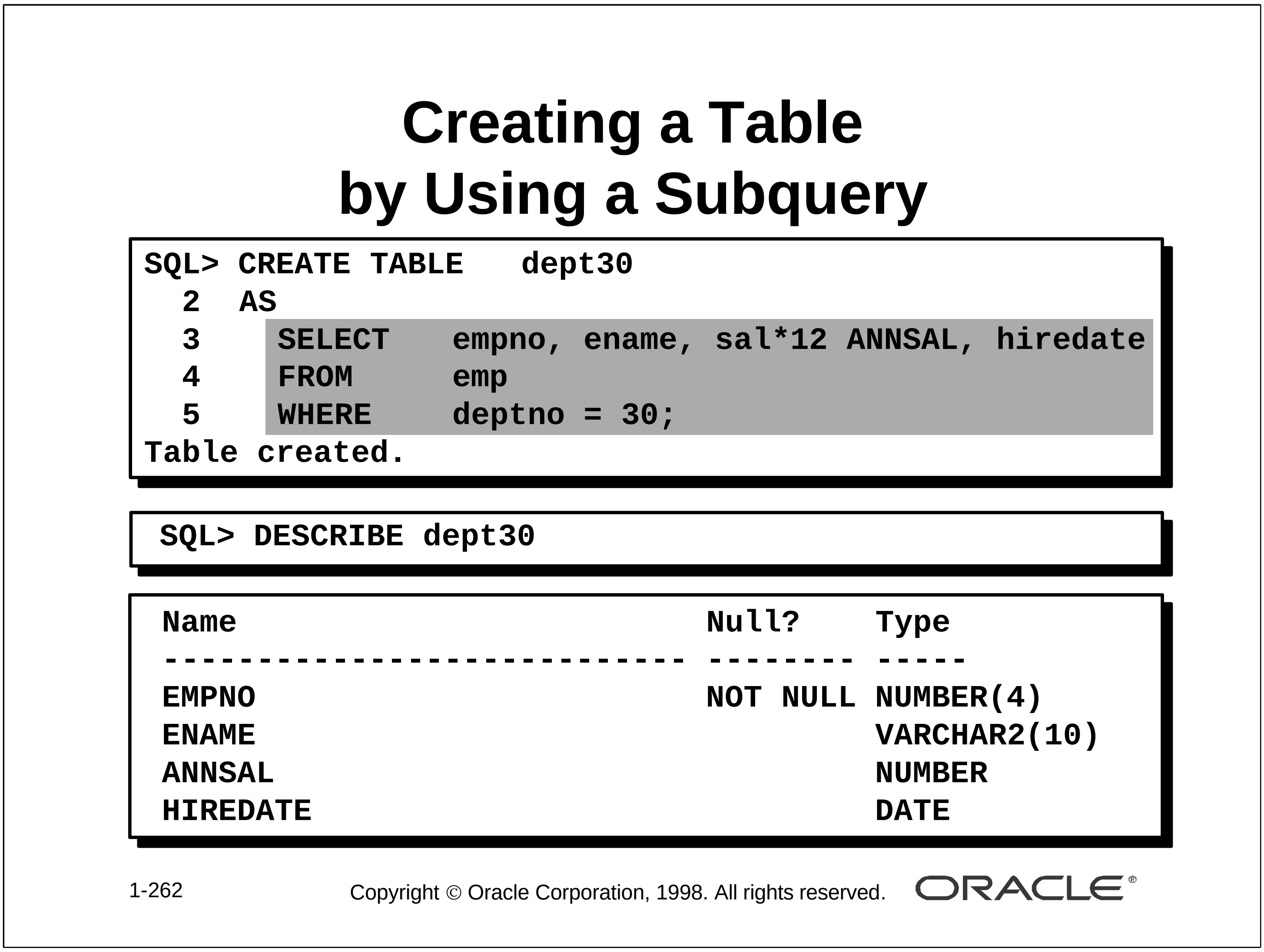
A second method to create a table is to apply the AS *subquery* clause to both create the table and insert rows returned from the subquery.

In the syntax:

| | |
|---|---|
| *table* | is the name of the table |
| *column* | is the name of the column, default value, and integrity constraint |
| *subquery* | is the SELECT statement that defines the set of rows to be inserted into the new table |

### Guidelines

- The table will be created with the specified column names, and the rows retrieved by the SELECT statement will be inserted into the table.
- The column definition can contain only the column name and default value.
- If column specifications are given, the number of columns must equal the number of columns in the subquery SELECT list.
- If no column specifications are given, the column names of the table are the same as the column names in the subquery.



# Creating a Table by Using a Subquery

```
SQL> CREATE TABLE   dept30
  2  AS
  3    SELECT   empno, ename, sal*12 ANNSAL, hiredate
  4    FROM     emp
  5    WHERE    deptno = 30;
Table created.
```

```
SQL> DESCRIBE dept30
```

```
Name                            Null?    Type
---------------------------- -------- -----
EMPNO                           NOT NULL NUMBER(4)
ENAME                                    VARCHAR2(10)
ANNSAL                                   NUMBER
HIREDATE                                 DATE
```

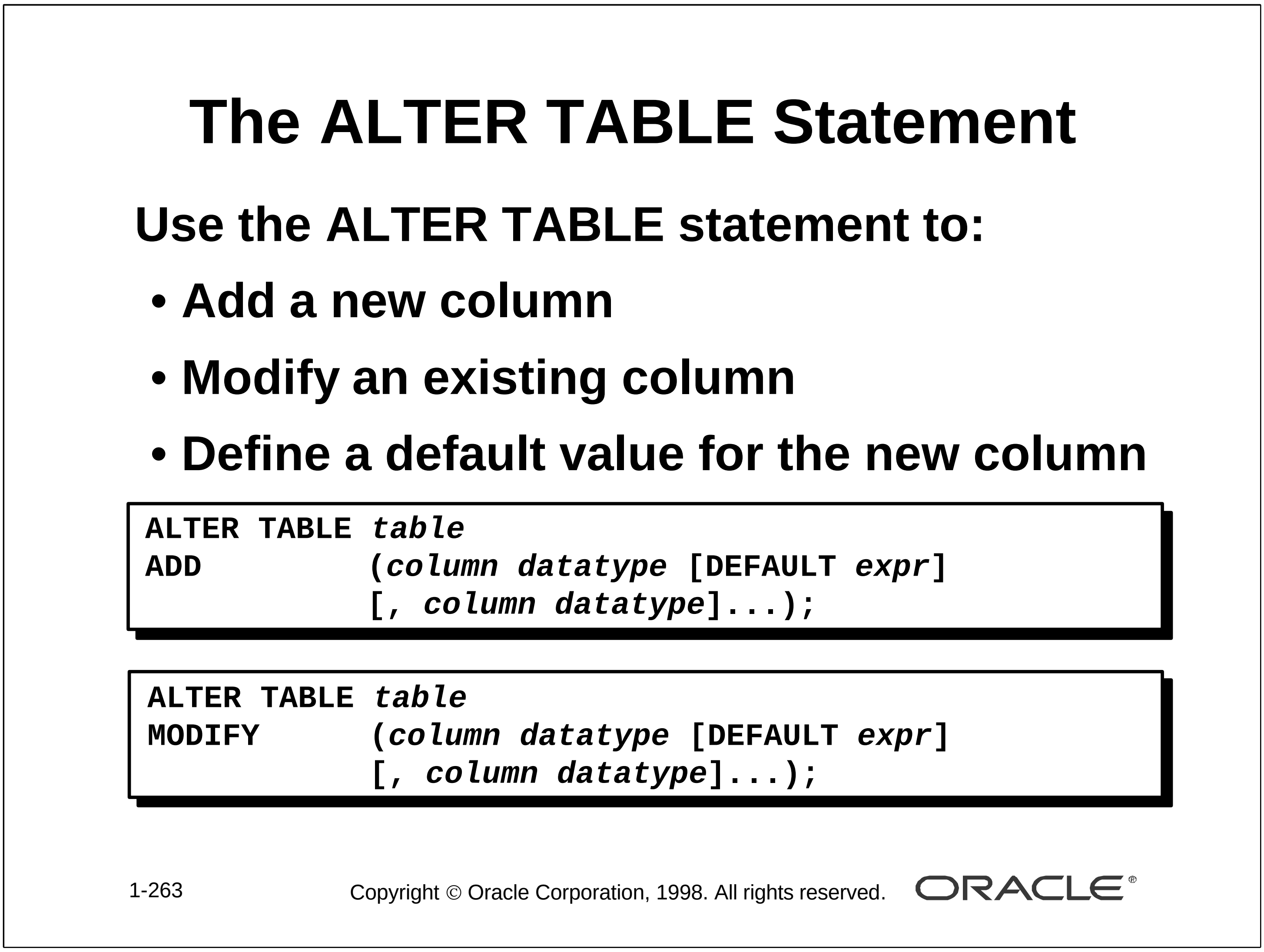
1-262

### Creating a Table from Rows in Another Table

The example above creates a table, DEPT30, that contains details of all the employees working in department 30. Notice that the data for the DEPT30 table is coming from the EMP table.

You can verify the existence of a database table and check column definitions by using the SQL*Plus DESCRIBE command.

Give a column alias, when selecting an expression.



# The ALTER TABLE Statement

## Use the ALTER TABLE statement to:

- Add a new column
- Modify an existing column
- Define a default value for the new column

```
ALTER TABLE table
ADD         (column datatype [DEFAULT expr]
            [, column datatype]...);
```

```
ALTER TABLE table
MODIFY      (column datatype [DEFAULT expr]
            [, column datatype]...);
```

1-263

### ALTER TABLE Statement

After you create your tables, you may need to change the table structures because you omitted a column or your column definition needs to be changed. You can do this by using the ALTER TABLE statement.

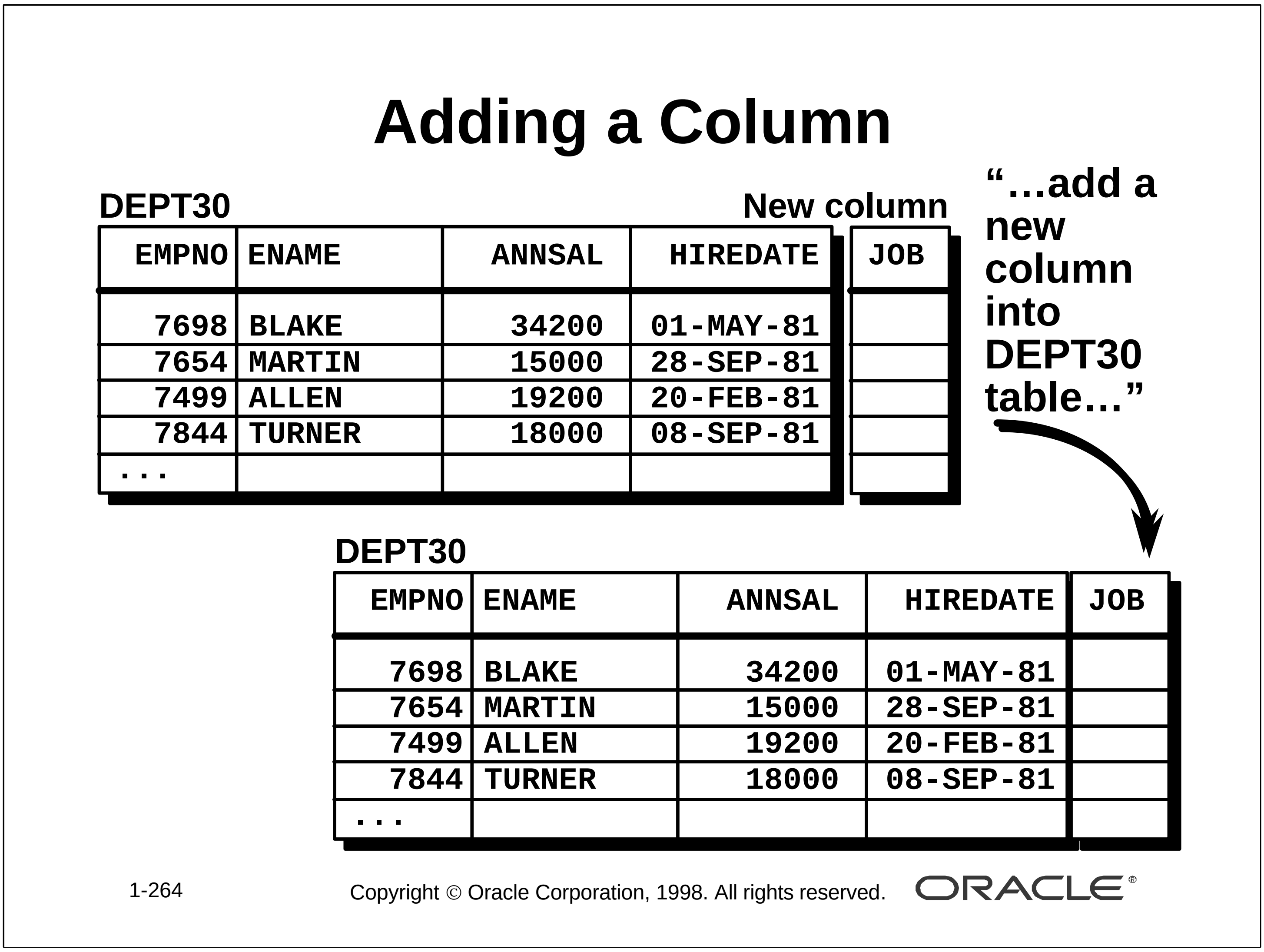
You can add columns to a table by using the ALTER TABLE statement with the ADD clause.

In the syntax:

| | |
|---|---|
| *table* | is the name of the table |
| *column* | is the name of the new column |
| *datatype* | is the datatype and length of the new column |
| DEFAULT *expr* | specifies the default value for a new column |

You can modify existing columns in a table by using the ALTER TABLE statement with the MODIFY clause.

**Note:** The slide gives the abridged syntax for ALTER TABLE. More about ALTER TABLE is covered in lesson 11.



# Adding a Column

DEPT30

New column

| EMPNO | ENAME | ANNSAL | HIREDATE | JOB |
|---|---|---|---|---|
| 7698 | BLAKE | 34200 | 01-MAY-81 | |
| 7654 | MARTIN | 15000 | 28-SEP-81 | |
| 7499 | ALLEN | 19200 | 20-FEB-81 | |
| 7844 | TURNER | 18000 | 08-SEP-81 | |
| ... | | | | |

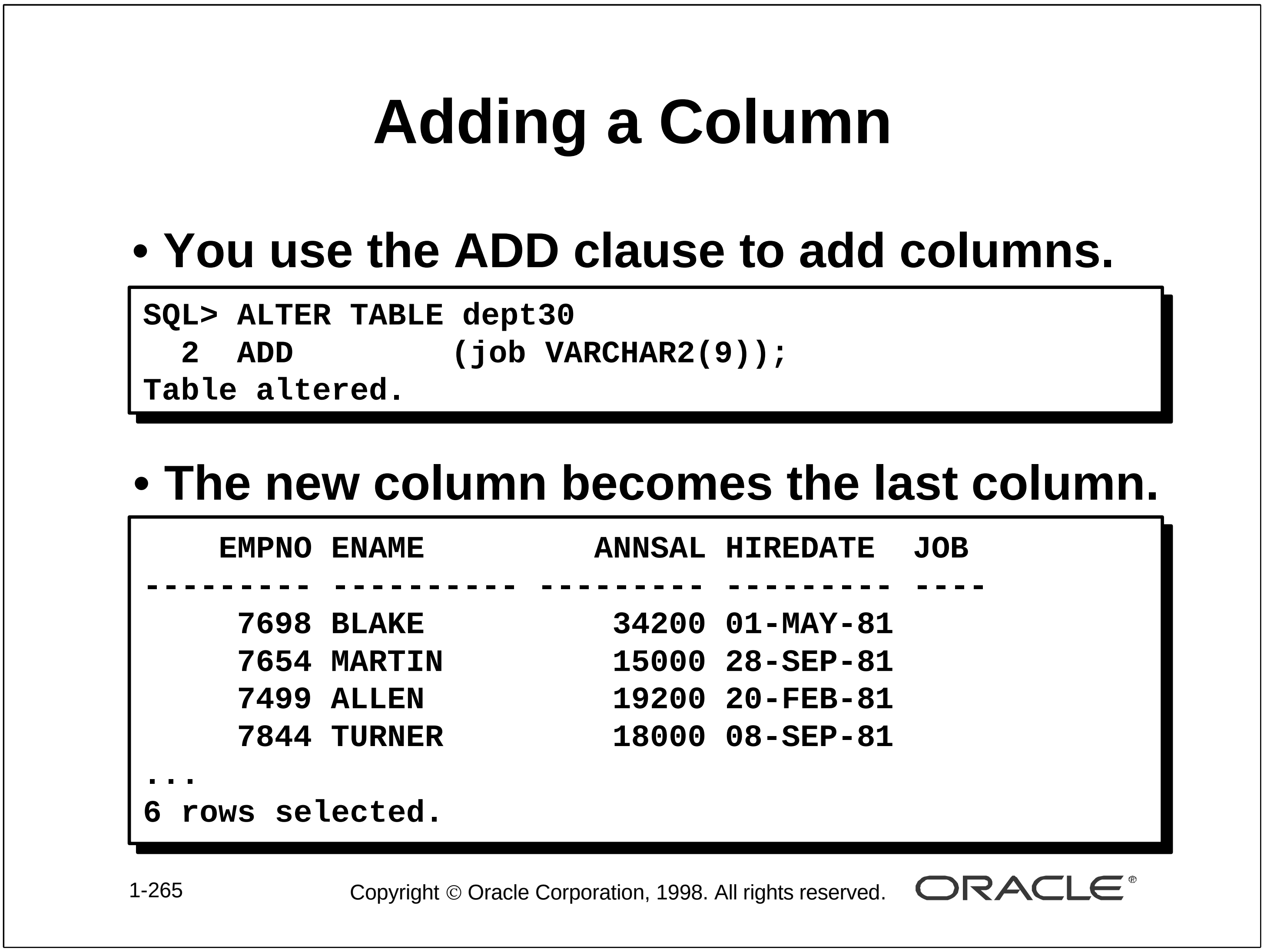
"…add a new column into DEPT30 table…"

DEPT30

| EMPNO | ENAME | ANNSAL | HIREDATE | JOB |
|---|---|---|---|---|
| 7698 | BLAKE | 34200 | 01-MAY-81 | |
| 7654 | MARTIN | 15000 | 28-SEP-81 | |
| 7499 | ALLEN | 19200 | 20-FEB-81 | |
| 7844 | TURNER | 18000 | 08-SEP-81 | |
| ... | | | | |

### Adding a Column

The graphic adds the JOB column to DEPT30 table. Notice that the new column becomes the last column in the table.



# Adding a Column

- You use the ADD clause to add columns.

```
SQL> ALTER TABLE dept30
  2  ADD        (job VARCHAR2(9));
Table altered.
```

- The new column becomes the last column.

```
    EMPNO ENAME         ANNSAL HIREDATE  JOB
--------- ---------- --------- --------- ----
     7698 BLAKE          34200 01-MAY-81
     7654 MARTIN         15000 28-SEP-81
     7499 ALLEN          19200 20-FEB-81
     7844 TURNER         18000 08-SEP-81
...
6 rows selected.
```

1-265

ORACLE®

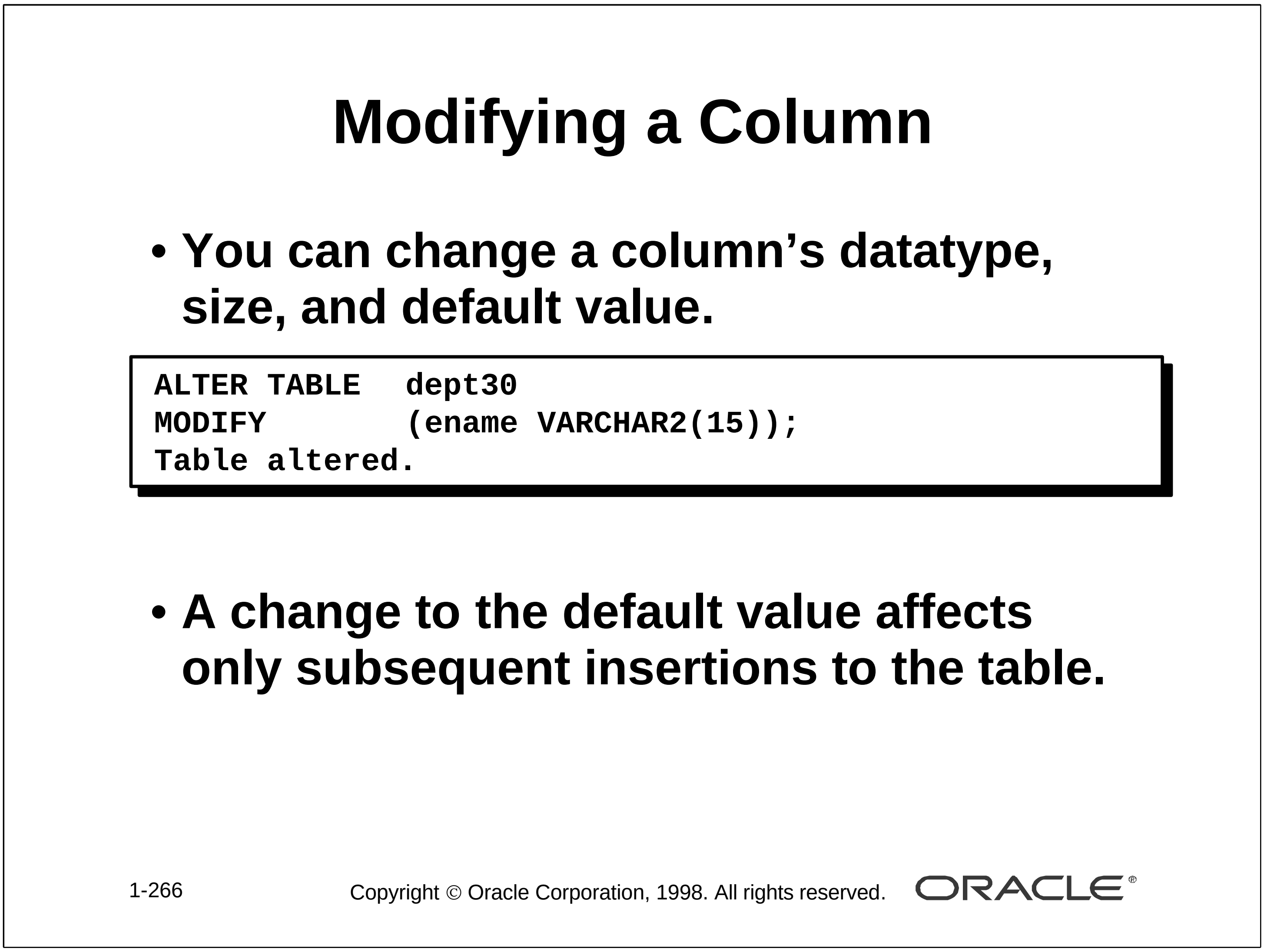
### Guidelines for Adding a Column

- You can add or modify columns, but you cannot drop them from a table.
- You cannot specify where the column is to appear. The new column becomes the last column.

The example above adds a column named JOB to the DEPT30 table. The JOB column becomes the last column in the table.

**Note:** If a table already contains rows when a column is added, then the new column is initially null for all the rows.

You can define a NOT NULL column only if the table contains no rows because data cannot be specified for existing rows at the same time that the column is added.



# Modifying a Column

- You can change a column’s datatype, size, and default value.

```
ALTER TABLE   dept30
MODIFY        (ename VARCHAR2(15));
Table altered.
```

- A change to the default value affects only subsequent insertions to the table.

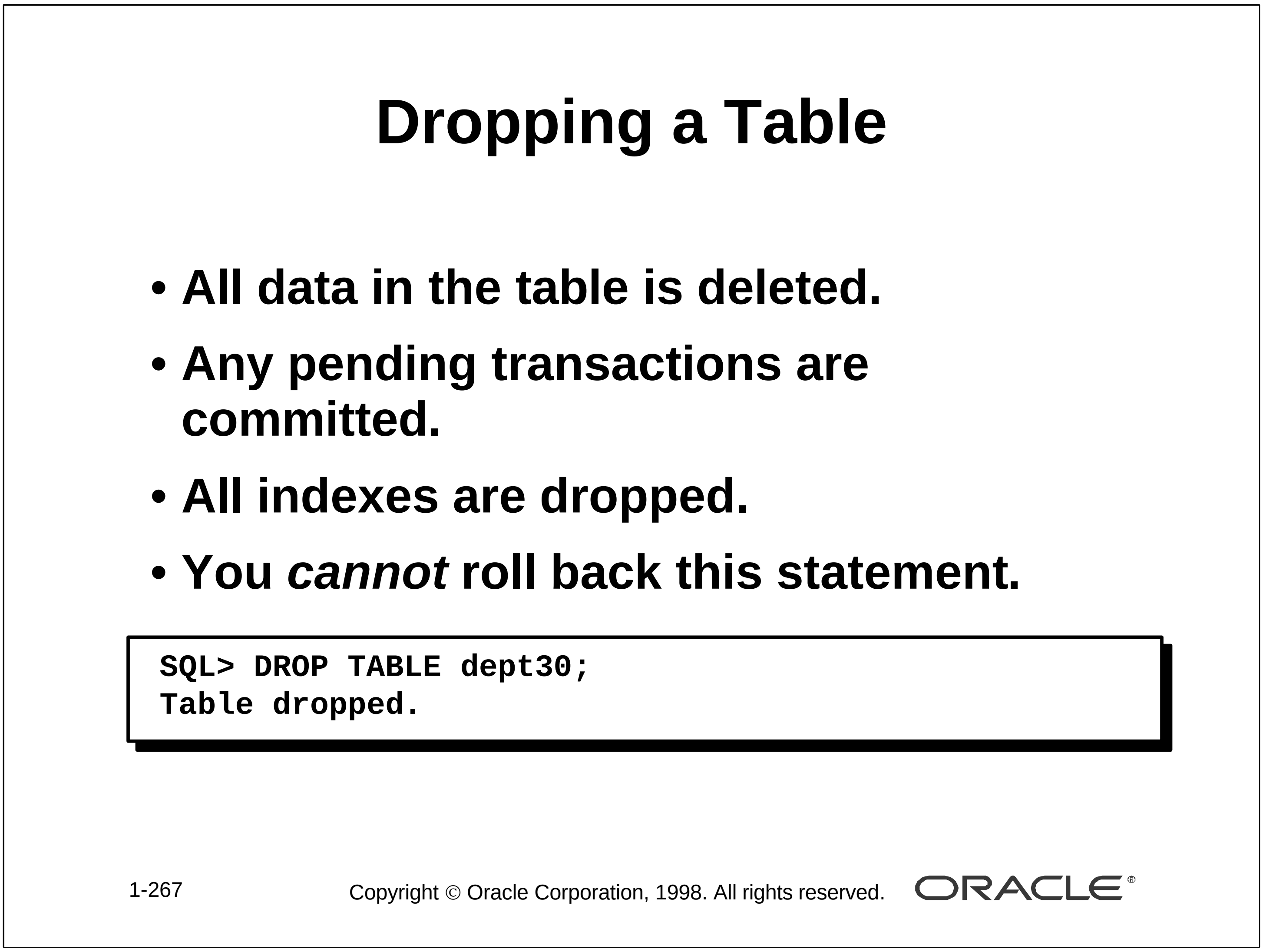
1-266

### Modifying a Column

You can modify a column definition by using the ALTER TABLE statement with the MODIFY clause. Column modification can include changes to a column’s datatype, size, and default value.

### Guidelines

- Increase the width or precision of a numeric column.
- Decrease the width of a column if the column contains only null values or if the table has no rows.
- Change the datatype if the column contains null values.
- Convert a CHAR column to the VARCHAR2 datatype or convert a VARCHAR2 column to the CHAR datatype if the column contains null values or if you do not change the size.
- A change to the default value of a column affects only subsequent insertions to the table.



# Dropping a Table

- All data in the table is deleted.
- Any pending transactions are committed.
- All indexes are dropped.
- You *cannot* roll back this statement.

```
SQL> DROP TABLE dept30;
Table dropped.
```

1-267

### Dropping a Table

The DROP TABLE statement removes the definition of an Oracle8 table. When you drop a table, the database loses all the data in the table and all the indexes associated with it.

### Syntax

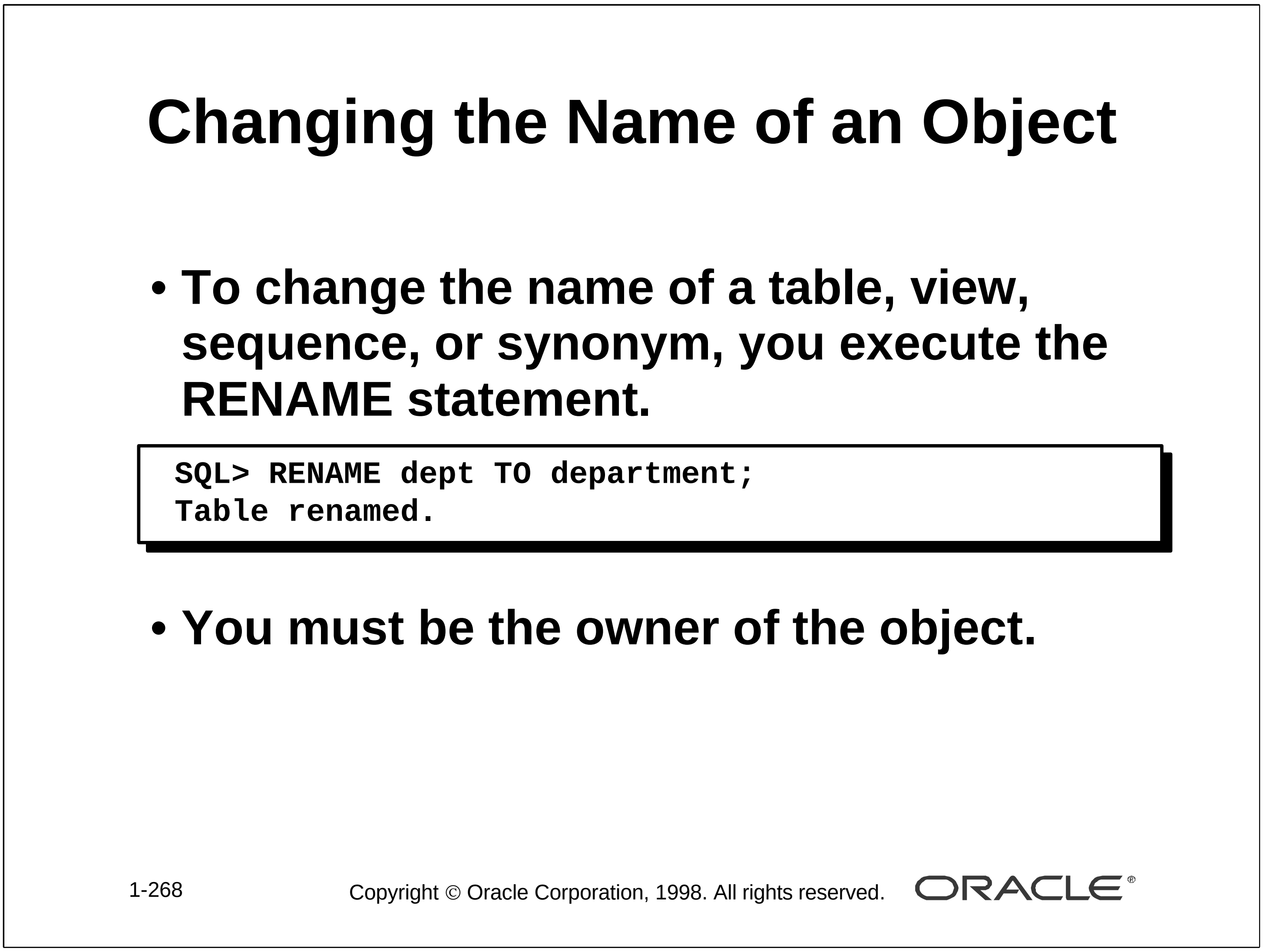
```
DROP TABLE table;
```

**where:** *table* is the name of the table

### Guidelines

- All data is deleted from the table.
- Any views, synonyms, stored procedures, functions, and packages will remain but are invalid.
- Any pending transactions are committed.
- Only the creator of the table or a user with the DROP ANY TABLE privilege can remove a table.

The DROP TABLE statement, once executed, is irreversible. The Oracle8 Server does not question the action when you issue the DROP TABLE statement. If you own that table or have a high-level privilege, then the table is immediately removed. All DDL statements issue a commit, therefore making the transaction permanent.



# Changing the Name of an Object

- To change the name of a table, view, sequence, or synonym, you execute the RENAME statement.

```
SQL> RENAME dept TO department;
Table renamed.
```

- You must be the owner of the object.

### Renaming a Table

Additional DDL statements include the RENAME statement, which is used to rename a table, view, sequence, or a synonym.

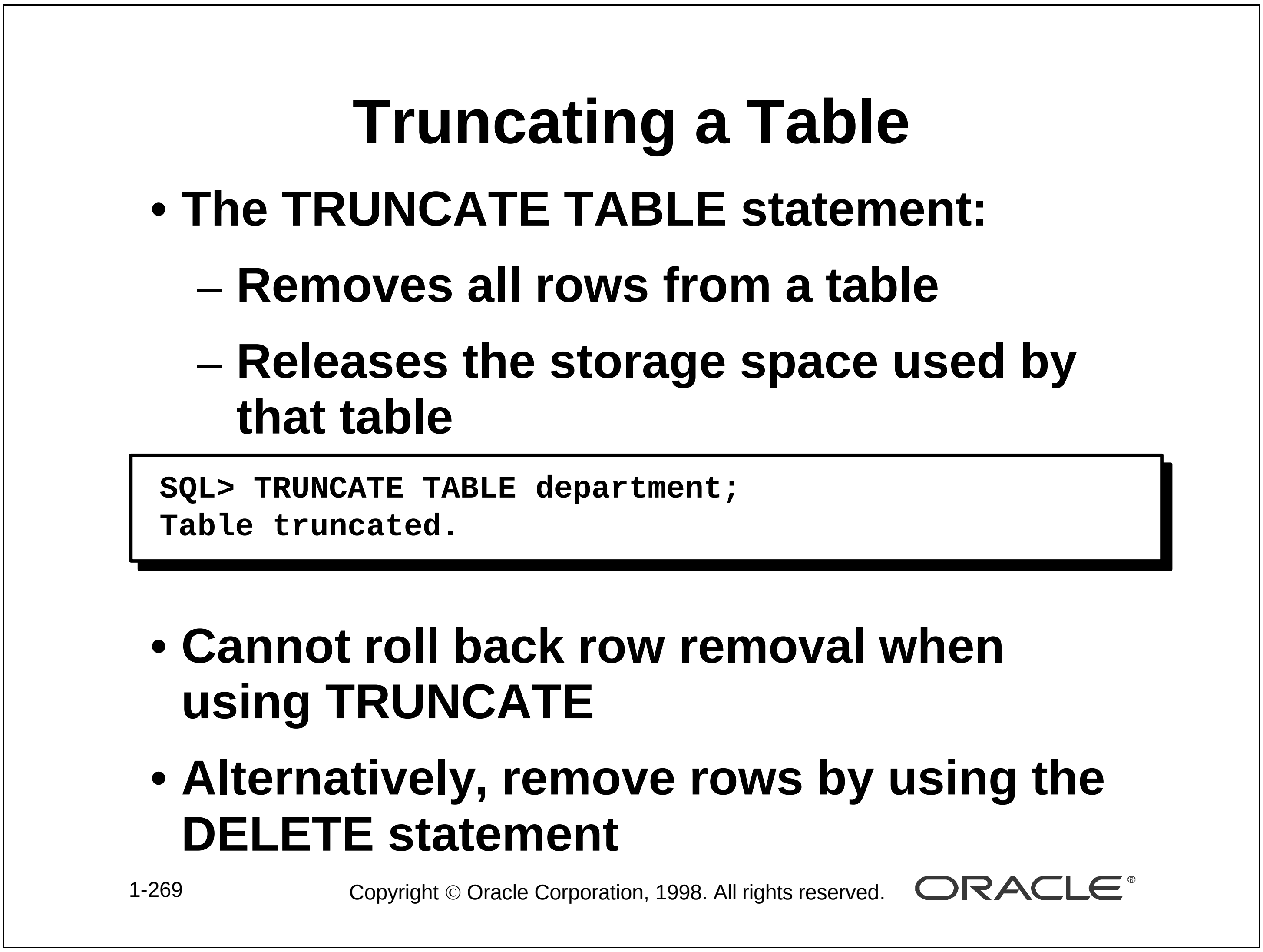
### Syntax

```
RENAME    old_name  TO  new_name;
```

**where:** *old_name* is the old name of the table, view, sequence, or synonym

*new_name* is the new name of the table, view, sequence, or synonym

You must be the owner of the object you rename.



# Truncating a Table

- The TRUNCATE TABLE statement:
  - Removes all rows from a table
  - Releases the storage space used by that table

```
SQL> TRUNCATE TABLE department;
Table truncated.
```

- Cannot roll back row removal when using TRUNCATE
- Alternatively, remove rows by using the DELETE statement

### Truncating a Table

Another DDL statement is the TRUNCATE TABLE statement, which is used to remove all rows from a table and to release the storage space used by that table. When using the TRUNCATE TABLE statement, you cannot rollback row removal.

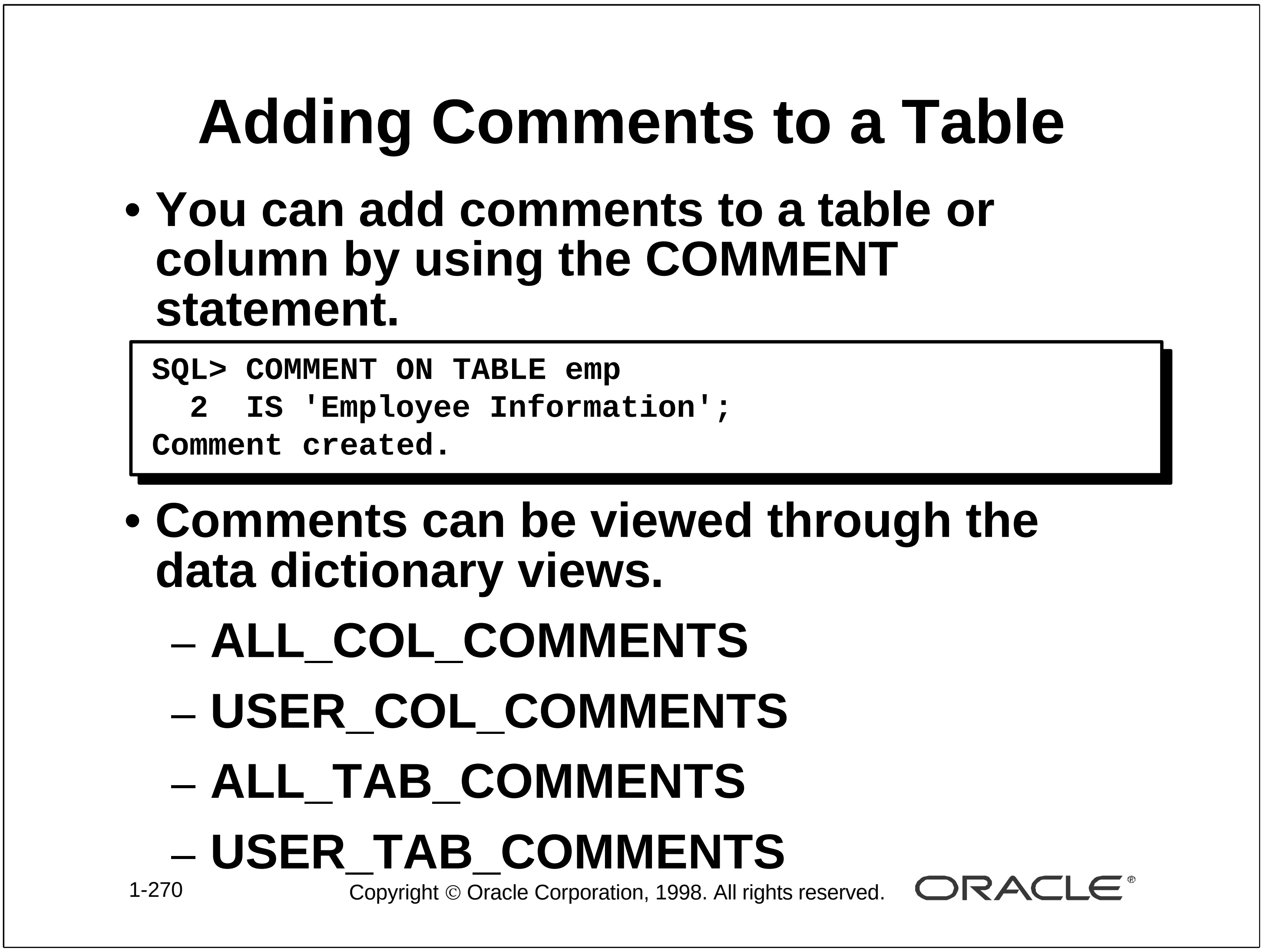
### Syntax

```
TRUNCATE  TABLE   table;
```

**where:** *table* is the name of the table

You must be the owner of the table or have DELETE TABLE system privileges to truncate a table.

The DELETE statement can also remove all rows from a table, but it does not release storage space.



# Adding Comments to a Table

- You can add comments to a table or column by using the COMMENT statement.

```
SQL> COMMENT ON TABLE emp
  2  IS 'Employee Information';
Comment created.
```

- Comments can be viewed through the data dictionary views.
  - ALL_COL_COMMENTS
  - USER_COL_COMMENTS
  - ALL_TAB_COMMENTS
  - USER_TAB_COMMENTS

### Adding a Comment to a Table

You can add a comment of up to 2000 bytes about a column, table, view, or snapshot by using the COMMENT statement. The comment is stored in the data dictionary and can be viewed in one of the following data dictionary views in the COMMENTS column:

- ALL_COL_COMMENTS
- USER_COL_COMMENTS
- ALL_TAB_COMMENTS
- USER_TAB_COMMENTS

### Syntax

```
COMMENT ON TABLE table | COLUMN table.column
         IS 'text';
```

**where:** *table* is the name of the table
*column* is the name of the column in a table
*text* is the text of the comment

You can drop a comment from the database by setting it to empty string (‘ ’).

```
SQL> COMMENT ON TABLE  emp IS ' ';
```

# Summary

| Statement | Description |
|---|---|
| CREATE TABLE | Creates a table |
| ALTER TABLE | Modifies table structures |
| DROP TABLE | Removes the rows and table structure |
| RENAME | Changes the name of a table, view, sequence, or synonym |
| TRUNCATE | Removes all rows from a table and releases the storage space |
| COMMENT | Adds comments to a table or view |

**CREATE TABLE**

- You can create a table.
- Create a table based on another table by using a subquery.

**ALTER TABLE**

- Modify table structures.
- Change column widths, change column datatypes, and add columns.

**DROP TABLE**

- Remove rows and a table structure.
- Once executed, this statement cannot be rolled back.

**RENAME**

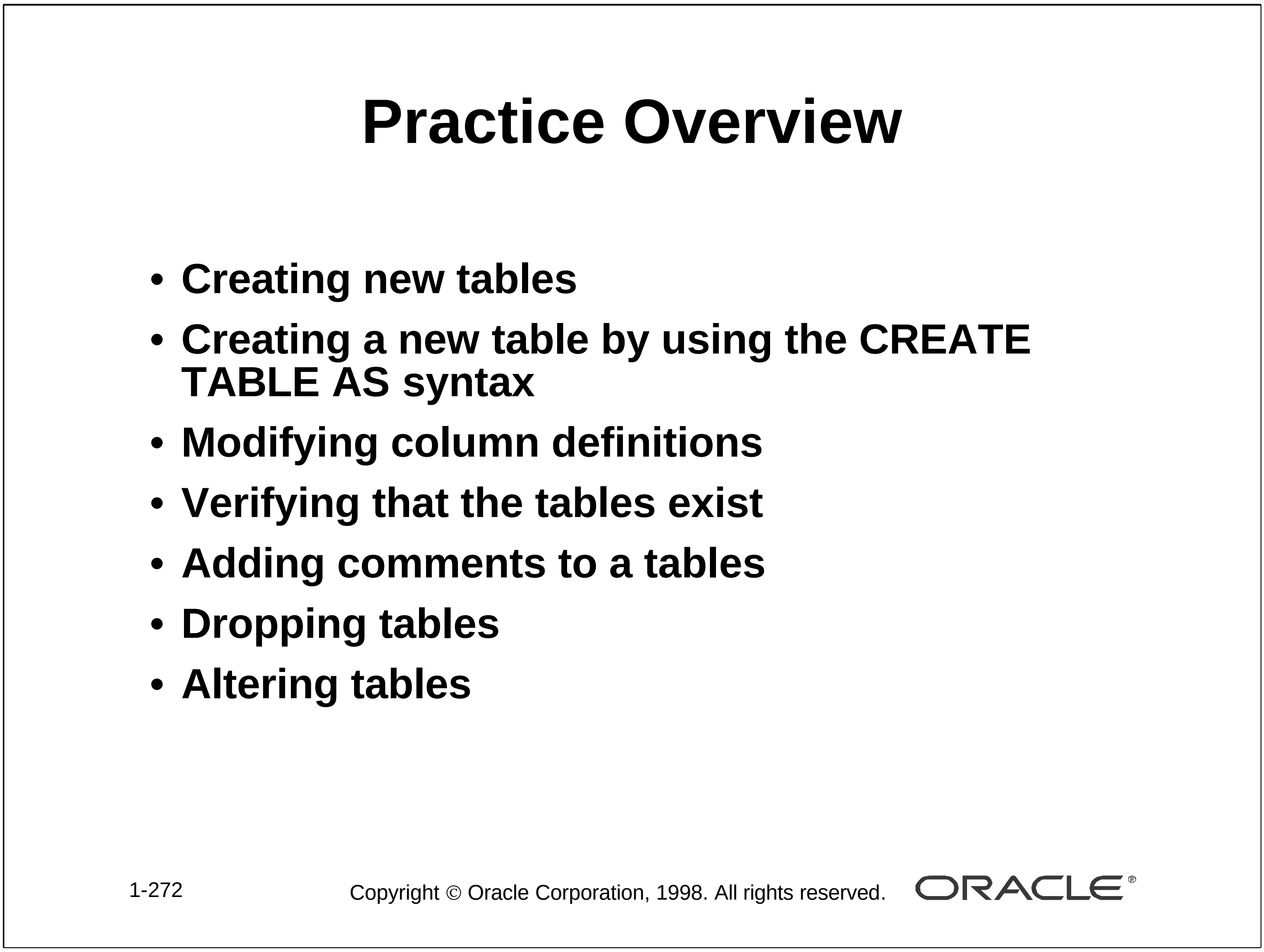
- Rename a table, view, sequence, or synonym.

**TRUNCATE**

- Remove all rows from a table and release the storage space used by the table.
- DELETE statement only removes rows.

**COMMENT**

- Add a comment to a table or a column.
- Query the data dictionary to view the comment.



# Practice Overview

- Creating new tables
- Creating a new table by using the CREATE TABLE AS syntax
- Modifying column definitions
- Verifying that the tables exist
- Adding comments to a tables
- Dropping tables
- Altering tables

### Practice Overview

Create new tables containing constraints by using the CREATE TABLE statement. Confirm that the new table was added to the database. Create the syntax in the command file, and then execute the command file to create the table.